Lutz Wegner

Programmierung graphischer Benutzerschnittstellen mit Tcl/Tk

Universität
Kassel

SKRIPTEN DER
PRAKTISCHEN INFORMATIK

Lutz Wegner

Universität Kassel

Fachbereich Elektrotechnik/Informatik

D-34121 Kassel

4. Auflage April 2009

Vorwort

Tcl/Tk ist die Kombination einer einfachen, eleganten Skriptsprache (Tcl steht für Tool command language, ausgesprochen wie in „tickle“) und einer Bibliothek zur Erstellung einer graphischen Benutzeroberfläche (Tk: Tool kit). Tcl wurde von John Ousterhout, damals an der UC at Berkeley, 1987/88 entwickelt, Tk wurde um 1991 herum fertiggestellt.

Die Sprache und der Baukasten für die X11-Oberfläche wurden schnell in der UNIX-Gemeinde populär. In einem Interview aus dem Jahr 1999 erklärt Brian Kernighan, der Vater von UNIX und AWK, sowie einer der Geburtshelfer von C, zur Rolle von Tcl/Tk:

> Much of my programming in the past few years has been in Tcl/Tk, since I’ve been doing user interfaces for several different systems, (...). I’ve also done experimental versions of some of these in Java and Visual Basic. I use AWK for simple scripting where I probably would have used the shell in years past. [15]

Für Tcl/Tk gibt es auch eine gut aufgebaute Interprozeßkommunikationsbibliothek, etwa für Socket-Verbindungen, so daß sich auch einfach und schnell Client-Server-Systeme aufbauen lassen.

Trotzdem ist das Interesse an Tcl/Tk seit ca. 2001 etwas rückläufig. Das hängt mit der Dominanz von Java zusammen, das auch für Web-Anwendungen in Form von Applets praktisch überall zu finden ist. Zwar gibt es auch die Tcl-Variante Safe-Tcl als mobilen Code, die sog. Tclets, doch zögern verständlicherweise viele Benutzer, in ihren Browser ein weiteres Plug-in, hier für Tclets, zu laden.

Zweitens ist mit Python eine weitere Skriptsprache entstanden, die sich großer Beliebtheit erfreut, daneben ist auch PHP mit etwas anderen Anwendungsfeldern sehr populär.

Insgesamt lohnt sich das Erlernen von Tcl/Tk in jedem Fall, weil es ein Musterbeispiel für den modularen Aufbau einer graphischen Umgebung ist und Tcl mit einem minimalistischen Typsystem, anders als etwa im XML-Umfeld, sehr portabel bleibt.

Das Skript beginnt mit drei Kapiteln, die nochmals für die Vorlesung im Sommer 2006 überarbeitet wurden, um eine bessere Einführung in Tcl zu bieten. Dafür sind Ausführungen in allgemeiner Form zu graphischen Oberflächen und Fensterssystemen, die auf dem Buch von Klingert [5] beruhen, stärker gekürzt worden. Der Einstieg in Tcl erfolgt auf der Basis des Buches von Ousterhout [7] und frei verfügbaren Tutorials, u.a. dem von Wolfram Schenck an der Universität Bielefeld. Der Rest des Skripts lehnt sich sehr eng an das ausgezeichnete Werk von Harrison und McLennan [12] an.

Nützliche Adressen im Web

- http://www.db.informatik.uni-kassel.de
  Unter dem Menüpunkt „Lehre“ finden Sie die Web-Seite dieser Vorlesung, unter „Lehrmaterial“ einen Link zur Tcl/Tk-Dokumentation.
- http://www.softpanorama.org/
  Scripting/tcl_tk_expect.shtml und
  http://www.tcl.tk/doc/
  Tutorials und viele nützliche Informationen zu Tcl/Tk.
- http://www.awprofessional.com/title/0201634740
  Web-Seite von Addison-Wesley mit dem Quellcode für die Beispiele zum Buch von Harrison&McLennan.

Nachtrag zur 4. Auflage April 2009: Die Neuauflage des Klassikers von Welch und Brent [11] und das – allerdings wenig empfehlenswerte – Einsteigerbuch von Kurt Wall [19] bestätigen das weiterhin bestehende Interesse an Tcl/Tk. Das Skript selbst blieb ansonsten unverändert.

Inhalt

1 Einleitung

Warum sollte man sich mit graphischen Benutzungsschnittstellen (graphical user interfaces, GUIs) beschäftigen?

- offensichtliche Verbreitung (UNIX, Windows, Mac)
- große Bedeutung für die Anwendungsentwicklung (Brad Myers [16] schätzte schon 1995 den Anteil der Entwicklungskosten für die Benutzerschnittstelle auf 50% der gesamten Anwendungsentwicklung)
- nochmalige Zunahme durch das Web mit Web-Browsern als universelle Schnittstelle
- Bedeutung auch für *embedded applications*
- kommerzielle Bedeutung: schon in den Neunzigern ein Milliarden-Markt für *visual development tools*
- mit z. B. Tcl/Tk und Java ausgereifte Produkte auf dem Markt
- gewisse reizvolle ästhetische Aspekte

Wie sollte man sich mit GUIs (*graphical user interfaces*) beschäftigen?

- historische und grundsätzliche Aspekte herausarbeiten
- Fallstudien mit OSF/Motif, X-Windows, Tcl/Tk, Java, ...
- selbst kleine Anwendungen schreiben

☞ Gefahr sich in (schnell veraltenden) produktspezifischen Details zu verlieren, zeitaufwendig!

Nach Klingert [5] spielen die folgenden Aspekte bei *graphischen Fenstersysteme* eine Rolle:

- direktere Abbildung der Vorgänge im Rechner auf die menschliche Gedankenwelt als bei textbasierten Dialogsysteme durch
  - visuelle Ausgaben (Bsp. Uhr, Papierkorb)
  - Parallelität durch simultane Fenster
- enge Verzahnung mit dem Betriebssystem (ein bedeutendes Produkt eines Herstelles heißt sogar so)

Das Wissenschaftsgebiet, das sich mit der Dialogsteuerung und der Ergonomie der Schnittstelle beschäftigt, heißt meist HCI: *human-computer-interaction*, bzw. CHI, allgemeiner auch *Mensch-Maschine-Schnittstelle*.

Wie sind GUIs zu entwickeln?

- wie für jede Art von portabler Software: durch Definition einer *logischen Maschine* [5],
- die Befehle (Methoden) und Datenobjekte zur Verfügung stellt.
- generisch, damit unabhängig von spez. Anforderungen eines Anwendungsprogramms
- in der Regel mittels einer Bibliothek

1.1 Historie

- *Xerox Star* (1981) gilt allgemein als erstes System mit einer graphischen Benutzungsschnittstelle
- erste Ideen gehen zurück auf *Vannevar Bush* (1945) und auch *Zuses* erste Rechner (z. B. Z4) hatten Plotter
- *Ivan E. Sutherland* entwickelte Anfang der 60er Jahre am MIT *Sketchpad* mit direktmanipulativen Elementen (Click-and-Drag) und mit Lichtgriffel
- die *Maus* wurde 1966 am Stanford Research Institute (SRI) durch Doug Engelbart erfunden und unter dem Namen "X-Y Position Indicator for a Display System" patentiert. 1968 zeigte Engelbart sie mit Elementen eines Hypermedia und Videokonferenz-Systems auf der Fall Joint Computer Conference in San Francisco.

1.2 Dynabook, Xerox Star und Alan Kay

Grundidee der direkten Manipulation und des sofortigen Reagierens (*direct manipulation* und *immediate response/feedback*)

> Kay and Goldberg [4], S. 32:
> „There should not be a discernible pause between cause and effect. One of the metaphors we used when designing such a system was that of a musical instrument, such as a flute, which is owned by its user and responds instantly and consistently to its owner's wishes. Imagine the absurdity of a one-second delay between blowing a note and hearing it!"

- *Dynabook* war Anfang der 70er Alan Kays Idee eines kleinen Rechners, der mobil an ein Computernetz angeschlossen sein sollte.
- Gruppe am Xerox PARC (Palo Alto Research Centre) entwickelte Ende der 70er den *Star* (bis 1 MB RAM, 40 MB Platte, 17-Zoll Rasterbildschirm, mechanische 3-Tastenmaus).
- später Darstellung von Objekten als Icons, Textausgabe in mehreren voneinander abgesetzten Fenstern
- Schreibtischmetapher (desktop), für E-Learning auch Schultafelmetapher, bzw. Steuerpult für Spiele
- erstmals objekt-orientierte (wörtlich genommene) Handlungsweise: wähle Objekt aus, bestimme anwendbare Aktion:
  Objekt-Aktion-Paradigma,
- im Gegensatz zum *Aktion-Objekt-Paradigma*, z. B. UNIX Kommando `rm *.backup` (was anwenden worauf)

Xerox PARC hat den Star kaum vermarktet, Alan Kay ging 1979 mit einigen Kollegen zu Apple, dort entstand zunächst

- der Lisa Rechner (1983), dann
- der Macintosh (MC 68000 Prozessor, zunächst 0,5 MB RAM, 8" Monocrom Bildschirm, Eintastenmaus, Würfelform)

MS-Windows in der jetzigen Erscheinung seit Anfang der 90er, Ursprünge in Mitte der 80er

- große Ähnlichkeit (vgl. Rechtsstreit!) mit Apple Konzepten
- wichtig für große Verbreitung dieser Art von Oberflächen

X Windows System für die UNIX Welt (ab 1986)

- eigentlich hardware und systemunabhängig
- verteilt!

Tcl/Tk (ab 1987) als ein Fenstersystem, das eine höhere Schnittstelle hat als X, ferner:

- auch public domain
- sehr portabel
- interpretierend und erweiterbar

Wesentlich neue Ideen jetzt durch das *Internet*, *Web*, *Netscape Browser*, *Java*, *Qt* und *KDE*, usw. ggf zukünftig Rechner, die ihre Software nur noch aus dem Netz beziehen.

Aber auch diese Systeme bedienen sich mehrerer Fenster, lassen sich über Mausaktionen steuern, insofern keine große Innovation.

2 Grundlagen

Für die Gestaltung interaktiver Systeme gibt es eine große Anzahl an interessanten Texten zu den technischen und physiologischen/psychologischen Aspekten des Mensch-Maschine-Dialogs. Im weiteren Sinne geht es um eine ergonomische Arbeitsplatzgestaltung.

> Der Begriff **Ergonomie** setzt sich aus den griechischen Wörtern *ergon* (Arbeit, Werk) und *nomos* (Gesetz, Regel) zusammen. Die Ergonomie ist die Wissenschaft von der Gesetzmäßigkeit menschlicher Arbeit. Zentral ist dabei die Verbesserung der Mensch-Maschine-Schnittstelle zwischen Benutzer (= Mensch) und Objekt (= Maschine) in einem Mensch-Maschine-System. [Wikipedia]

Ein heute häufig gebrauchter Begriff ist dabei die Funktionalität, d.h. die erfolgreich realisierte Eigenschaft eines Produkts oder einer Komponente, eine bestimmte Aufgabe zu lösen. Für die Gestaltung einer Schnittstellenkomponente spricht man im Englischen von *affordance* (besser: *percieved affordance*), ein Begriff, den Don Norman in dem Buch *The Psychology of Everyday Things* (1988) gebrauchte, das später unter dem Titel *The Design of Everyday Things* erschien. Im Deutschen würde man von „Leistbarkeit“ sprechen (was leistet die Komponente, besser: was scheint sie zu leisten), oder auch vom Angebots- oder Aufforderungscharakter (wozu lädt sie ein, wozu fordert sie auf).

Beispiel: Eine Türe bietet sich dafür an (lädt dazu ein), durchzugehen; ein Stuhl „leistet“ das Sitzen, er fordert dazu auf, Platz zu nehmen. Eine

Türe ist gut entworfen, wenn man ihr sofort ansieht, ob es eine Schiebe-, Pendel- oder eine nur nach einer Seite sich öffnende Tür ist.

Ursprünglich stammt der Begriff affordance von dem Wahrnehmungspsychologen *J. J. Gibson* (1904-1979), der damit in seinem 1979 erschienenen Buch *The ecological approach to visual perception* die sich in Aktionen ausdrückende Beziehung zwischen einem Handelnden (Person oder Tier) und der Welt bezeichnete.

Verwandt mit der affordance ist die *usability*, im Deutschen die *Gebrauchstauglichkeit*. Laut Wikipedia gilt:

> Die Definition der Gebrauchstauglichkeit ist in DIN 55350-11, 1995-08, Nr. 4 geregelt. Demnach ist unter Gebrauchstauglichkeit die Eignung eines Gutes zu verstehen im Hinblick auf seinen bestimmungsgemäßen Verwendungszweck; diese Eignung beruht auf subjektiv und nicht objektiv feststellbaren Gebrauchseigenschaften. Die Beurteilung der Gebrauchstauglichkeit leitet sich aus individuellen Bedürfnissen ab.
>
> Geläufiger ist allerdings die englische Übersetzung Usability und die Definition der DIN EN ISO 9241 Teil 11, wonach die Gebrauchstauglichkeit sich aus *Effektivität*, *Effizienz* und *Zufriedenheit* zusammensetzt: Die Gebrauchstauglichkeit (usability) ist das Ausmaß, in dem ein Produkt durch bestimmte Benutzer in einem Nutzungskontext genutzt werden kann, um bestimmte Ziele effektiv, effizient und zufriedenstellend zu erreichen. Der Nutzungskontext besteht aus den Benutzern, Arbeitsaufgaben, Arbeitsmitteln (Hardware, Software und Materialien) sowie der physischen und sozialen Umgebung, in der das Produkt eingesetzt wird.

Es existieren umfangreiche Untersuchungen und reichlich Literatur zu diesen Themen. Die ursprüngliche DIN 66234 - Bildschirmarbeitsplätze, Grundsätze ergonomischer Dialoggestaltung wurde eingearbeitet in die oben genannte DIN EN ISO 9241 und ist daher nicht mehr relevant.

2.1 Kurzübersicht VDI-Richtlinie 5005 und ISO 9241

Die folgenden Stichworte stammen von Jens Jacobsen aus der Web-Seite http://www.contentmanager.de/magazin/artikel_582-199_usability_normen_din_iso.html.

2.1.1 VDI-Richtlinie 5005 – Software-Ergonomie

Die Richtlinie der Vereins Deutscher Ingenieure (VDI) verlangt diese Eigenschaften von einer Anwendung:

- Kompetenzförderlichkeit
  Das System soll den Benutzer fördern und ihm erlauben, den Umgang mit ihm leicht zu erlernen. Sein erworbenes Wissen soll er auch auf neue Aufgaben, die er mit dem System erledigt, übertragen können.

- Handlungsflexibilität
  Aufgaben sollen sich auf mehreren Wegen erledigen lassen. Je nach Erfahrung des Benutzers sollte es unterschiedliche Möglichkeiten anbieten - z.B. einen sehr schnellen für erfahrene Nutzer und einen mit viel Hilfestellung für Neulinge. Auch sollte die Arbeitsweise mit dem System gleich bleiben, wenn sich die Aufgabe ändert.

- Aufgabenangemessenheit
  Die Aufgabe muss sich rein technisch mit dem System durchführen lassen. Das sollte schnell und ohne Probleme möglich sein.

2.1.2 DIN EN ISO 9241 – Ergonomische Anforderungen

Die mit Abstand wichtigste Norm für Websites ist die DIN EN ISO 9241. Sie besteht aus 17 Teilen, die, neben einer Einführung, Anforderungen an Tastaturen, Displays, die Arbeitsplatzgestaltung und die

Arbeitsumgebung enthalten. Teile 10 bis 17 beschäftigen sich mit den für Software- und Website-Entwicklung bedeutendsten Punkten. Sie beschreiben eine benutzerfreundliche Anwendung mit folgenden Eigenschaften:

- der Aufgabe angemessen
  Die Anwendung soll das leisten, was der Benutzer von ihr erwartet. Dabei soll sie ihn unterstützen und schnell zum Ziel führen. Die eingesetzte Technik soll für den Nutzungsfall angemessen sein.

- selbst beschreibend
  Die Anwendung soll dem Benutzer deutlich machen, wie er sein Ziel erreicht. Klare Navigation und verständliche Anweisungen bei jedem Schritt sind dazu Voraussetzung.

- steuerbar
  Der Benutzer soll die Anwendung steuern, nicht umgekehrt. Das heißt, dass Animationen abgebrochen und erneut gestartet werden können, es immer einen Weg zurück gibt und bei Ton die Lautstärke reguliert werden kann.

- erwartungskonform
  Die Anwendung soll den Benutzer nicht überraschen. Konsistenz innerhalb der Anwendung ist daher Pflicht, aber auch die Berücksichtigung weit verbreiteter Konventionen.

- fehlertolerant
  Das System soll mit falschen Benutzereingaben umgehen können und bei Fehlern klare Rückmeldung geben. Der Korrekturaufwand für den Benutzer soll minimal sein.

- individualisierbar
  Der Benutzer soll die Möglichkeit haben, die Anwendung an sein Vorwissen bzw. an seine Vorlieben anzupassen. Üblicherweise wird das mit "Personalisierbarkeit" bezeichnet, z.B. indem eine Website die Voreinstellungen bzw. Angaben des Benutzers

speichert, so dass er sie beim nächsten Besuch nicht erneut eingeben muss.

- lernförderlich
  Die Anwendung soll den Benutzer dabei unterstützen, den Umgang mit ihr schrittweise zu erlernen.

Die Norm gibt keine konkrete Anleitung, wie die Gebrauchstauglichkeit geprüft werden kann. Deshalb hat die DATech (Deutsche Akkreditierungsstelle Technik e.V.) ein Handbuch dazu erstellt: das "DATech-Prüfhandbuch Gebrauchstauglichkeit - Leitfaden für die software-ergonomische Evaluierung von Software-Erzeugnissen auf der Grundlage von DIN EN ISO 9241, Teile 10 und 11". Diese ist jedoch extrem umfangreich und für nicht-Geschulte kaum zu verwenden. Hilfestellung für alle, die nicht zu Experten werden möchten, gibt die Site von IBM.

Man kann die Gebrauchstauglichkeit von Anwendungen durch von DATech akkreditierte Prüflabors zertifizieren lassen, wovon bisher aber noch sehr wenige Unternehmen Gebrauch gemacht haben (v.a. öffentliche Verwaltung, Banken, Versicherungen). (Ende des Zitats der Web-Seite).

Es folgen einige praktischere Betrachtungen zu Einzelproblemen der Schnittstellengestaltung.

2.2 Bildschirmaufteilung

Optimierungsproblem bei beschränktem Bauplatz (real estate): Bildschirm hat zu geringe Fläche

Größe eines Dialogobjekts (Fenster, Kopf, Anzeigefeld) sollte von der Bedeutung, dem Kontekt und einigen weiteren Kriterien abhängen (Klingert). Daneben spielen motorische Gegebenheiten eine Rolle.

2.2.1 Fitts'sches Gesetz

Quantitative Aussage über Anfahren und Auswählen von Dialogobjekten bei Fenstersystemen – Koordination Hand ↔ Auge.

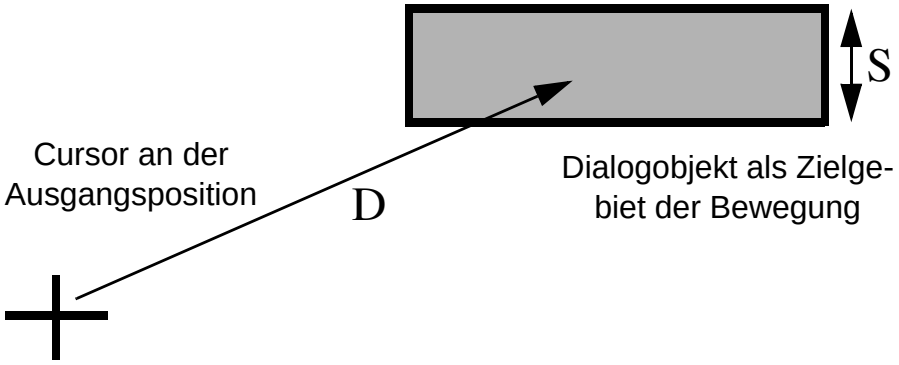


Abb. 2–1 *Positionierung als Koordinationsproblem nach Fitts, zitiert in [5], S. 13*

Größere und näherliegende Objekte sind „leichter" zu erreichen (leichter auch gleichzusetzen mit schneller)

$$T_{\text{pos}} = I_t + I_m \log_2(D/S + 1)$$

wobei I_t und I_m zu messende Personenkonstanten sind, S die Ausdehnung des Zielobjekts (kleinere Kante bei rechteckigen), D die Anfangsdistanz.

Vergleiche auch *Gesetz von Merkel* für die Reaktionszeit T einer Versuchsperson bei Auswahl unter n Objekten:

$$T \approx 200 + 180 \log_2 n \text{ [msec]}$$

Größe und Distanz sollen „angemessen" sein → Zielkonflikt: überladener Bildschirm durch Dialogobjekte → Icons, überlappende Fenster, aber hohe Überlappung = mühsames Arbeiten.

2.2.2 Gestaltungsgesetze

Neben dem bereits erwähnten Donald Norman gelten Jef Raskin, Jakob Nielsen und Steve Krug als wichtige Namen für das Interfacedesign als Teil der kognitiven Psychologie. Ihr Schwerpunkt ist der "Common-Sense-Approach", der Rücksicht auf kulturelle Gewohnheiten nimmt.

Formen und Anordnungen werden unabhängig vom persönlichen Geschmack weitgehend einheitlich als angenehm und angemessen empfunden (ca. 100 Gesetzmäßigkeiten), vgl. auch Wissenschaft von der Bedeutung der Zeichen (*Semiotik*)

- Einheitlichkeit (Form, Linienstärke, Fontart, Fontgrößen)
- Farbkomposition
- Symmetrie

☞ Gesetz der Prägnanz: wenige, ausgezeichnete, deutliche Formen werden gut erkannt und bleiben gut in Erinnerung!

☞ Gesetz der Nähe: Gruppierung durch Nebeneinanderstellen, z. B. Menupunkte

☞ Gesetz der Geschlossenheit: Gruppierung durch Umrandung

☞ Gesetz der Gleichheit: gleiche Form, Farben, Fonts für semantische Gruppen

☞ Gesetz der Erfahrung

Nlzub, Vygklinaq

QTF Ukloxdriih, Drawutz ikl Edfjeh

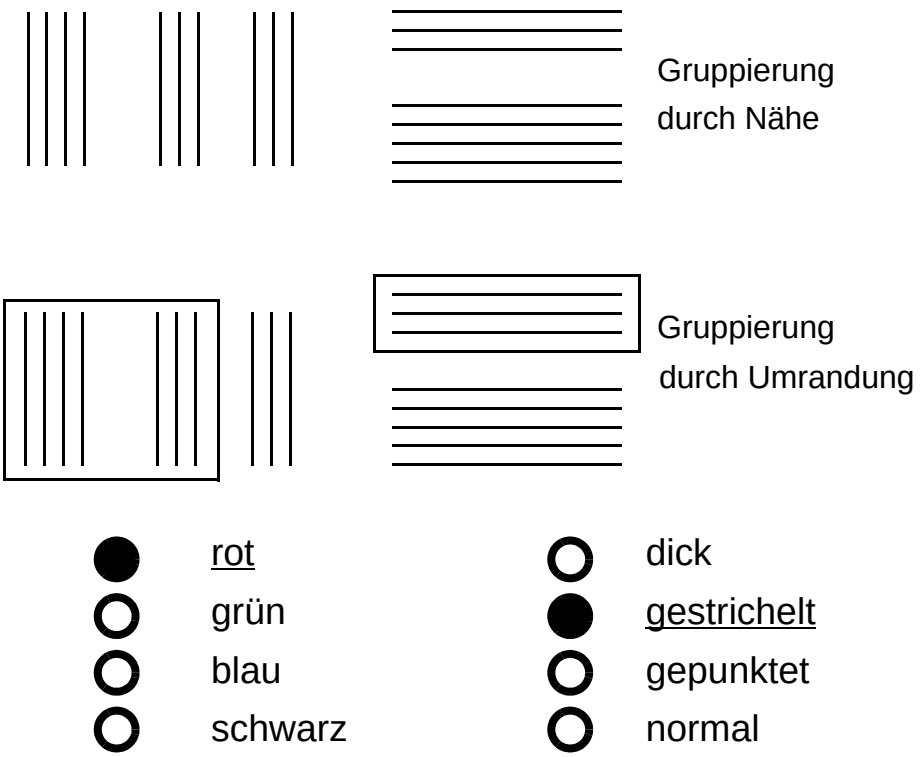


Abb. 2–2 *Gruppierung und Gleichheit [Klingert, S. 16]*

Klimsa und Bruns [18] sprechen von interner und externer Konsistenz.

- *interne Konsistenz*: Innerhalb einer Applikation müssen die gleichen Informationen immer gleich dargestellt werden, ähnliche Anweisungen müssen auf ähnliche Art und Weise ausgelöst werden.
- *externe Konsistenz*: Neue Applikationen sollen in der Dialoggestaltung vergleichbar mit anderen sein, so daß der Lernaufwand für die Steuerung neuer Applikationen gering ist.

Dies betrifft

- Grundlagen (Verwendung von Werkzeugen, Dokumenten, Hilfen)
- Befehle (Befehlsauslösung: Knopf, Menu, Eingabezeilen)

- Beschriftungen (Einsatz von Text zur Steuerung, Erklärung und als Hilfe)
- Farben
- Interaktionselemente (Art, Form, Funktion der möglichen Interaktion)
- Sicherheitselemente (Abwehren von Benutzerfehlern, UnDo-Funktionen, Kopien)
- Visuelle Gestaltung (Verwendung von metaphern mit eingeschränkter Wirkung, z.B. Fenster und Icons, Anordnung der Inhalte, Schrift)

2.3 Eingabegeräte

2.3.1 Texteingabegerät

- Tastatur
- Schrifteingabefläche
- Spracherkennung

Graphisches Eingabegerät zum Zeigen auf Dialogelemente

- Maus
- Lichtgriffel
- Tablett
- Gesten (Datenhandschuh)
- Augenverfolgung (eye tracking)
- Joystick

Interessant: Simulation anderer Eingabegeräte am Bildschirm durch die obigen Dialogelemente, z. B. ein Schieberegler zur Kontrolle der Stärke von ...

Die These vom fehlenden Organ (Hermann Maurer et al.)

Der Mund ist das natürliche Gegenstück zum Ohr. Welches Organ ist das Gegenstück zum Auge?

Die These vom fehlenden Organ besagt, daß der Mensch sich erst durch den Computer (den Bildschirm) ein Gegenstück zum Auge geschaffen hat.

2.3.2 Graphische Zeigegeräte

direkt arbeitend oder *indirekt* ~

- indirekt: über ein Gerät den *Cursor* bewegen
  - Maus
  - Tablett
- direkt: Antippen, Berühren der Bildschirmoberfläche
  - Berührbildschirm (touch screen, touch panel)
  - Lichtgriffel

mit *absoluten* oder *relativen* Koordinaten

- Mausbewegung ist relativ zur letzten Position registriert (Hochheben, woanders wieder absetzen ist möglich)
- Lichtgriffel und Tablett liefern Koordinaten im globalen Koordinatensystem des Bildschirms oder des Tabletts

2.3.3 Die Maus

kontinuierliches (bis auf Maustasten), indirektes, relatives Eingabegerät

- alternativ Trackball, Trackpoint, Trackscreen

kommt als

- optische (oder herkömmlich als)
- mechanische Maus (Ballbewegung übertragen auf x-/y-Achsen)

✎ Wenn eine Maus auch als *handmouse* bezeichnet wird, was ist dann ein „Maulwurf“ (*mole*)?

2.3.4 Die Tastatur

Üblich im Deutschen: die QWERTZ-Tastatur, im Englischen die QWERTY-Tastatur.

Kommt aus dem Schreibmaschinenzeitalter

Alternativ

- die Dvorák-Tastatur (andere Anordnung der Tasten)
- Akkord-Tastatur (chord keyboard)

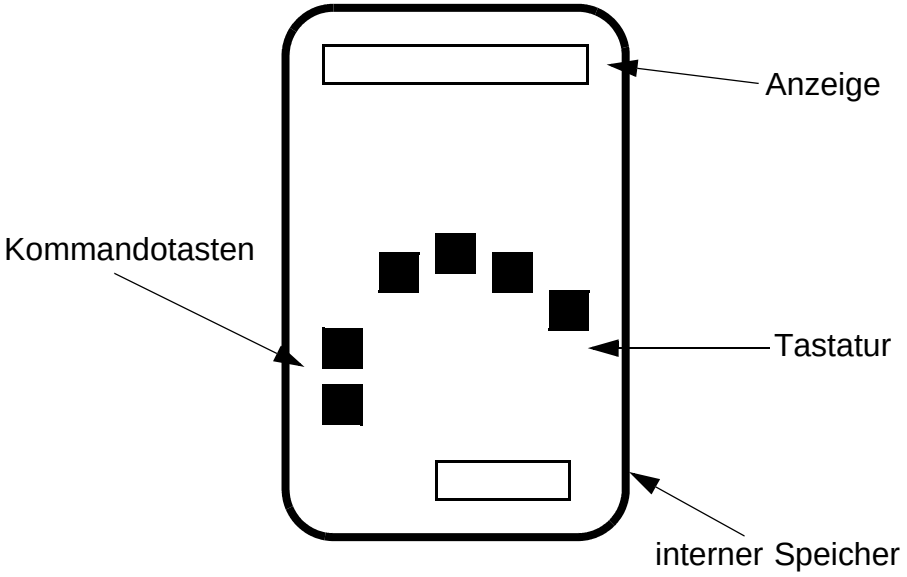


Abb. 2–3 *Akkord-Tastatur (nach [8], S. 216)*

Nutzung der Pfeiltasten als Mausersatz (dann Tastatur als diskretes und absolutes Eingabegerät)

2.3.5 Andere Eingabegeräte

- Spracheingabe, noch in der Entwicklung; Problem: starke Sprecherabhängigkeit
- Lippenlesen; Vorteil: im Englischen nur 16 deutlich verschiedene Lippenstellungen
- Augenbeobachtung, Kopfbewegungen
- 3D-Eingabe: zukünftig von Bedeutung (Spacemouse, Spaceball, data glove)
- Gestenbeobachtung per Video
- Handschriftensysteme (pen based systems)

2.4 Ausgabegeräte

Komponenten der Ausgabe für graphische Fenstersysteme:

- Bildschirm
- Bildspeicher
- Graphikprozessor

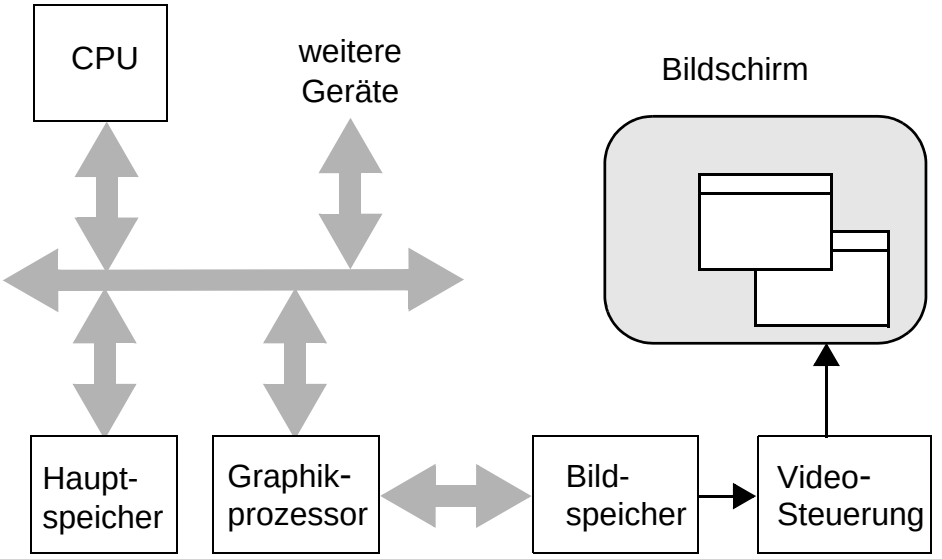


Abb. 2–4 *Anordnung der Ausgabe-Hardware [Klingert S. 25]*

Graphikprozessor

Soll parallel zur CPU elementare Zeichenoperationen ausführen, z. B. `ZeichneLinie` und `FüllePolygon`

Wichtig auch: Verschieben von Blöcken im Bildspeicher

Bildspeicher

Schneller RAM, ggf. dual ported; Größe bestimmt Auflösung und Farbtiefe.

- Mensch kann ca. 2.000.000 Farben unterscheiden, damit fast 3 Byte je Bildpunkt (Truecolor: 24 bit = 16.777.216 Farben, je ein Byte für rot, blau und grün).

Bildschirme

Kriterien sind

- Anzahl der Bildpunkte
- Bildschirmdiagonale, gemessen in Zoll (auch wenn's selbsternannten Wettbewerbshütern nicht paßt!)
- Auflösung (z. B. in x-Richtung) heute üblich 70-110 ppi (pixel per inch, 1 Inch = 2,54 cm)
- Bildwiederholfrequenz bei Röhrenmonitor sollte bei mindestens 85 Hz liegen; abhängig von Auflösung und leistbarer Horizontalfrequenz (üblich mind. 80 kHz)
- Qualität der Lochmaske, Abstand der Löcher machbar z. Zt. ca. 0,25 mm
- Nachleuchten
- Strahlungsarmut

Alternativ Flüßigkristallanzeige (LCD) mit

- DSTN (*dual super twisted nematic*) oder
- TFT (*thin film transistor*, auch *active matrix* genannt) Technik

- heute noch häufig 1024 × 768 Pixel (XGA) bei 32-bit Farben, künftig auch UXGA mit 1600 × 1200 für Verhältnis 4:3; häufig heute auch andere Seitenverhältnisse, etwa 16:10, dann also z.B. 1920 × 1200.
- strahlungsfrei!
- Reaktionszeiten der Flüssigkristalle möglichst klein

Sprachausgabe

Noch unbefriedigend, aber attraktiv.

2.5 Softwaretechnik

Grundprinzipien:

- effiziente Grundobjekte
- ereignisgesteuerter Ablauf (Eingabe)

daneben

- Objektorientierung
- Multitasking
- deklarative Spezifikationssprachen

Fenster als effiziente Objekte

Abbildung aller Dialogobjekte auf ein einheitliches Grundobjekt: das Fenster. Das Fenstersystem verwaltet alle Objekte nach dem gleichen Muster → Effizienz, mehrere hundert auf Bildschirm möglich.

Das Ereignismodell

Abbildung aller möglichen Eingabevarianten auf das Datenobjekt „Ereignis“. Aus Sicht der Anwendung eine Endlosschleife (interruptgesteuert) zum „Lauschen“ auf Eingabe, z. B. Mausbewegung. Ereignisse werden sequentiell abgearbeitet durch Routinen, die an die Klassen von Ereignissen gebunden sind.

Multitasking

Mehrere Fenster mit simultanen Aktionen erfordern ein echtes Multitasking-Betriebssystem.

Objektorientierung

Bündelung von Daten und Methoden für Objekte ist ideal für Fenstersysteme; genauso Interaktion untereinander über Botschaften; Vererbungsmechanismen erlauben Entwicklung von Spezialdialogobjekten aus elementaren Objekten, die viel von deren Funktionalität übernehmen (z. B. Iconisierbarkeit, Verschiebbarkeit, usw.)

Deklarative Spezifikationssprachen

Dialogobjekte bestehen zum größten Teil aus Angaben über Größen, Position, Farben, verwendete Fonts, usw. Auch Ereignisverarbeitung läßt sich mit Listen (Ereignis e_i, Reaktion r_i) beschreiben und durch einen geeigneten Regelinterpreter abarbeiten.

Stichwort: *Resource Files*

Diese Angaben sind also deklarativer Form, weniger prozeduraler Art, daher bietet es sich an, das Fenstersystem in einer deklarativen Sprache zu spezifizieren.

Alternative (Wegner): Dialogobjekte in Datenbank ablegen, Abarbeitungsskripte als Attributwerte zulassen.

2.6 Aufgaben eines Fenstersystems

Die theoretischen Grundlagen sollen an einem einfachen Fenstersystem herausgearbeitet werden.

2.6.1 Minimalforderungen

- Eingabeverwaltung:
  Aktionen des Benutzers an Anwendung weitergegeben

- Ausgabeverwaltung:
  Ausgaben des Anwendungsprogramms (Text, Graphik) in Fenstern sichtbar machen
- Fensterverwaltung:
  System weiß, wo sich Fenster befinden, so daß Graphikoperationen richtig relativ zum Fenster ausgeführt werden

2.6.2 Schnittstellenbeschreibung für minimales Fenstersystem

inkl. Problem der Überlappung (aus [5], S. 43ff)

```
Return SetupWindowDisplay();
```

/* Es soll geprüft werden, ob der Arbeitsplatz die Fähigkeit besitzt, Fenster zu verwalten. Gegebenenfalls wird das System initialisiert. */

```
Window OpenWindow( int x, y, width, height );
```

/* Fenster erzeugen. Die Position kann beliebig gewählt werden. Die Fenster werden am Rand der Bildschirmfläche abgeschnitten. Die Koordinaten sind bezogen auf den linken Rand des Schirms. */

```
Return CloseWindow ( Window w );
```

/* Ein Window wird bei Programmende geschlossen und vernichtet. Diese Funktion dient also nur der Übersichtlichkeit des Bildschirms, indem man überflüssige Fenster verschwinden lassen kann. */

```
Return ClearWindow( Window w, ColorIndex idx );
```

/* Der Inhalt eines Windows wird gelöscht (weiß bzw. die spez. Farbe). */

```
Return RaiseWindow( Window w );
```

/* Falls das Fenster (teilweise oder vollständig) unter einem anderen lag, wird es nun zuoberst plaziert sein. Alle bisher verdeckten Ausgaben sind jetzt wieder zu sehen. Wenn das Fenster schon das oberste war, dann geschieht garnichts. */

```
Return CopyRectangle( Window w, int x1, y1,
                      int x2, y2, width, height );
```

/* Der Teil des Fensters, der von (x1, y1) nach rechts die Breite width, nach unten die Ausdehnung height hat, wird an die Stelle x2, y2 übermalend kopiert. */

```
Return DrawLine( Window w, int x1, y1, x2, y2,
                 int width, ColorIndex idx );
```

/* Zieht eine Linie in das spezifizierte Window. Die Linie braucht nicht an der Window-Umrandung abgeschert zu sein. Das Windowkoordinatensystem hat seinen Origo in der oberen linken Ecke mit positiver x-Achse nach rechts und positiver y-Achse nach unten (Einheit: int). */

```
Return WriteText( Window w, char *info,
                  int x, y, ColorIndex idx );
```

/* Schreibt einen Text an die Stelle x, y in ein Fenster, Der Text sollte ein terminierter String sein. Er wird gegen die Window-Umrandung geklippt. Koordinaten wie bei DrawLine. */

```
Return SelectInput( Window w, Modus modi );
```

/* Bestimmt, welches Fenster ab jetzt welche Eingaben erhalten bzw. registrieren soll. */

```
Input NextInput( Window w );
```

/* Holt eine Input-Aktivität für ein Fenster ab. Geliefert wird nur, was vorher mit SelectInput() bestellt wurde. */

Wesentliche Definitionen sind weggelassen. Die Anwendung lautet:

```
Window     einFenster;
Input      dasEreignis;
Bool       nicht_fertig = TRUE;
Return     RetCode;
ColorIndex schwarz = 0, weiss = 1;

main()
  {
  RetCode = SetupWindowDisplay();
  einFenster = OpenWindow(100, 100, 400, 400);
```

```
    RetCode = DrawLine(einFenster, 0, 0, 400, 400,
                       schwarz);
    RetCode = WriteText(einFenster,
                        "Hallo Fenstergucker",
                        204, 200, schwarz);
    RetCode = SelectInput(einFenster,
                          TastaturOderMausklick);
    while (nicht_fertig){
      dasEreignis = NextInput(einFenster);
      switch (dasEreignis.modus) {
      case Maus: nicht_fertig = FALSE;
                 ClearWindow(einFenster, weiss);
                 break;
      case Tastatur: CopyRectangle(
                          einFenster, 0, 20, 0, 10,
                          400, 10);
      } /* switch */
    } /* while */
  } /* main */
```

Hinweis: Klingert schreibt als Kommentar zum Aufruf von `CopyRectangle` „Text rollen“; allerdings wird nur ein 10 Einheiten breiter Streifen oben verschoben!

Offensichtliche Mängel des Minimalsystems:

- unzureichende Graphikfähigkeiten (nur Linie und Text, keine Kreise, Polygone, usw.)
- großer Aufwand für alltägliche Aufgaben (Beispiel Menü)

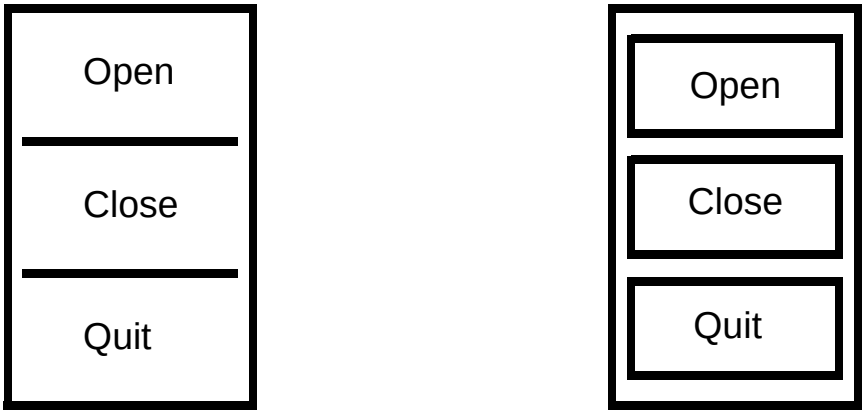


Abb. 2–5 *Menü mittels eines und mehrerer Fenster [5]*

Im Beispiel links aufwendiger Code um herauszubekommen, auf welchen Menüpunkt geklickt wurde; rechts ineffiziente Schleife über drei Fenster um herauszubekommen, welches Fenster die Eingabe erhält.

2.6.3 Kriterien für Fenstersysteme

- Verfügbarkeit
  Hardwareanforderung, Betriebssystem
- Produktivität
  Wie schnell und leicht lassen sich – auch auf lange Sicht – die Dialogkomponenten von Anwendungen erzeugen und pflegen? Muß man dazu Systemprogrammierer sein?
- Parallelität
  Gibt es *echte interne Parallelität* (pre-emptive scheduling and context switches) oder ist dies z. B. von der Anwendung zu emulieren?
- Leistung
  zukünftig bis zu 90% Prozessorleistung für multimediale Benutzungsschnittstelle geopfert
- Raster- versus Vektormodell
  Vektormodell hat größere Abstraktion, Rastermodell ist einfach zu implementieren, hat Probleme beim Skalieren
- Stil
  fest vorgegeben oder sehr flexibel
- Erweiterbarkeit
  gar nicht, per Quellcode, per Interpreter
- Anpaßbarkeit
  z. B. auf Landessprache, sonst. Gewohnheiten; deshalb
  wie (gar nicht, ein wenig ...), wann (Laufzeit, Neukompilieren), von wem (Anwender, Anbieter)
- Teilbarkeit der Ressourcen
  z. B. mehrere Applikationen teilen sich Fontsdefinition
- Verteilung
- API-Schnittstellen
  prozedural, objektorientiert, Komfort (Prototyping möglich?)

- Trennung Applikation - Dialogteil
- Kommunikation zwischen Anwendungen, speziell interaktionsgesteuerter Austausch → *Cut-and-Paste*
  - entweder statisch (Kopieren) oder
  - dynamisch (aufnehmen Verweis, *DDE: Dynamic Data Exchange*)
  - Austausch 1:1 (*Selection*, Herkunftsanwendung macht Formatumwandlung) oder
  - Clipboard (Zwischenspeicher), dann m:1:n-Austausch, d. h. kopieren von vielen Stellen ins Clipboard und aus Clipboard nach vielen Stellen kopierbar
  - OLE (Object Link and Embedding), kein Datentransfer im engeren Sinne, nur Verzeigerung, Objekte machen ihre Funktionalität verfügbar (ungerichtet, n:m-Beziehung)

2.6.4 Komponenten eines Fenstersystems

Rasterbildschirm, Graphikprozessor, graph. Eingabegerät, Tastatur, ggf. Vernetzung, weitere HW-Komponenten

SW als Schichtenarchitektur

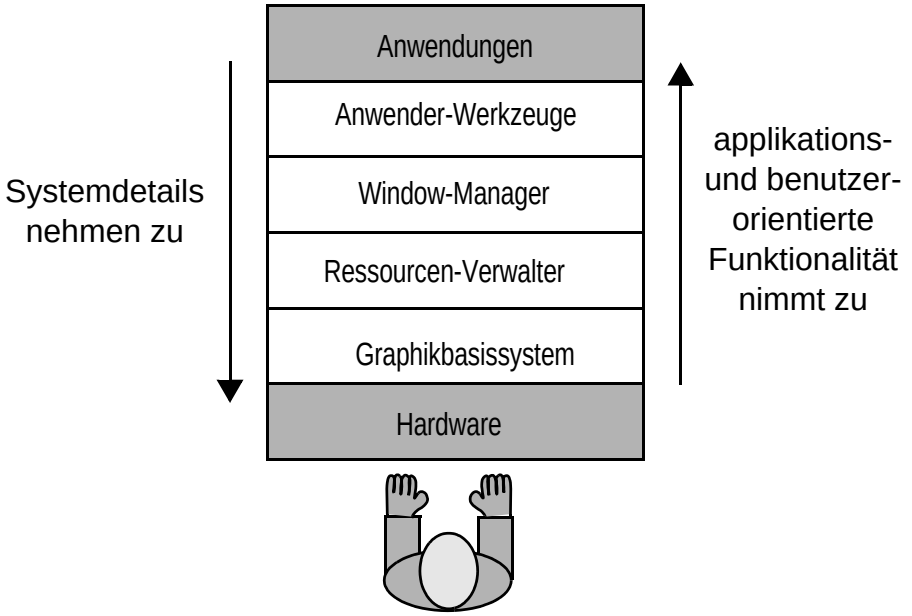


Abb. 2–6 *Schichten-Modell ([5], S. 57)*

- **Anwender-Werkzeuge**: Toolkit, funktions- oder objekt-orientiert
- Window-Management: Verwaltung der Bildschirmbereiche
- Ressourcen-Verwaltung: regelt konkurrierenden Zugriff auf Ressourcen, z. B. synchronisiert Mausaktionen
- Graphikbasissystem (Graphikbasis): elementare Graphikfunktionen hardware-nah realisiert, z. B. auch Registrierung von Mausclicks

Ziel

generisches, flexibles, mächtiges, intuitives, klares ... System
→ Zielkonflikt!

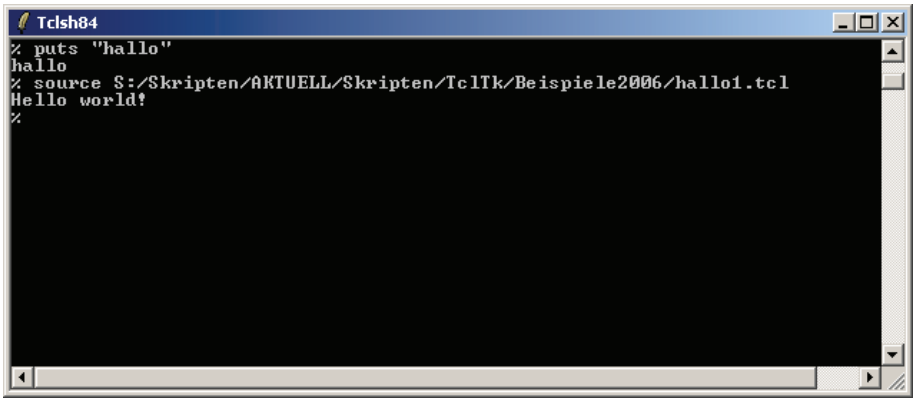
3 Einführung in die Skriptsprache Tcl

3.1 Tclsh und wish

Die folgenden Ausführungen lehnen sich eng an das Tutorial von Wolfram Schenk an.

Tcl (Tool command language, ausgesprochen „tickle“) ist eine Skriptsprache die interpretiert wird (Übersetzen und Ausführen in einem Zug). Tcl gehört zur Kategorie der klassischen imperativen Programmiersprachen.

Einzelne Kommandos kann man direkt aus der Shell mit dem Interpreter `tclsh` ausführen, meist nur zum Testen von Kommandozeilen. Auf einem Windows-Rechner kann man z. B. `Tclsh84` aus der unter
`http://www.activestate.com/Products/ActiveTcl/`
zur Verfügung gestellten Tcl/Tk 8.4 Implementierung verwenden.



Alternativ verwendet man von der selben Quelle die Windowing Shell `wish`, die bei Aufruf sofort ein leeres Fenster erzeugt. Anders als `tclsh` kennt `wish` auch alle Tk-spezifischen Kommandos.

In der Regel erstellt man eine Textdatei, die Tcl-Kommandos enthält. In der Bildschirmanzeige oben haben wir eine Datei `hallo1.tcl` mit dem Inhalt

```
#mein erstes Programm
puts "Hello world!"
```

erstellt. Mit `source S:/Skripten/.../hallo1.tcl` können wir das Skript in der `Tclsh84` zur Ausführung bringen.

Unter UNIX wäre ein Tcl-Skript eine zur Ausführung freigegebene (UNIX: `chmod +x` *`dateiname`*) Textdatei. Damit der Tcl-Interpreter, z. B. `tclsh`, das Skript abarbeitet und nicht die UNIX Shell, käme dann noch eine neue Startzeile mit dem Pfadnamen des gewünschten Interpreters in das Skript. Die erzeugte Ausgabe des Beispiels bleibt gleich.

```
#!/usr/bin/tclsh
#mein zweites Programm
puts "Hello world!"
```

Diese eher verwirrenden Details der Ausführung besprechen wir in den Übungen.

3.2 Kommandos und Variablen

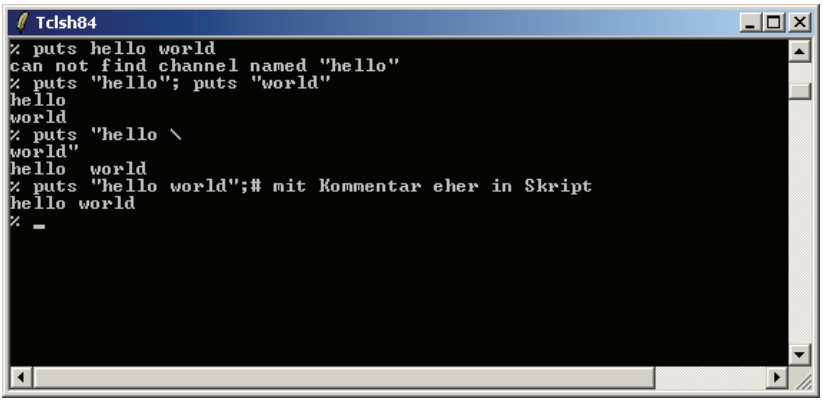
In der Regel ist das erste Wort einer Kommandozeile der Kommandoname. Tcl unterscheidet (in der Tradition von UNIX) Groß- und Kleinschreibung.

```
Puts "gross und klein"
invalid command name "Puts"
```

An Sonderzeichen ist zu beachten (nach Schenk):

- `;` Kommandoabschluss (optional), weiteres Kommando kann dahinter geschrieben werden
- `\` Fortsetzung des Kommandos in der nächsten Zeile

- # Beginn einer Kommentarzeile
- ;# Beginn eines Kommentars im Anschluss an ein Kommando in derselben Zeile
- "..." Anführungszeichen; notwendig für die Definition von Zeichenketten, die Leerzeichen enthalten



Hinweis zur Syntaxangabe der Kommandos hier im Skript: Kommandonamen schreiben wir im normalen Schriftstil, formale Parameter *kursiv*, wenn eine Angabe optional ist, wird sie in Fragezeichen eingeschlossen, Wiederholungsoptionen werden durch drei Punkte angedeutet.

```
puts string
after ms ?arg? ?arg...?
```

Alle Kommandos liefern als Ergebnis ihrer Ausführung ein Resultat zurück, was wir bei der Syntaxangabe nicht extra zeigen. Das Resultat ist eine Zeichenkette (string), da dies der einzige Datentyp für Variablen und Konstanten in Tcl ist. Kommandos, die von ihrer Art her eigentlich kein Ergebnis liefern, geben die *leere Zeichenkette* zurück.

Mit dem `set`-Kommando weist man einer Variablen einen Wert zu.

```
set varname string
```

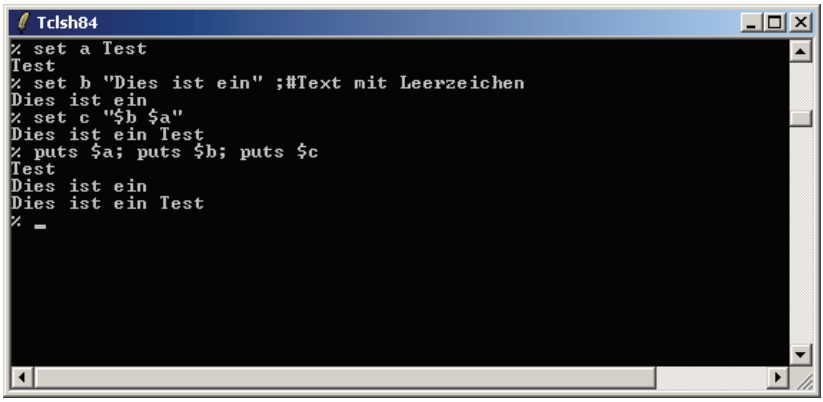
also z. B. `set a 4711` oder `set gruss "hallo"` oder `set gruss hallo`. Man sieht an diesen drei Beispielen auch, daß die Verwendung von Anführungszeichen in vielen Situationen entbehrlich ist. Ferner sei erwähnt, daß manche Ausdrücke auch numerische Operanden erwarten, d. h. die Zeichenketten müssen sich entweder als ganze Zahlen oder Gleitkommazahlen interpretieren lassen.

```
expr 3 * 4
12
```

Insofern ist die Klassifizierung von Tcl als *typlose Sprache* (typeless language) nicht ganz richtig. In jedem Fall muß man, wie bei interpretierten Sprachen üblich, Variablen nicht deklarieren. Durch die erste Verwendung mit Wertzuweisung sind sie bekannt gemacht.

Den Wert einer Variablen erhält man durch vorgestelltes „Dollarzeichen“.

```
set gruss "Lieber typlos als treulos!"
puts $gruss
Lieber typlos als treulos!
```



Wie für UNIX gibt es keine Einschränkung für Variablenbezeichner, man wird sich aber vernünftigerweise auf Buchstaben, Ziffern und Unterstrich beschränken und mit einem Buchstaben anfangen. Auf die Unterscheidung von Groß- und Kleinschreibung sei nochmals hingewiesen. Das

set-Kommando ohne zweites Argument liefert übrigens den Wert der Variablen zurück.

```
% set 122 a ;# nicht empfehlenswert
a
% puts $122
a
%set 122
a
%set a "Hallo Leute"
Hallo Leute
%set $122
Hallo Leute
%
```

Mehr hierzu, wenn wir unten die Substitution besprechen.[1]

Für die zeilenweise Eingabe steht das gets-Kommando zur Verfügung.

```
gets channelId ?varname?
```

Dabei bezeichnet *ChannelId* einen Dateibezeichner (file descriptor) zu einer Datenquelle (Datei, Tastatur, Socket), die vorher geöffnet sein muß. Standardmäßig steht immer stdin zur Verfügung.

```
%gets stdin zeile
Dies ist meine Eingabe
22
%
```

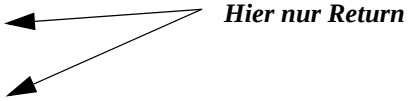
Die eingegebene Zeile wird ohne das abschließende Zeilenendezeichen (*newline*, NL-Zeichen) als Zeichenkette der Variablen zugewiesen. Das Kommando liefert als Resultat die Anzahl der gelesenen Zeichen (ohne NL) zurück.

Fehlt die Variable, wird der gelesene String (ohne NL) geliefert. Wird an einer Tastatur nur Return gedrückt, liefert gets die leere Zeichenkette.

---

1. Um Mißverständnissen vorzubeugen: wie in C kann man in Tcl leicht „Programmierrätsel“ mit eingebauten Denkfallen hinschreiben („na, was liefert dieser Ausdruck wohl?“). Das sollte man sowohl bei der Vermittlung als auch in der Praxis später vermeiden, um praktisches Probieren wird man aber nicht herumkommen.

```
% gets stdin
Dies ist eine Eingabe ohne Variable
Dies ist eine Eingabe ohne Variable
% gets stdin zeile

0
% gets stdin

%
```



Zuletzt sei der Fall erwähnt, daß `gets` auf das Dateiende stößt (aus dem Channel nichts mehr zu lesen ist). In diesem Fall erhält die Variable den leeren String und das Kommando liefert -1 als Rückgabewert, unter UNIX kann man das auch interaktiv ausprobieren, weil CTRL-D das Dateiende signalisiert. Unter Windows mit `Tclsh84` funktioniert dies weder mit CRTL-D noch mit CTRL-Z, letztere Eingabe schließt die Shell.

3.3 Ausdrücke

Das `expr`-Kommando wertet einen Ausdruck aus und liefert das Ergebnis der Auswertung als Zeichenkette zurück.

```
expr expression
```

Dabei muß *expression* ein gültiger arithmetischer Ausdruck sein. In der Angabe des Ausdrucks dürfen Leerzeichen enthalten sein.

```
%expr 3.14 * 5
15.7
%
```

Arithmetische Ausdrücke bestehen aus Zahlen, numerischen Variablen, Operatoren und mathematischen Funktionen.

```
set a 0.5
set b [expr 2.0*sin($a)]
```

Pseudo-Datentypen

- Integer (Ganzzahl)
  - Besteht nur aus Ziffern, optional führendes – oder +

- Double (Fließkommazahl)
  - Besteht aus Ziffern, optional führendes – oder +
  - Muss zusätzlich Dezimalpunkt und/oder Trennzeichen zwischen Mantisse und Exponent enthalten (e oder E)
  - Für Exponent: Optional führendes – oder +
  - Bsp.: `0.1 1e-01 -5.6E9`
- Boolean (Wahrheitswert)
  - Entweder 0 (false) oder 1 (true)
  - Konvention zur Interpretation von Zahlenwerten als Wahrheitswert: `0` oder `0.0` bedeuten “false”, ansonsten “true”
- Arithmetische Operatoren (Auswahl)

`- +`	unäres Minus-/Pluszeichen als Vorzeichen,
`! ~`	logisches NICHT, bitweises Komplement für numerische Werte
`* / %`	Multiplikation, Division, Restbildung
`+ -`	Addition, Subtraktion
`< > <= >=`	Kleiner-/Größer-Vergleiche
`== !=`	Vergleiche: gleich, ungleich
`eq ne`	Vergleich (equal, not equal) für Zeichenkettenoperanden
`&&`	logisches UND
`||`	logisches ODER

  Absteigende Reihenfolge entsprechend der Präzedenz.
  Mit runden Klammern kann man Berechnungsreihenfolge festsetzen: `3+4*5` ergibt `23`, `(3+4)*5` ergibt `35`

Das `expr`-Kommando hat die Tendenz, Operanden numerisch zu interpretieren. Ausnahme ist `eq` und `ne`.

```
% expr 12 ne 13
1
% expr 12.0 eq 12.00
0
```

```
% expr 12.0 == 12.00
1
%
```

Grundsätzlich gilt: eine Kommandozeile wird zunächst vom Tcl-Interpreter rein syntaktisch untersucht und zerlegt („geparst“), (ungeschützte) Leerzeichen, ein NL bzw. „;“ werden entfernt, es erfolgt eine Variablen-, Kommando- oder Backslashersetzung. Das erste Wort im Resultat wird als Kommandoname interpretiert. Das Kommando wird aufgerufen und analysiert jetzt semantisch die übergebenen Argumente.

Die Ergebnisse sind nicht immer intuitiv, wie man an den Beispielen unten sieht. Es geht nochmals um den Vergleich von Zeichenketten.

In einem Ausdruck wird bei einem Vergleich mit z. B. < bei Zeichenkettenoperanden tatsächlich auch ein lexikographischer Vergleich (`"abc"` ist lexikographisch kleiner als `"ac"`, anders gesagt: `"abc"` kommt in der Sortierfolge der Wörter vor `"ac"`) gemacht – allerdings muß man erst den Tcl-Interpreter überwunden haben, der bei nichtnumerischen Zeichenfolgen immer Variablenreferenzen vermutet. Das `expr`-Kommando analysiert semantisch und macht andererseits bei numerischen Zeichenketten sofort einen arithmetischen Vergleich. Auf die Substitutionen des Interpreters gehen wir unten ein.

```
% expr ab ne ab
syntax error in expression "ab ne ab": variable
references require preceding $
% expr {ab ne ab}
syntax error in expression "ab ne ab": variable
references require preceding $
% expr {"ab" ne "ab"}
0
% expr "ab" ne "ab"
syntax error in expression "ab ne ab": variable
references require preceding $
% expr {ab} ne {ab}
syntax error in expression "ab ne ab": variable
references require preceding $
% expr {{ab} ne {ab}}
0
% expr {"ab"} eq {"ab"}
1
```

```
% expr {"ab"} == {"ab"}
1
% expr {"ab"} < {"ab"}
0
% expr {"aa"} < {"ab"}
1
% expr {"100"} < {"11"};# kein lexikogr. Vergleich
0
%
```

Grundsätzlich sei an dieser Stelle der Hinweis gegeben, daß man für Zeichenketten-Vergleiche das `string`-Kommando benutzen sollte, das für

```
string compare string1 string2
```

`-1` liefert, wenn `string1 < string2`, `0` bei Gleichheit und `+1` bei `string1 > string2`. Hier geht es jetzt weiter mit arithmetischen Ausdrücken.

- Mathematische Funktionen (Auswahl)
  - sin(x) Sinus von x
  - cos(x) Cosinus von x
  - tan(x) Tangens von x
  - exp(x) e^x
  - log(x) Natürlicher Logarithmus von x
  - pow(x,y) x^y
  - sqrt(x) Quadratwurzel aus x
  - abs(x) Betrag von x
  - rand() Zufallszahl im Intervall (0, 1)
  - double(x) Verwandelt x in Gleitkommazahl
  - int(x) Verwandelt x in Ganzzahl

3.4 Beispiel: Tcl als Addierer

```
set s1 ""; set s2 ""
puts "Bitte geben Sie den ersten Summanden ein:"
gets stdin s1
puts "Bitte geben Sie den zweiten Summanden ein:"
gets stdin s2
set result [expr $s1+$s2]
puts "Ergebnis: $result"
```

```
Bitte geben Sie den ersten Summanden ein:
5
Bitte geben Sie den zweiten Summanden ein:
9
Ergebnis: 14
```

3.5 Substitution und Befehlsausführung

Befehle in Tcl werden in zwei Phasen evaluiert. Zuerst findet eine *Analyse* (Parsing) statt, die rein syntaktisch erfolgt und nur Kommandonamen und Argumente liefert. Dabei findet aber bereits eine Substitution statt, z. B. wird der Text `$a` durch den Wert der Variablen a ersetzt, wobei der Parser nicht prüft, ob eine Variablenreferenz an dieser Stelle korrekt ist oder ob der gelieferte Wert an dieser Stelle paßt.

In der zweiten Phase wird der nach der Substitution gelieferte String semantisch als Kommando evaluiert, wobei das erste Wort der Befehlsname ist, dessen Existenz geprüft wird. Bei Erfolg wird die Kontrolle an diese Prozedur abgegeben, die in der Interpretation der Argumente völlig frei ist.

Diese Zweiteilung entspricht der Aufgabentrennung von Shell und Kommandos in UNIX, bei der bekanntlich die Shell einheitlich die Ein-/Ausgabeumlenkung regelt, jedes Kommando aber andere Optionen akzeptieren kann.

Variablen-Substitution

Die Zeichenfolge nach dem $-Zeichen wird als Variablenname interpretiert und durch den Inhalt der Variablen ersetzt
Beispiel:

```
foreach Zahl {1 2 3 4 5} {
  button .b$Zahl
}
```

erzeugt fünf Buttons mit den Namen `.b1`, `.b2` bis `.b5`.

Kommando-Substitution

Die Zeichenkette innerhalb eines Paars eckiger Klammern wird als Kommando (Tcl Skript) interpretiert und durch den Rückgabe-String des Kommandos ersetzt

```
set Kilogramm 20
set Pound [expr $Kilogramm*2.2046]
44.092
```

Backslash-Substitution

Einige Backslash-Sequenzen werden durch bestimmte Zeichen ersetzt

```
pack .basis .label1 .exponent .ergebnis \
  -side left -padx 1m -pady 2m
set Nachricht US-Dollar:\ \$1=EUR0.8084
US-Dollar: $1=EUR0.8084
```

Im 1. Beispiel ging es nur um die Verlängerung des Kommandos über zwei Zeilen. Das „Neue-Zeile-Zeichen“ (NL) und die führenden Leerzeichen der nächsten Zeile werden zu einem Leerzeichen gewandelt. Dies geschieht sogar vor der eigentlichen Analyse in einem Vorverarbeitungsschritt und stellt sicher, daß das erzeugte Leerzeichen als Argumenttrenner erkannt wird.

Im 2. Beispiel verhindert der Backslash vor dem Leerzeichen, daß `set` mehr als 2 Argumente bekommt (nimmt dem Leerzeichen die Bedeutung als Argumenttrenner) und sorgt dafür, daß $1 nicht als Variablenwert (Wert der Variablen 1) aufgefaßt wird.

Sonderzeichen II (nach Schenk)

- `$` Variablen-Substitution
- `[...]` Aufruf von Kommandos durch Kommando-Substitution
- `"..."` Anführungszeichen; notwendig für die Definition von Zeichenketten, die Leerzeichen enthalten
- `{...}` Zur Definition von Zeichenketten, innerhalb derer keine Substitution durchgeführt werden soll
- `\` Fortsetzung des Kommandos in der nächsten Zeile und Einleitung einer Backslash-Sequenz

Beispiele:

```
set a 5
set b 7
puts "$a + $b = [expr $a+$b]"
5 + 7 = 12
puts {$a + $b = [expr $a+$b]}
$a + $b = [expr $a+$b]
```

Gängige Backslash Ersetzungen

```
\$      $
\[      [
\]      ]
\{      {
\}      }
\\      \
\"      "
\n      newline (Zeilenumbruch 0xa)
\t      Tabulator
\a      Alert (Piepton 0x7)
```

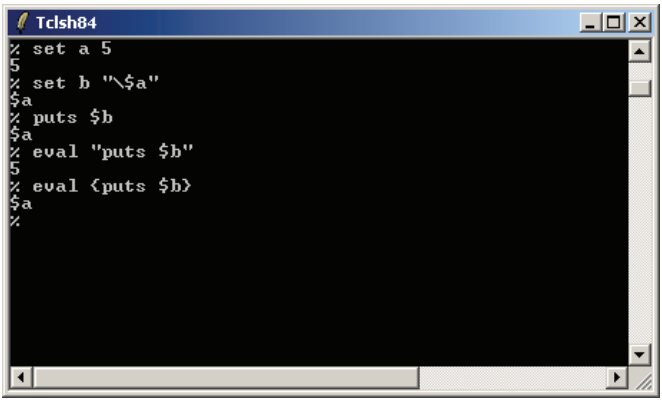
Allgemeine Hinweise zur Substitution

Geschweifte Klammern verhindern alle Arten der Substitution. Ausnahmen:

- Das einfache \ zur Kommandofortsetzung in der nächsten Zeile

- Kommando `eval` erzwingt Substitution jeder Zeichenkette
  `eval` *`string`*

Nur ein Substitutionsdurchgang, also keine Substitution von bereits Substituiertem!

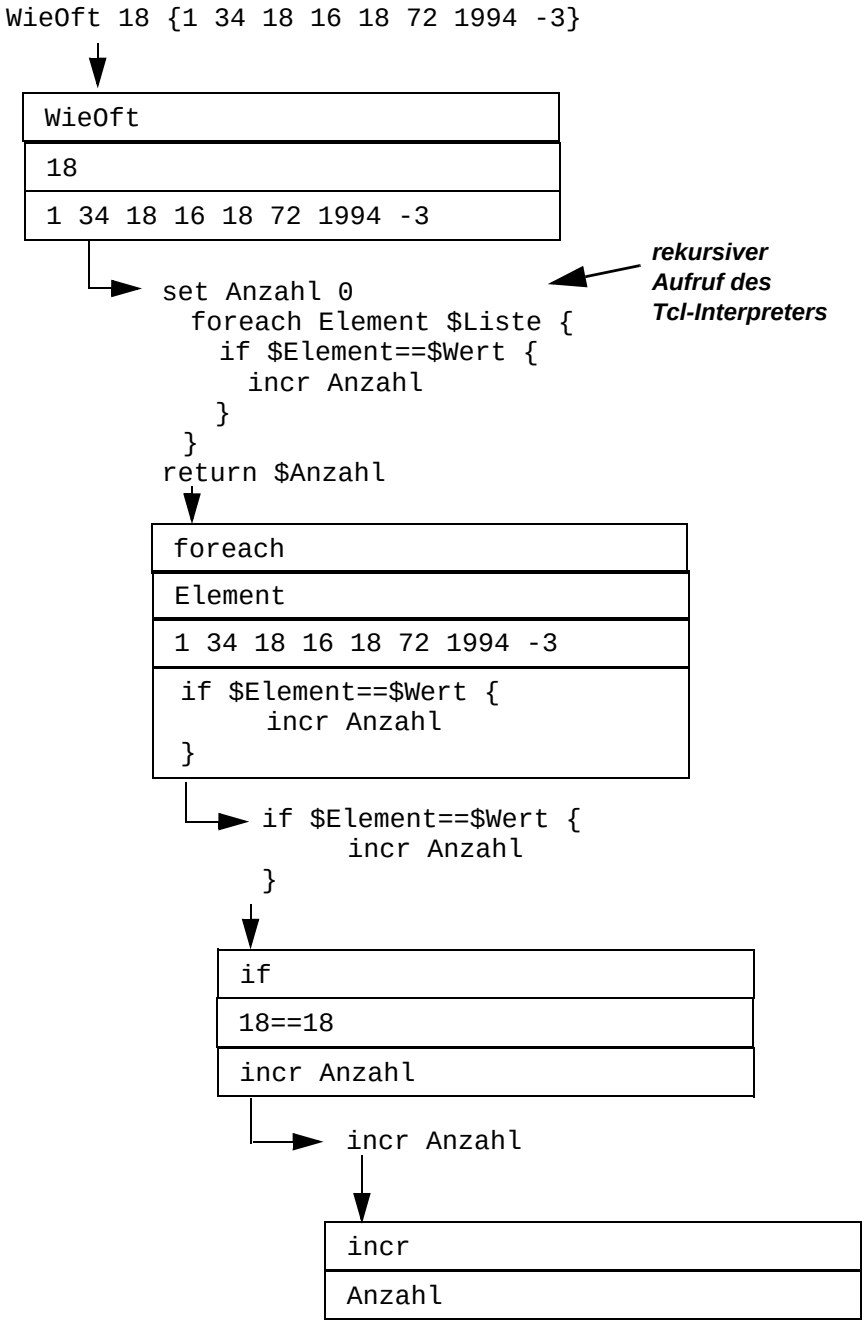


Das Klammern von Text mit {...} verhindert demnach Variablen-, Befehls- und Backslash-Substitution, einzige Ausnahme ist die Backslash-Newline-Substitution, die im Vorverarbeitungsschritt erkannt wird. Leerzeichen, Tabulatoren, Zeilenvorschübe und Semikolons bleiben innerhalb der geschweiften Klammern ohne Bedeutung und werden uninterpretiert übernommen.

Die wichtigste Anwendung der Interpretationsunterdrückung ist im Zusammenhang mit Prozedurvereinbarungen. Erst zur Laufzeit wird eine Parameterreferenz im Prozedurtext durch ein aktuelles Argument ersetzt. Das Beispiel aus [7] zählt das Auftreten eines Elements in einer Liste.

```
proc WieOft {Wert Liste} {
  set Anzahl 0
  foreach Element $Liste {
    if $Element==$Wert {
        incr Anzahl
    }
  }
  return $Anzahl
}
```

Bei einem Aufruf, z. B. mit `WieOft 18 {1 34 18 16 18 72 1994 -3}` ergibt sich die folgende Abarbeitung.



4 Fortsetzung Tcl und Einführung in Tk

4.1 Listen

Die folgenden Erläuterungen stammen wieder aus dem Tutorial von Schenk.

- Jede Variable kann als Liste aufgefasst werden
- Listenelemente werden durch Leerzeichen voneinander getrennt
- Listen können beliebig verschachtelt werden

```
set gruppe1a "Anna Paul Maria Georg"
set gruppe1b {Anna Paul Maria Georg}
set gruppe2a "\"Anna Schmidt\" \"Paul Krueger\" \
  \"Maria Reinhardt\" \"Georg Fuhrmann\""
set gruppe2b {{Anna Schmidt} {Paul Krueger} \
  {Maria Reinhardt} {Georg Fuhrmann}}
```

Wichtige Listen-Kommandos

```
lindex list i
# Gibt Listenelement mit Index i zurück
# ACHTUNG: Das erste Element hat Index 0

llength list
# Gibt Anzahl der Listenelemente zurück

lrange list i j
# Gibt die Listenelemente zwischen
# Index i und j (inklusive) als Liste zurück

lappend listVar arg ?...?
# Fügt neue Elemente an die Liste mit Variablennamen
# listVar an
```

```
linsert list i arg
# Fügt vor Position i das neue Element arg ein. Gibt
# die modifizierte Liste zurück
```

Beispiel: Listen-Kommandos

```
set l1 "Erwin Siegfried Bruno"
puts "Erstes Listenelement: [lindex $l1 0]"
=> Erstes Listenelement: Erwin
puts "Länge der Liste: [llength $l1]"
=> Länge der Liste: 3
puts "Ohne das letzte Element: \
[lrange $l1 0 [expr [llength $l1]-2]]"
=> Ohne das letzte Element: Erwin Siegfried
lappend l1 Rainer
set l1 [linsert $l1 2 Michael]
puts "Die modifizierte Liste: $l1"
=> Die modifizierte Liste: Erwin Siegfried Michael Bruno
Rainer
```

4.2 Kontrollstrukturen

- Bedingte Verzweigung: `if-else`
- Schleifen

  `while`

  `for`

  `foreach`

- Schleifenabbruch: `break` und `continue`

Bedingte Verzweigung

```
if {expression} {
    commands
} else {
    commands
}
```

- Wenn *expression* wahr ist, wird der `if`-Zweig ausgeführt, anderenfalls der `else`-Zweig
- `else`-Zweig ist optional

- Aufteilung auf mehrere Zeilen syntaktisch nicht notwendig
- Achtung: Leerzeichen auf keinen Fall weglassen!

Beispiel: Münzwurf

```
set r [expr rand()]
if {$r < 0.5} {
   puts "Kopf!"
} else {
   puts "Zahl!"
}
```

- Bildschirmausgabe: Entweder *Kopf!* oder *Zahl!* mit je 50% Wahrscheinlichkeit

Die while-Schleife:

```
while {expression} {
     commands
}
```

- Solange *expression* wahr ist, werden die *commands* im Schleifenrumpf immer wieder ausgeführt
- Aufteilung auf mehrere Zeilen syntaktisch nicht notwendig
- Achtung: Leerzeichen auf keinen Fall weglassen!

Beispiel: Zahlenraten

```
set rint [expr int(rand()*10)+1]
# generates random integer between 1 and 10
set uint 0
while {$uint != $rint} {
   puts "Bitte raten Sie eine Zahl zwischen 1 und 10:"
   gets stdin uint
}
puts "Richtig!"
```

Die for-Schleife

```
for {commands1} {expression} {commands2} {
   commands3
}
```

- Vor dem ersten Schleifendurchlauf: *commands1*
- Nach jedem Schleifendurchlauf: *commands2*
- Solange *expression* wahr ist, werden die *commands3* im Schleifenrumpf immer wieder ausgeführt
- Aufteilung auf mehrere Zeilen syntaktisch nicht notwendig
- Achtung: Leerzeichen auf keinen Fall weglassen!

Eine Zähl-Schleife mit for

```
for {set i 0} {$i < 5} {incr i} {
  puts $i
}
```

- Hinweis: Das Kommando `incr` *`int`* zählt die Integer-Variable *`int`* um eins hoch.

Die foreach-Schleife

```
foreach var list {
  commands
}
```

- Die Liste *list* wird durchiteriert: Je Schleifendurchlauf wird der Variable *var* das jeweils nächstfolgende Listenelement zugewiesen
- In jedem Schleifendurchlauf: *commands* werden ausgeführt
- Aufteilung auf mehrere Zeilen syntaktisch nicht notwendig
- Achtung: Leerzeichen auf keinen Fall weglassen!

Beispiel zu foreach

```
set gruppe "Hans Helga Martin"
foreach mitglied $gruppe {
  puts $mitglied
}
```

Schleifenkontrolle mit `break` und `continue`

- `break`: Schleife wird verlassen, Sprung zum ersten Befehl nach Schleifenende
- `continue`: Sofortiger Start eines neuen Schleifendurchlaufs; die Kommandos zwischen `continue` und dem Schleifenende werden übersprungen

Beispiel zu `break` und `continue`

```
set maxIter 10
for {set i 0} {$i < $maxIter} {incr i} {
  if {$i == 7} break
  if {$i%2 == 0} continue
  puts $i
}
```

Die Bildschirmausgabe liefert die Zahlen 1, 3, 5.

4.3 Prozeduren

Problem: Wiederkehrende Operationen müssen immer wieder neu kodiert werden, d.h. Code wird aufgebläht, unübersichtlich, fehleranfällig.

Lösung: Prozedurale Programmierung, d. h. Aufteilung des Codes in sog. Prozeduren, die Teilaufgaben übernehmen

Syntax

```
proc name {?params...?} {
    commands
}
```

- *name*: Name der Prozedur unter dem die Prozedur im Skript als Kommando genutzt werden kann.
- *params*: Liste der Übergabe-Parameter die innerhalb der Prozedur unter ihrem jeweiligen Namen als Variable mit den Übergabewerten zur Verfügung stehen
- Prozeduren können keine, einen oder mehrere Parameter haben
- **commands**: Kommandos, die innerhalb der Prozedur ausgeführt werden

Prozeduren: Syntax (Forts.)

Allg. Hinweise:

- Aufteilung auf mehrere Zeilen syntaktisch nicht notwendig
- Achtung: Leerzeichen auf keinen Fall weglassen!
- Alle außerhalb der Prozedur mit `set` definierten Variablen sind innerhalb der Prozeduren zunächst unbekannt
- Besondere Kommandos innerhalb von Prozeduren:
  - Kommando `global`:
    `global` *`varname ?varname...?`*
    zur Deklaration der externen Variablen, die auch innerhalb der Prozedur nutzbar sein sollen
  - Aufruf von `global` jederzeit und mehrfach innerhalb der prozedur möglich
  - Kommando `return`:
    `return ?`*`list`*`?`
    beendet die Ausführung der Prozedur; *list* ist die Rückgabeliste der Prozedur

Beispiel: Prozeduren (Schenk)

```
set c 5
proc plusC {a b} {
  global c
  set result [expr $a+$b+$c]
  return $result
}
puts [plusC 3 4]
=>
12
```

Zweites Beispiel (Wegner/Schweinsberg)

```
% proc max2 {a b} {if {$a > $b} {return $a} else {return $b}}
% max2 30 50
50
% max2 40 20
40
% max2 {20 "max2 30 40" }
```

```
wrong # args: should be "max2 a b"
% max2 20 "max2 30 40"
max2 30 40
% max2 20 [max2 30 40]
40
% proc max4 { a b c d } { if {[max2 $a $b] > [max2 $c $d]}
     {return [max2 $a $b]} else {return [max2 $c $d]} }
% max4 20 50 70 30
wrong # args: no script following "{[max2 $a $b] > [max2 $c
$d]}" argument
% proc max4 { a b c d } { if {[max2 $a $b] > [max2 $c $d]} {
     return [max2 $a $b]} else {return [max2 $c $d]} }
% max4 20 50 70 30
70
% proc neumax4 { a b c d } {
     return [max2 [max2 $a $b] [max2 $c $d] ] }
% neumax4 20 50 70 30
70
% proc neuneumax4 { a b c d } {
     return [max2 $a [max2 $b [max2 $c $d]]] }
% neuneumax4 20 50 70 30
70
%
```

Beispiel: Fakultät rekursiv berechnen (Wegner)

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

fac 4
=> 24
```

Hinweis: Programm inkl. Aufruf **`fac 4`** in Datei schreiben, z. B. **`testf`**. Dann **`tclsh`** aufrufen und **`source testf`** eingeben.

Beispiel: Prozeduren und Listen (Schenk)

```
set group1 "Britta Anna Hanna"
set group2 "Rainer Michael Paul Mark"
set group3 "Helga Inga Tanja Katja"

set allGroups "\{$group1\} \{$group2\} \{$group3\}"
puts $allGroups
```

```
proc reorderGroup group {
   set newGroup ""
   foreach elem $group {
      set newGroup [linsert $newGroup 0 $elem]
   }
   return $newGroup
}

set count 0
foreach group $allGroups {
   incr count
   set newGroup [reorderGroup $group]
   puts "\[$count\] $newGroup"
}
```

Bildschirmausgabe:

```
=>
{Britta Anna Hanna} {Rainer Michael Paul Mark}
{Helga Inga Tanja Katja}
[1] Hanna Anna Britta
[2] Mark Paul Michael Rainer
[3] Katja Tanja Inga Helga
```

4.4 Prozeduren (Teil II)

Es geht um Default-Werte für Parameter, eine variable Anzahl von Parametern, um Variablennamen als Parameter mit dem `upvar`-Kommando.

Default-Werte für Parameter:

```
proc plusC {a b {c 5}} {
   return [expr $a+$b+$c]
}
puts [plusC 3 4]
```

Zu beachten bei Verwendung von Default-Werten:

- Parameter-Liste muss in geschweiften Klammern stehen!
- Definition eines Parameters mit Default-Wert:
  `{`*`paramName defaultValue`*`}`

- Nachdem ein Parameter mit Default-Wert definiert wurde, müssen alle nachfolgenden Parameter ebenfalls einen Default-Wert erhalten

Variable Anzahl von Parametern

```
proc argTest {a b args} {
   foreach param {a b args} {
      puts "$param = [set $param]"
   }
}

argTest Dies ist ein Test!
```

Bildschirmausgabe:

```
=>
a = Dies
b = ist
args = ein Test!
```

Zu beachten:

- Das Schlüsselwort `args` muss am Ende der Parameter-Liste stehen!
- `args` nimmt alle überschüssigen Parameter als Liste auf

Variablennamen als Parameter

```
upvar #0 varName localvar
```

`upvar #0` wird innerhalb von Prozeduren gebraucht, um globale Variablen (im Kontext `0`, daher `#0`) unter einem lokalen Variablennamen verfügbar zu machen

Einsatzgebiete:

- Zeitersparnis bei umfangreichen Variableninhalten
- Ggf. einfacher, bestimmte Variablen direkt zu verändern, als alles im Rückgabe-String der Prozedur unterzubringen

Beispiel zu `upvar`

```
set globalVar 0; puts "globalVar = $globalVar"
```

```
proc setVarTo5 {varName} {
   upvar #0 $varName localvar
   set localvar 5
}

setVarTo5 globalVar; puts "globalVar = $globalVar"
```

Bildschirmausgabe:

```
=>
globalVar = 0
globalVar = 5
```

Hinweis (Wegner): Die Angaben im Schenk-Tutorial sind richtig, aber treffen den Sachverhalt nicht voll. Es geht um einen „Call by reference“, wie auch Ousterhout selber schreibt ([7], S. 89). Den Zugriff auf eine globale Variable würde man ja auch über das `global`-Kommando erreichen.

Das setzt aber voraus, daß man zum Zeitpunkt des Prozedurentwurfs schon weiß, wie die globalen Variablen heißen und was sie bedeuten. Das ist nicht die Regel. Zudem gilt es als schlechter Programmierstil, globale Variable aus einer Prozedur heraus zu modifizieren, allenfalls wird man einen lesenden Gebrauch tolerieren.

Wie im Schenk-Beispiel zu sehen, umgeht man dies, indem man mit `upvar` zur Laufzeit einen übergebenen Variablennamen mit dem lokalen Namen (dem alias) verknüpft. Noch wichtiger, ohne diese Konstruktion kann man keine Arrays als Parameter übergeben, da sich hinter dem Namen nur eine Referenz auf den Array verbirgt (mehr unten).

Neben den absoluten Angaben (`#0` ist äußerster Kontext) sind auch relative Kontexte möglich (z. B. `-2`). Wir gehen darauf nicht ein. Anwendungen zu `upvar` werden wir in Kapitel 9 kennenlernen.

Ende Hinweis

4.5 Arrays

- Variable mit Index in runden Klammern, z. B. `a(1)`
- In Tcl: Index kann beliebige Zeichenkette sein, z. B. `a(gelb)`
  Wegner: sog. *assoziative Arrays*
- Spezielle Anwendung: Darstellung mehrdimensionaler Matrizen
  - Anzahl der Dimensionen entspricht Anzahl der Indizes
  - Indizes durch Komma getrennt, keine Leerzeichen!
  - Beispiel: `a(1,5,2)`
  - Bei Zugriff beachten: Nur die Array-Elemente existieren, die mit `set` gesetzt wurden

Das `array`-Kommando

```
array exists arrayName

# Rückgabewert 1 falls arrayName der Name einer Array-
Variablen ist

array names arrayName

# Gibt alle für das Array definierten Indizes als Liste
zurück

array size arrayName

# Gibt Größe des Arrays zurück
```

Kommando `array`: Beispiel für Tcl-Kommandos, bei denen der erste Parameter die Funktion des Kommandos bestimmt.

Beispiel 1 zu Arrays (Schenk)

```
set a(1,1) 3.4
set a(1,2) 1.2
set a(2,2) -1.4
set a(1,1,3) 5.4
puts [array names a]
=>
2,2 1,1,3 1,1 1,2
```

Wie Ousterhout zugibt, sind die Arrays nicht wirklich mehrdimensional.

- Beliebige Arten von Indizes können in einem Array gemischt werden
- Ein Variablenname kann nicht gleichzeitig für eine einfache Tcl-Variable und ein Tcl-Array genutzt werden

Beispiel 2 zu Arrays

```
set augenfarbe(Helga) graugruen
set augenfarbe(Max) blau
set augenfarbe(Gerd) braun
set augenfarbe(Inge) gruen

foreach index [array names augenfarbe] {
   puts "$index: $augenfarbe($index)"
}
=>
Helga: graugruen
Gerd: braun
Max: blau
Inge: gruen
```

4.6 Kontrollstrukturen (Teil II)

- Mehrfachverzweigung: `switch`
- Fehlerbehandlung: `catch`

Mehrfachverzweigung: `switch`

```
switch ?flags? string {
   pat1 {commands1}
   pat2 {commands2}
   ...
}
```

Es werden die *commands* ausgeführt, deren zugehöriges Muster *pat* mit *string* übereinstimmt. In dieser Schreibweise: Keine Substitution innerhalb des switch-Körpers. Aufteilung auf mehrere Zeilen syntaktisch nicht notwendig, Leerzeichen auf keinen Fall weglassen!

Beispiel 1 zu `switch`

```
set a 3
switch $a {
1 {puts Eins}
2 {puts Zwei}
3 {puts Drei}
}
=>
Drei
```

`switch`-Ergänzungen

- Schlüsselwort `default` als letztes Pattern: Fängt alle nicht aufgeführten Ausprägungen von `string` auf
- `-` anstelle von commands: Keine Aktion

```
set a test
switch $a {
  1 {puts Eins}
  2 -
  default {puts "Weder eins noch zwei"}
}
```

Bildschirmausgabe

```
==> Weder eins noch zwei
```

`switch`-Flags

Mögliche Werte für *flags*:

`-exact`	Exakte Übereinstimmung (default)
`-glob`	glob-style pattern matching
`-regexp`	regular expression pattern matching
`--`	Notwendig, wenn string mit `-` beginnen kann (muss als letztes Flag stehen)

glob-style pattern matching:

`*` beliebige Zeichenkette

`?` beliebiges Zeichen

`[`*`chars`*`]` Ein Zeichen aus *chars*

Beispiel 2 zu `switch`

```
set a test

switch -glob $a {
   1 {puts Eins}
   2 {puts Zwei}
   t* {puts "t als erster Buchstabe"}
}
```

Bildschirmausgabe:

```
=> t als erster Buchstabe
```

Fehlerbehandlung: `catch`

`catch {`*`commands`*`} `*`?resultVar?`*

- Normalerweise: Tcl steigt bei jedem Syntaxfehler im Skript aus; `catch` verhindert das
- Rückgabewert von `catch`: Signalisiert Auftreten eines Fehlers (0 bedeutet kein Fehler), aber
  - wenn *commands* mit `return`, `continue` oder `break` verlassen werden, liefert `catch` ebenfalls einen Wert ungleich 0 zurück
- *resultVar*: Enthält den Rückgabe-String eines `return` in *commands* oder des letzten Kommandos in *commands*

Rückgabewerte von `catch` (Beispiel aus Welch [11], S. 84)

```
switch [catch {
   command1
   command2
   ...
} result] {
   0 { # Normal completion }
```

```
    1 { # Error case }
    2 { return $result ;# return from procedure}
    3 { break ;# break out of the loop}
    4 { continue ;# continue loop}
    default { # User-defined error codes }
}
```

Beispiel: Tcl als Taschenrechner

```
set inp ""
set catchReturnValue 1
while {$catchReturnValue != 0} {
   puts "Bitte gueltige Tcl-Expression eingeben:"
   gets stdin inp
   set catchReturnValue [catch {expr $inp} calcResult]
}
puts "Ergebnis: $calcResult"
```

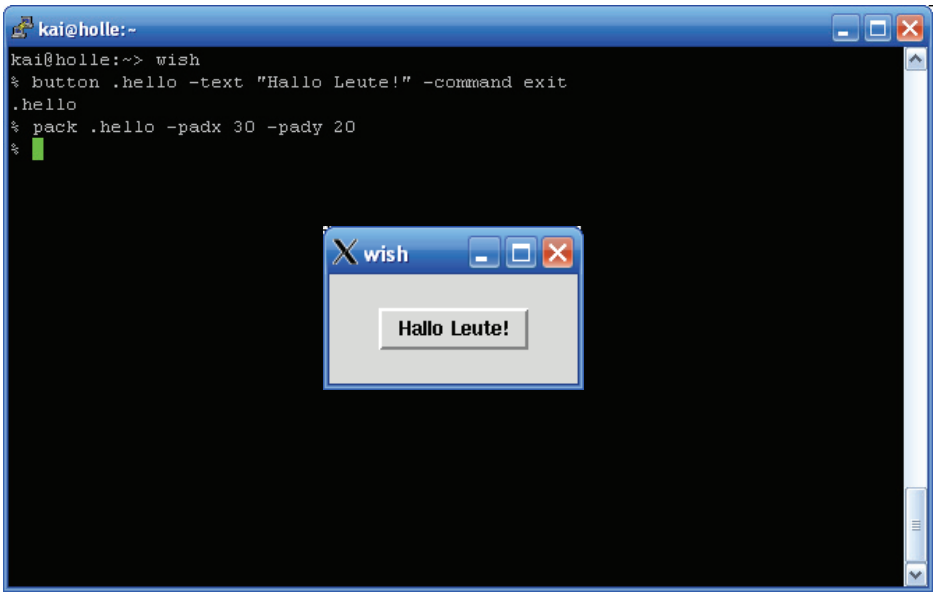
- `catchReturnValue` erhält Rückgabe-String von `catch`
- `calcResult` erhält Rückgabe-String von `{expr $inp}`

Bildschirmausgabe:

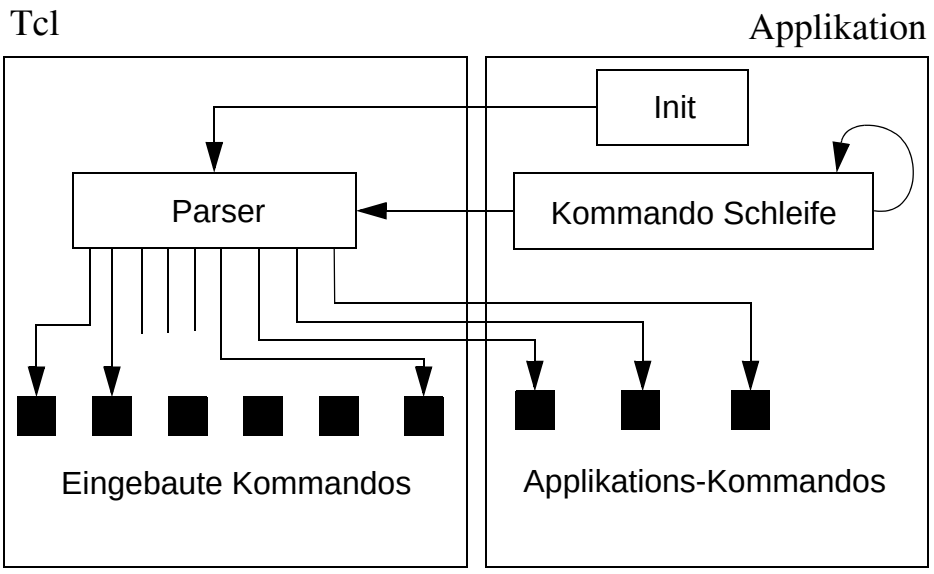
```
=>
Bitte gueltige Tcl-Expression eingeben:
adfs
Bitte gueltige Tcl-Expression eingeben:
sin(adsfaf)
Bitte gueltige Tcl-Expression eingeben:
sin(2.0)
Ergebnis: 0.909297426826
```

4.7 Erste Schritte mit Tk

Vorab als Motivation den berühmten „Hallo Leute“-Knopf als Tk-Skript für die Window-Shell `wish`.



Einbettung von Tcl in Applikationen



- Applikation erzeugt Tcl-Skripte
- Tcl interpretiert Skripte, ruft Kommandoprozeduren auf mit argc, argv

- Applikation erweitert eingebauten Kommandosatz
  - Definiert neue Objekttypen in C
  - Implementiert primitive Operationen auf Objekten als neue Tcl-Kommandos
  - Baut komplexe Features mit Tcl-Skripten

4.8 Der Tk-Toolkit

Das Problem:

- Es ist zu schwierig, Anwendungen mit schönen Benutzerschnittstellen zu schreiben.

Die falsche Lösung:

- C++, objekt-orientierte Toolkits
- Nur geringe Verbesserung (10-20%?):
  weiterhin Programmierung auf tiefer Ebene

Die richtige Lösung:

- Die Ebene der Programmierung anheben
- Schnittstellen erzeugen mit Tcl-Skripten

Erzeugen von Schnittstellen mit Tk

Widgets/Fenster haben Pfadnamen:

```
.dlg.quit
```

Erzeugen eines Widgets mit Kommandos, die wie ihre Klasse heißen:

```
button .dlg.quit -text Quit \
  -foreground red -command exit
```

Dem Geometrie-Manager sagen, wo das Widget angezeigt werden soll:

```
place .dlg.quit -x 0 -y 0
```

oder

```
pack .dlg.quit -side bottom
```

Andere Tk-Eigenschaften

Widgets manipulieren mit Widget-Kommandos:

```
.dlg.quit flash
.dlg.quit configure -relief sunken
```

Tcl für Verknüpfungen benutzen:

- Buttons, Menüeinträge lösen Tcl-Kommandos aus.
- Scrollbars und Listboxes kommunizieren mit Tcl.
- Neue Ereignisverknüpfungen können in Tcl definiert werden.
- Auswahl, Fokus in Tcl verfügbar.

Tk stellt auch C-Schnittstellen zur Verfügung:

- Erzeugen neuer Widget-Klassen
- Erzeugen neuer Geometrie-Manager

Was ist eine Tk-basierte Applikation?

1. Der Tcl-Interpreter.

2. Der Tk-Toolkit.

3. Applikationsspezifischer C-Code (Primitiva!):

   - Neue Objekttypen
   - Neue Widgets

4. Tcl-Skripten (setze Primitiva zusammen):

   - Baue die Benutzerschnittstelle zusammen
   - Reagiere auf Ereignisse

Die einfachste Tk-Anwendung: wish

Kein C-Code außer dem Kommandozeileninterpreter.

Viele Anwendungen als wish-Skripte schreibbar

- Hello, world:

```
label .hello -text "Hello, world"
pack .hello
```

- Einfacher Dateibrowser mit nur 30 Zeilen!

Dateibrowser:

```
#!/usr/bin/wish
listbox .list -yscroll ".scroll set" \
              -relief raised -width 20 -height 20
pack .list -side left
scrollbar .scroll -command ".list yview"
pack .scroll -side right -fill y
if {$argc > 0} {
  set dir [lindex $argv 0]
} else {
  set dir "."
}
foreach i [exec ls -a $dir] {
  .list insert end $i
}
bind .list <Double-Button-1> {
  browse $dir [selection get]
}
bind .list <Control-c> {destroy .}
focus .list

proc browse {dir file} {
  global env
  if {$dir != "."} {set file $dir/$file}
  if [file isdirectory $file] {
     exec browse $file &
  } else {
     if [file isfile $file] {
        exec xedit $file &
     } else {
        puts stdout "can’t browse $file"
     }
  }
}
```

Perspecta Presents!

Kommerzielles Präsentationspaket (damals als Alternative zu Powerpoint gedacht). Hat keine Bedeutung mehr, hier als Anwendungsbeispiel zitiert:

- Präsentation = Folge von Folien
- Text, Graphiken, Bilder
- Hintergrund, Folien, Notizen
- Postscript-Ausgabe, on-line Folienshow

Implementiert mittels Tcl und Tk:

- 29000 Zeilen neuen C-Code
- 1 neues Widget zur Anzeige von Folien
- ~30 andere Tcl Kommandos zur Manipulation von Präsentationen
- 11000 Zeilen Tcl-Skript

Gebrauch von Tcl in Perspecta Presents!

1. Powertext: Von Tcl-Skript erzeugter Text, nicht vom Benutzer eingetippt.

- Foliennummer
- Listennummer
- Werte aus Datenbank ergänzt?

2. Dateiformat = Tcl-Skript. Zum Laden nur die Datei ausführen.

3. Auswahl ausgetauscht als Tcl-Skript (ausgewähltes Kopieren von Hintergrund, Sichten, usw.)

4. Undo/redo:

- Undo/redo Skript Paare werden in Logdatei gerettet
- unbegrenztes undo/redo
- Recovery (Neuaufsetzen) nach Absturz

5. Folienshows, etc. etc.

4.9 Anwendungen produzieren

Das Problem:

- Einzige Kommunikation zwischen Anwendungen ist mittels Selektion.
- Resultat: monolithische Applikationen

Die Lösung: `send` Kommando

- `send appName command`
- Implementiert mittels X11 Eigenschaften
- Jede Tk-Anwendung kann alles in einer anderen Tk-Anwendung aufrufen: Schnittstelle oder Aktionen
- Resultat: mächtiges Kommunikationsmittel

Beispiele:

- Debugger: sendet Kommando an Editor zum Herausheben einer gerade ausgeführten Zeile
- Benutzungsschnittstelleneditor: sendet Kommando um das Interface einer laufenden Anwendung zu verändern
- Multimedia: Senden eines Stücks, Abspielkommandos für Audio- und Videoanwendungen
- Spreadsheets (Tabellenkalkulation): eine Zelle sendet Kommandos an die Datenbank zum Holen aktueller Werte

Revolutionäre Resultate:

- Zusammenbau komplexer Systeme als Kollektion spezialisierter, aber wiederverwendbarer Hypertools
- Aktive Objekte leicht zu erzeugen: eingebettete Tcl-Kommandos. Hypertext, Hypermedia einfach.

4.10 Status

- Quelltext und Dokumentation frei verfügbar:
  `http://www.tcl.tk/software/tcltk/`
- Große Benutzergemeinde: 10000 - 50000 im Januar 1994
- Viele kommerzielle Produkte
- Newsgroup: comp.lang.tcl
- Bücher, darunter von Ousterhout selbst bei Addison-Wesley

Nachteile:

- Zwang, eine neue Sprache lernen zu müssen; Substitutionsregeln für viele verwirrend.
- Interpretierende Sprache hat Leistungsgrenzen (aber erstaunlich hohe)
- C-Schnittstelle inkompatibel mit Xt, Motif library

5 Schnittstellen bauen mit Tcl und Tk

Überblick

- Grundstrukturen: Fenster, Widgets, Prozesse
- Widgeterzeugungs-Kommandos.
- Geometrie Management: der „placer“ und der „packer“.
- Widget Kommandos.
- Anbindungen (bindings).
- Andere Kommandos: send, focus, selection, window manager, grabs.
- 2 Beispiele: Dialog box, Browser.

5.1 Aufbau einer Tk-Applikation

1. Widget Hierarchie.

2. Ein Tcl Interpreter.

3. Ein Prozeß.

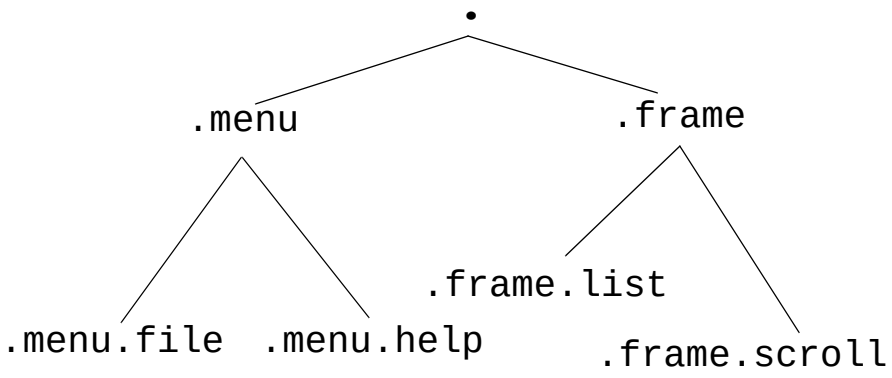
(Man kann mehr als 1 Anwendung in einem Prozeß haben)

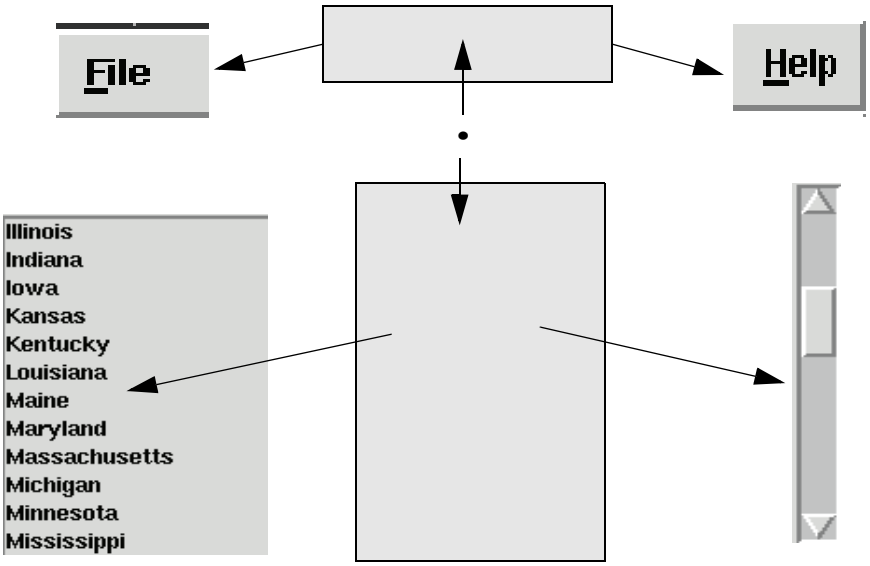
Widget = Fenster mit einem besonderen „look and feel“.

Von Tk angebotene Widget Klassen:

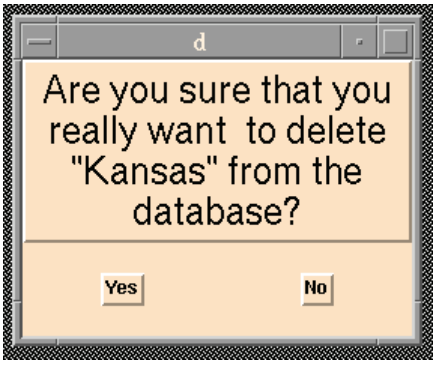
Frames	Menubuttons	Canvases
Labels	Menus	Scrollbars
Buttons	Messages	Scales
Checkbuttons	Entries	Listboxes
Radiobuttons	Texts	Toplevels

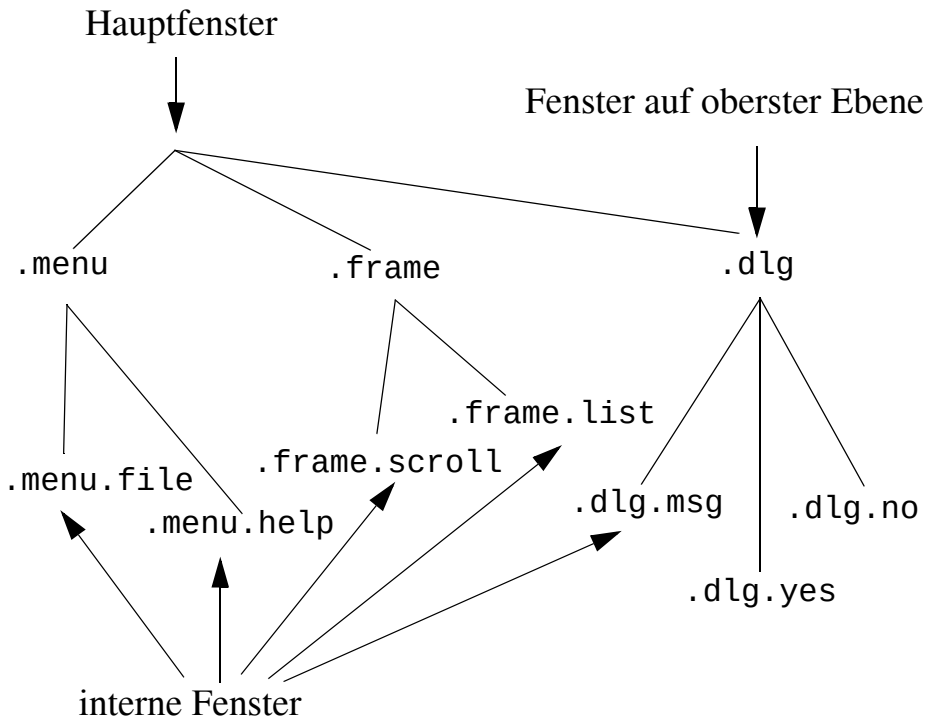
5.2 Die Widget Hierarchie





Arten von Fenstern:





Tcl-Skript für Staatenliste

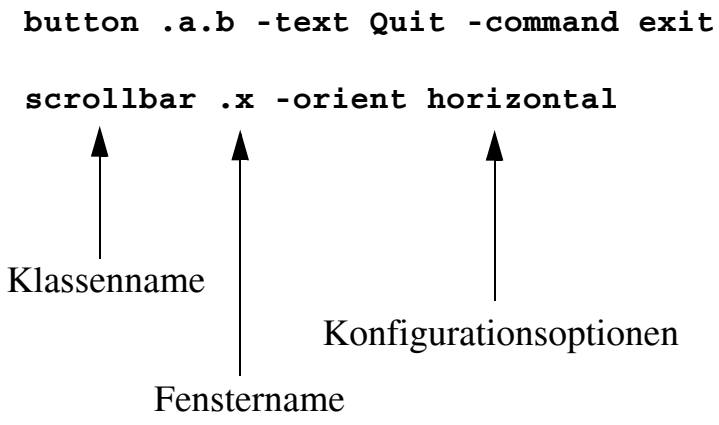
```
#!/bin/sh
# the next line restarts using wish \
exec wish "$0" "$@"
frame .menu -relief raised -borderwidth 2
pack .menu -side top -fill x
set m .menu.file.m
menubutton .menu.file -text "File" -menu $m\
  -underline 0
menu $m
$m add command -label "Quit" -command "exit"
pack .menu.file -side left
set m .menu.help.m
menubutton .menu.help -text "Help" -menu $m\
  -underline 0
menu $m
$m add command -label "Help" -command "puts NYI"
pack .menu.help -side right
frame .frame -relief raised -borderwidth 2
```

```
pack .frame -side top -expand yes -fill y
scrollbar .frame.scroll -command \
  ".frame.list yview"
listbox .frame.list -yscroll ".frame.scroll set"\
  -setgrid 1 -height 12
pack .frame.scroll -side right -fill y
pack .frame.list -side left -expand 1 -fill both
.frame.list insert 0 Alabama Alaska Arizona\
Arkansas California Colorado Connecticut Delaware\
Florida Georgia Hawaii Idaho Illinois Indiana Iowa\
Kansas Kentucky Louisiana Maine Maryland\
Massachusetts Michigan Minnesota Mississippi\
Missouri Montana Nebraska Nevada "New Hampshire"\
"New Jersey" "New Mexico" "New York"\
"North Carolina" "North Dakota" Ohio Oklahoma \
Oregon Pennsylvania "Rhode Island" "South Carolina"\
"South Dakota" Tennessee Texas Utah Vermont\
Virginia Washington "West Virginia" Wisconsin\
Wyoming
```

5.3 Widgets erzeugen

- Jedes Widget hat eine Klasse: button, listbox, scrollbar, etc.
- Ein Tcl Kommando nach jeder Klasse benannt, um Instanzen zu erzeugen

```
button .a.b -text Quit -command exit
scrollbar .x -orient horizontal
```



Konfigurationsoptionen

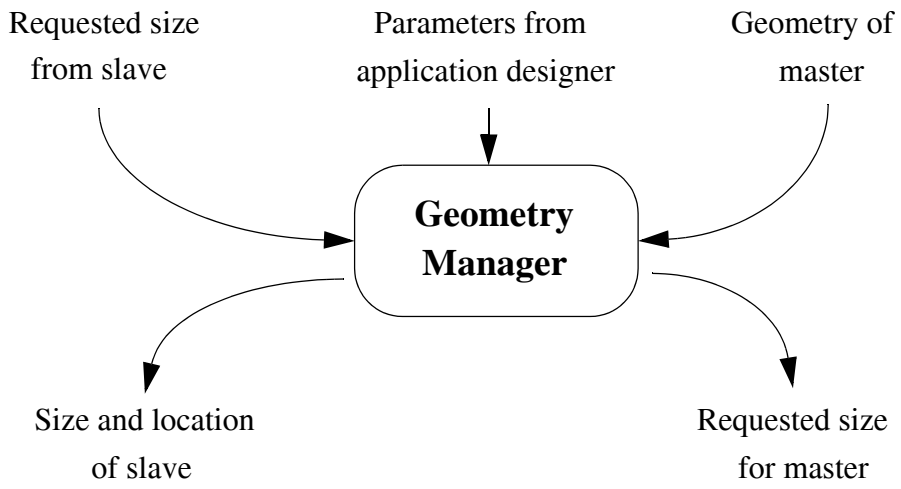
- Definiert durch die Klasse. Für Knöpfe (buttons):

`activebackground`	`cursor`	`relief`
`activeforeground`	`disabledforeground`	`state`
`anchor`	`font`	`text`
`background`	`foreground`	`textvariable`
`bitmap`	`height`	`width`
`borderwidth`	`padx`	
`command`	`pady`	

- Wird nichts in der Kommandozeile angegeben, dann aus Options-Datenbasis:
  - Geladen aus `RESOURCE_MANAGER` Eigenschaften oder `.Xdefaults` file.
  - Kann gesetzt und abgefragt werden mit Tk-Kommandos: `option add *Button.relief sunken`
- Wenn nicht in der Optionen-Datenbasis, Voreinstellung (default) der Klassenbibliothek benutzen (Ousterhout: defaults are reasonable!).

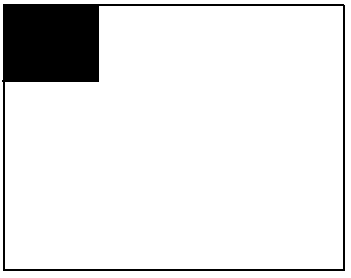
5.4 Geometrie-Management

- Widgets steuern nicht ihre eigene Positionen und Größen; Geometrie Manager tun das.
- Widgets erscheinen noch nicht einmal auf dem Bildschirm bis sie von einem Geometrie Manager „gemanaged“ werden.
- Geometrie Manager = Algorithmus, der für die Plazierung von abhängigen Fenstern relativ zum Hauptfenster sorgt.

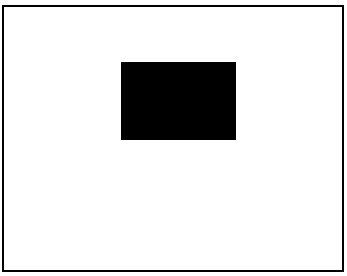


5.4.1 Der Plazierer (placer)

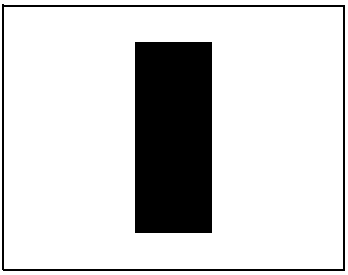
- Einfach, aber nicht sehr mächtig.
- Jedes abhängige Objekt (slave) wird individuell bezogen auf das kontrollierende Objekt (master) gelegt.
- Eine Anker-Angabe legt fest, auf welche Seite oder Ecke des zu plazierenden Objekts sich die angegebenen Koordinaten beziehen soll.
- Relative Positions- oder Größenangabe sind möglich (`relhight 0.5` bedeutet halbe Höhe des verfügbaren Platzes).



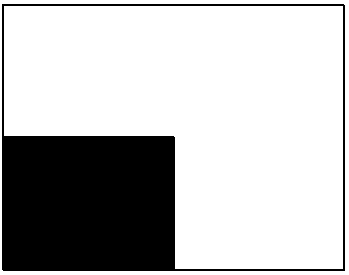
```
place .x -x 0 -y 0
```



```
place .x -relx 0.5 \
      -y 1.0c -anchor n
```



```
place .x -relx 0.5 \
      -rely 0.5 -height 3c \
      -anchor center
```



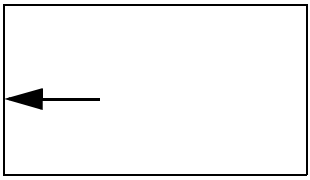
```
place .x -relheight 0.5 \
      -relwidth 0.5 \
      -relx 0 -rely 0.5
```

5.4.2 Der Packer (packer)

- Viel mächtiger als der Plazierer.
- Arrangiert Gruppen von abhängigen Objekten.
- Legt abhängige Objekte entlang den Kanten des noch verfügbaren leeren Raums des Hauptfensters.

Für jedes abhängige Objekt
mache nacheinander:

1. Wähle eine Seite des Hauptfensters



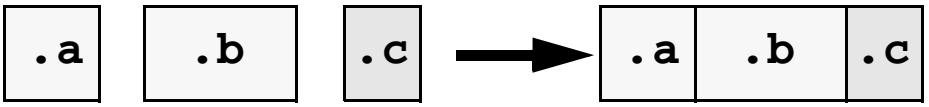
2. Schneide ein Rechteck ab für das abhängige Objekt

3. Laß das abhängige Objekt wachsen, um den Rahmen zu füllen

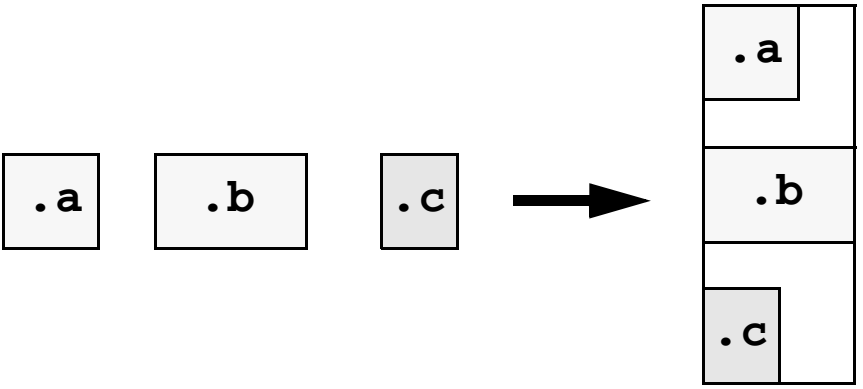
4. Stelle das abhängige Objekt in den Rahmen

Packer Beispiele

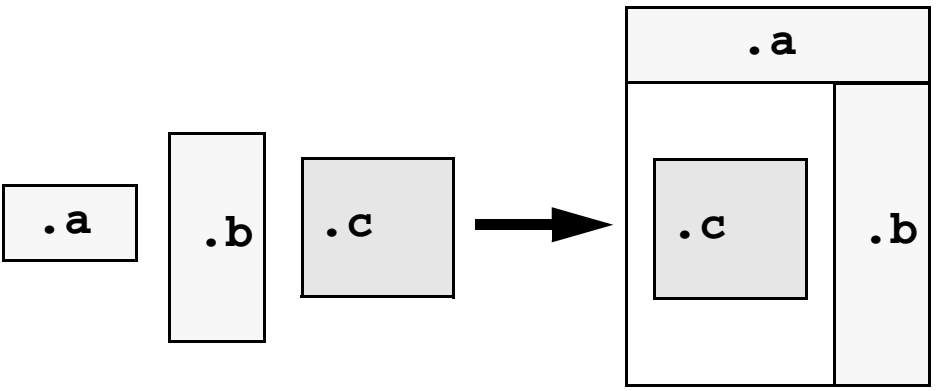
```
pack .a -side left
pack .b -side left
pack .c -side left
```




```
pack .a -side top -anchor w
pack .b -side top -anchor w -pady .5c
pack .c -side top -anchor w
```




```
pack .a -side top -fill x
pack .b -side right -fill y
pack .c -padx 0.5c -pady 1c -fill both
```



Vorteile des Packers

Abhängigkeiten zwischen „Slaves“ werden berücksichtigt (constraint-like):

- Zeilen- und Spaltenanordnungen leicht herzustellen.
- Paßt Anordnung an, falls Slave eine andere Größe verlangt.

Anforderungen seitens des „Master’s“

- Gerade groß genug, um alle abhängigen Objekte unterzubringen
- Paßt sich an, wenn abhängige Objekte eine andere Größe verlangen.
- Erlaubt ein hierarchisches Geometriemanagement.

5.5 Widget-Kommandos

- Ein Tcl-Kommando für jedes Widget, benannt nach dem Widget-Pfadnamen.
- Wird benutzt um das Widget zu rekonfigurieren und zu manipulieren.

```
button .a.b
.a.b configure -relief sunken
.a.b flash


scrollbar .x
.x set 100 10 5 14
.x get
```

- Widget-Kommando wird automatisch gelöscht, wenn das Widget entfernt wird.
- Prinzip: alle Zustände sollten jederzeit lesbar und modifizierbar sein.

5.6 Verbindungen

Frage: Wie können Widgets mit der Anwendung und untereinander zusammenarbeiten?

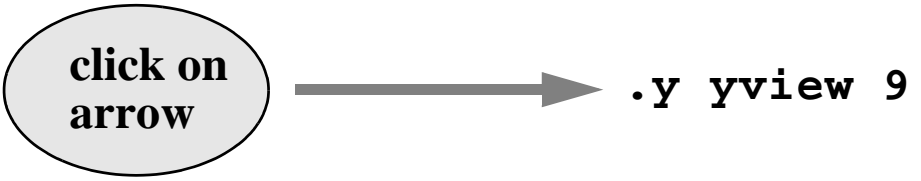
Antwort: Tcl Kommandos.

- Widget Aktionen sind Tcl Kommandos:

```
button .a.b -command exit
```

- Widgets benutzen Tcl Kommandos, um miteinander zu kommunizieren:

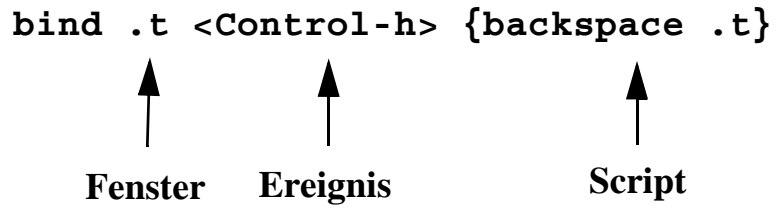
```
scrollbar .x -command ".y yview"
```



- Applikationen benutzen Widget Kommandos, um mit Widgets zu kommunizieren.

5.7 Bindings (Verknüpfung mit Ereignissen)

Verknüpfe Tcl Skripte mit X-Ereignissen

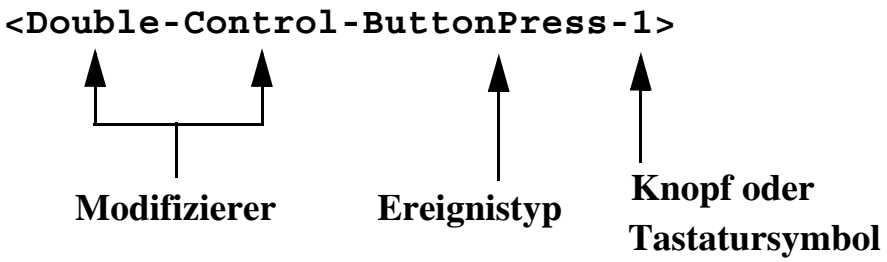


Kann ein oder mehrere Fenster auswählen:

- Einzelnes Fenster: **`.t`**

- Alle Fenster in der Klasse: **`Text`**
- Alle Fenster: **`all`**

Angabe von Ereignissen:




```
<3>
<Any-KeyPress>
a
```

% Substitutionen in der Verknüpfung mit Skripten:

- Koordinaten des Ereignisses: **`%x`** und **`%y`**
- Fenster: **`%W`**
- Vom Ereignis abgeliefertes Zeichen: **`%A`**
- ...

Beispiele:

```
bind .c <B1-Motion> {move %x %y}
bind .t <Any-KeyPress> {insert %A}
bind all <Help> {help %W}
```

Nur eine Verknüpfung löst aus (triggers) bei einem Ereignis:

- Fenster vor Klasse vor alle.
- Spezifischster Adressat erhält höchste Priorität.

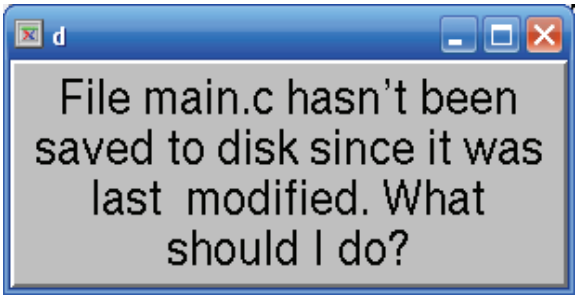
5.8 Andere Tcl Kommandos

- Die Auswahl:
```
selection get
selection get FILE_NAME
```
- Verschicken von Kommandos an andere Tk Anwendungen:
```
send tgdb "break tkEval.c:200"
winfo interps
```
Returns: `wish tgdb ppres`
- Fensterinformation:
```
winfo width .x
winfo children .
winfo containing $x $y
```

5.9 Zugriff auf andere X Hilfsmittel

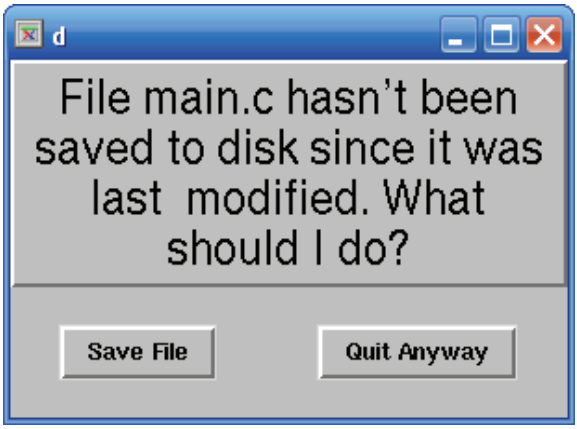
- Tastaturfokus:
```
focus .x.y
```
- Kommunikation mit Fenstermanager:
```
wm title . "Editing main.c"
wm geometry . 300x200
wm iconify .
```
- Entfernen von Fenstern
```
destroy .x
```
- Grabs:
```
grab .x
grab release .x
```

5.10 Beispiel: Dialogbox



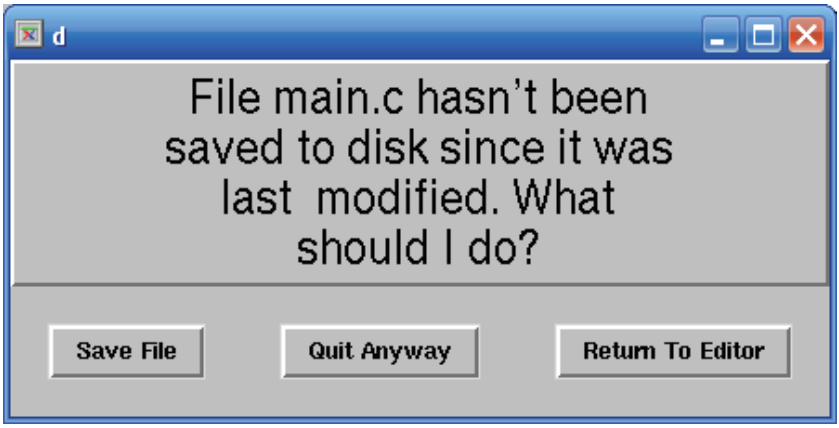

```
toplevel .d

message .d.top -width 3i -bd 2 \
               -relief raised -justify center \
               -font \
                *-helvetica-medium-r-normal--*-240-*\
               -text "File main.c hasn't been \
                saved to disk since it was last \
                 modified. What should I do?"

pack .d.top -side top -fill both
```

Fortsetzung Beispiel:




```
frame .d.bot
pack .d.bot -side bottom -fill both
button .d.bot.left -text "Save File" \
                   -command "quit save"
pack .d.bot.left -side left \
                 -expand yes -padx 20 -pady 20
```

```
button .d.bot.mid -text "Quit Anyway" \
                  -command "quit quit"
pack .d.bot.mid -side left \
                -expand yes -padx 20 -pady 20
```




```
button .d.bot.right -text "Return To Editor" \
                    -command "quit return"
pack .d.bot.right -side left \
                  -expand yes -padx 20 -pady 20

proc quit button {
  puts stdout "You pressed the\
               $button button; bye-bye"
  destroy .d
}
```

5.11 Zusammenfassung

Erzeugen von Schnittstellen mit Tcl-Skripten ist leicht

- Widgets erzeugen
- Anordnung bestimmen mit Geometrie Managern

Anbinden an Anwendung und miteinander

6 Tcl/Tk-Anwendungen erstellen

In den vorherigen Übersichten haben wir versucht, einen ersten Eindruck und eine Kurzanleitung für graphische Fenstersysteme und Tcl/Tk im besonderen zu geben. Harrison und McLennan [12] behaupten, daß es möglich sei, sich die Grundlagen von Tcl/Tk in wenigen Stunden anzueignen.

An diesem Punkt entsteht dann aber oft die Frage: Wie kann ich größere Anwendungen planen? Wie kann man am besten die Bildschirmoberfläche gestalten? Wie kann ich meine Anwendung zusammenstellen und verteilen.

Auf der Grundlage des Buches von Harrison und McLennan werden wir den Gedankengang für die Entwicklung einer kleinen Anwendung vom Anfang bis zum Ende beschreiben. Wir gestalten ein kleines Zeichenprogramm, angefangen mit einer Handskizze. Wir werden einen Prototypen bauen, um das Look&Feel auszuprobieren. Wir fügen einige Bindings und Prozeduren hinzu, um die Implementierung abzuschließen. Wir werden auch zeigen, wie man die Entwicklung so steuern kann, daß die meisten Programmteile in Bibliotheken gehen, die man wiederverwenden und mit denen man Anwendungen leichter erstellen und pflegen kann.

6.1 Der Prozeß der Anwendungserstellung

Wie erstellt man eine Tcl/Tk Anwendung? Gewöhnlich sieht der Prozeß wie folgt aus:

- Denke darüber nach, wie die Anwendung aussehen soll und entwerfe einige Skizzen. Beachte die HCI-Empfehlungen.
- Mache die Tk-Widgets aus, die Komponenten deines Entwurfs sind. Falls es gelegentlich welche gibt, die nicht direkt in Tk verfügbar sind, setze sie aus Tk-Komponenten zusammen. Das Leinwand- (canvas) und das Text-Widget sind in diesem Zusammenhang nützlich. Einiges davon wird sich in der eigenen Bibliothek finden, anderes im Internet.
- Schreibe den Tcl/Tk-Code, um die verschiedenen Widgets zu erzeugen und baue sie zusammen, um einen Gesamteindruck zu erhalten. Einige der Dialogboxen und Menüs sind ggf. nur Platzhalter oder haben wenige Einträge.
  Jetzt ist auch ein guter Zeitpunkt, um andere um eine erste Beurteilung zu bitten!
- Versuche, einige der Komponenten als Bibliotheksroutinen abzuspeichern und baue dir die Bibliotheksinfrastruktur auf, die im Abschnitt 8.2 [12] beschrieben wird und die dir später Arbeit abnimmt.
- Füge Funktionalität hinzu. Wo die richtigen Optionen fehlen, füge neue hinzu und binde die Funktionalität mit dem `bind`-Kommando an das Widget. Verwende möglichst existierende Funktionalität des BS, z. B. für eine Druckausgabe binde das UNIX `lpr`-Kommando dazu.
- Gib der Anwendung den letzten Schliff; suche nach festverdrahteten Einstellungen, die man besser in eine Optionsdatenbank stellt (vgl. 8.1.1 [12]), füge Online-Hilfe hinzu (vgl. 6.7.2 [12]), baue eine Ladeanzeige ein, wenn die Anwendung lange braucht, bis sie geladen ist.
- Testen, testen, testen! Tcl-Code, der nicht wenigstens einmal gelaufen ist, kann reichlich Fehler enthalten.
- Verpacke deine Anwendung gut, so daß ein Anwender, der sie sich holt, sie leicht installieren kann. Der erste Eindruck zählt!

6.2 Eine kleine Anwendung

Wir wollen ein kleines Malprogramm bauen, mit dem man mit der Maus Bilder zeichnen kann.

6.2.1 Anwendungsentwurf

Betrachten wir zunächst die notwendige Fähigkeiten unseres Programms. Da das Beispiel knapp gehalten werden soll, konzentrieren wir uns auf wenige Schlüsseleigenschaften.

- Man zeichnet, indem man die Maus bei gedrücktem Mausknopf bewegt.
- Man kann die Zeichenfläche wieder löschen.
- Man kann die Stiftfarbe wählen.
- Man sieht die *x*- und *y*-Koordinaten der Maus.
- Man kann die Anwendung auch wieder verlassen.

Jetzt überlegen wir uns, wie wir die Eigenschaften steuern.

Dazu lohnt es sich, andere Programme anzuschauen und Stilvorgaben zu beachten. Gängige Konvention ist z. B. eine Menüleiste mit den Einträgen **File**, **Edit**, **View**.

- **File** enthält Unterpunkte für Laden (**Load**), Speichern (**Save**, **Save as ...**) und den Ausstieg (**Quit)**.
- **Edit** enthält gewöhnlich Ausschneiden **(Cut)** und Einfügen (**Paste)**.
- **View** bietet sich für Zoomen, Rasterlinien, Lineale, Auf- und Zuklappen von Werkzeugboxen, etc. an.

Hält man sich an diese Vorgaben, haben es Millionen von Anwendern leichter.

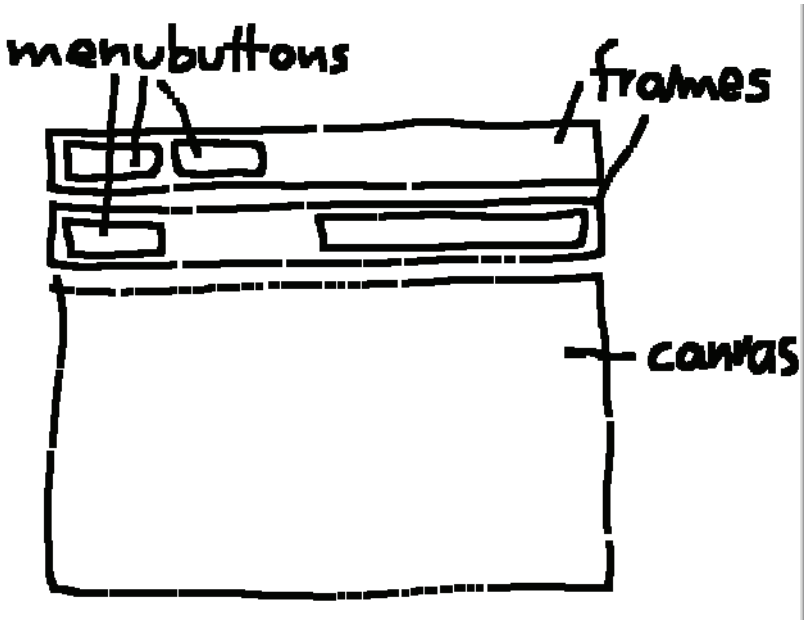
Da man ungern andere Dinge mit in die Menüleiste packt, sehen wir eine zweite Zeile für die Stiftfarbe und die Mauskoordinaten vor. Darunter kommt dann die eigentliche Malfläche.

Für die Farbauswahl kann man sich tausend Sachen ausdenken. Die gewünschte Farbe einzutippen („moosgrünmetallic“), wäre ganz schlecht, ein Farbrad mit eigenem Dialogsystem übertrieben; es wird ein einfaches Farbmenü mit den Grundfarben **black**, **white**, **red**, **green** und **blue** vorgesehen.[1]

6.2.2 Bildschirmentwurf

Zuerst entscheiden wir, welche Tk-Widgets wir brauchen. Die Zeichenfläche ist ein **canvas widget**. Wenn wir zeichnen, fügen wir ein kleines farbiges Rechteck ein. Für die Anzeige der Koordinaten verwenden wir ein **label widget**.

Die Menüknöpfe gruppieren wir in einen Rahmen, der nach ganz oben kommt, die Farbleiste und die Koordinatenanzeige packen wir in einen zweiten Rahmen darunter.



1. Eine Übungsaufgabe lautet, die isländische und finnische Exportversion mit den fremdsprachlichen Menüpunkten zu gestalten!

Das macht die Plazierungsarbeit leichter und portabler. Mit dem **canvas** darunter gehen alle drei in ein Fenster (siehe Abbildung).

6.2.3 Bildschirmprototyp

Zunächst geht es um das allgemeine Aussehen ohne Funktionalität.

Das Hauptfenster wird wie folgt erzeugt:

```
frame .mbar -borderwidth 1 -relief raised
pack .mbar -fill x

menubutton .mbar.file -text "File" -menu .mbar.file.m
pack .mbar.file -side left

menu .mbar.file.m
.mbar.file.m add command -label "Exit"

menubutton .mbar.edit -text "Edit" -menu .mbar.edit.m
pack .mbar.edit -side left

menu .mbar.edit.m
.mbar.edit.m add command -label "Clear"
```

Die Menüpunkte oben haben noch keine dazugebundenen Aktionen, d. h. die `-command` Option fehlt. Für den Farbstift fügen wir ein:

```
frame .style -borderwidth 1 -relief sunken
pack .style -fill x

menubutton .style.color -text "Color" \
  -menu .style.color.m
pack .style.color -side left

menu .style.color.m
.style.color.m add command -label "Black"
.style.color.m add command -label "Blue"
.style.color.m add command -label "Red"
.style.color.m add command -label "Green"
.style.color.m add command -label "Yellow"
```

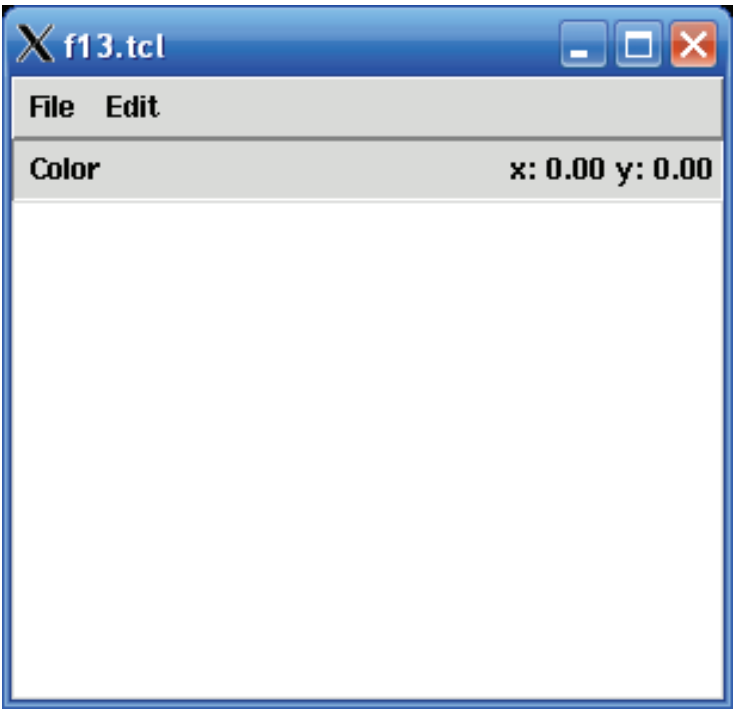
Jetzt erfolgt die Marke für die Stiftkoordinaten:

```
label .style.readout -text "x: 0.00 y: 0.00"
pack .style.readout -side right
```

Der Malbereich sieht wie folgt aus:

```
canvas .sketchpad -background white
pack .sketchpad
```

Die Abbildung unten zeigt den ersten Bildschirmeindruck (Programm `f13.tcl`[1]).



6.2.4 Überlegungen für die Bibliothek

Bevor man weiterstürmt mit der Programmierung, sollte man sich überlegen, welche Kombinationen man wiederverwenden könnte.

Z. B. könnte man auch die Hintergrundfarbe setzen. Dazu könnte man den Code von oben kopieren und modifizieren (Änderungen fett):

```
menubutton .style.bg -text "Background" \
  -menu .style.bg.m
pack .style.bg -side left
```

1. Die Datei finden Sie auf dem Rechner „holle“ im Verzeichnis `/home/students/TCL_share/scriptSamples/kapitel6`.

```
menu .style.bg.m
.style.bg.m add command -label "Black"
  -command {set bg black}
...
.style.bg.m add command -label "Yellow"
  -command {set bg yellow}
```

Stattdessen wird vorgeschlagen, eine Prozedur zum Erzeugen von Farbmenüs zu schreiben:

```
proc cmenu_create {win title cmd} {
  menubutton $win -text $title -menu $win.m

  menu $win.m
  $win.m add command -label "Black" \
     -command "$cmd black"
  $win.m add command -label "Blue" \
     -command "$cmd blue"
  $win.m add command -label "Red" \
     -command "$cmd red"
  $win.m add command -label "Green" \
     -command "$cmd green"
  $win.m add command -label "Yellow" \
     -command "$cmd yellow"
}
```

Wie man sieht, verlangt die Prozedur drei Argumente: einen Widgetbezeichner, einen Titel für den Knopf, ein Kommando, das auszuführen ist, wenn eine Farbe gewählt wurde.

Die zwei Farbmenüs aus unserer Anwendung würden dann mit den zwei Aufrufen

```
cmenu_create .style.color "Color" {set color}
pack .style.color -side left

cmenu_create .style.bg "Background" \
  {.sketchpad configure -bg}
pack .style.bg -side left
```

erzeugt. Die Gefahr, eine Farbe zu vergessen oder zu verwechseln, wird geringer. Änderungen lassen sich zentral an der Prozedur vornehmen, die man in eine Bibliothek stellt.

Dafür sollte sie noch etwas freundlicher aussehen, etwa in der Art wie in Abschnitt 8.2.3 [12] beschrieben:

```
colormenu_create .style.color
pack .style.color -side left
```

Diesen Code bauen wir schon jetzt in unsere Anwendung ein und ersetzen damit den Menücode oben. (Hinweis: Es fehlt aber noch der Knopftext oder ein Bildchen dafür wie in der Endversion von Harrison und McLennan vorgesehen.)

Wenn wir Markierungen auf der Malfläche aufbringen, fragen wir die Farbe mit der folgenden Routine ab:

```
set color [colormenu_get .style.color]
```

6.2.5 Funktionalität einfügen

Zunächst müssen wir den Ausstieg aus der Anwendung für den **Exit**-Menüpunkt vorsehen. Dazu fügen wir `-command exit` an den Eintrag an:

```
.mbar.file.m add command -label "Exit" -command exit
```

Auch das Löschen der Malfläche geht einfach, es gibt ein „lösche alles“ Kommando:

```
.mbar.edit.m add command -label "Clear" -command {
  .sketchpad delete all
}
```

Der schwierigste Teil ist offensichtlich das Malen. Die Malfläche reagiert nicht automatisch auf Ereignisse in ihrem Bereich. Wir müssen Reaktionen dazubinden, zunächst die Erkennung der Stiftkoordinaten bei einer Bewegung der Maus. Hierfür gibt es das `<Motion>`-Ereignis:

```
bind .sketchpad <Motion> {sketch_coords %x %y}
```

Wenn gebunden, wird Tk die Pixel-Koordinaten der Maus bei einer Bewegung abfragen (mit einer festen Taktrate) und an die für X-Events in Tk vordefinierten Felder `%x` und `%y` weitergeben, die dann Argumente für die Prozedur `sketch_coords` darstellen.

Diese sieht wie folgt aus:

```
proc sketch_coords {x y} {
  set size [winfo fpixels .sketchpad 1i]
  set x [expr $x/$size]
  set y [expr $y/$size]
  .style.readout configure \
    -text [format "x: %6.2fi y: %6.2fi" $x $y]
}
```

Die Prozedur rechnet Pixel-Koordinaten in Inch (Zoll) um, indem die gelieferten *x*- und *y*-Koordinaten skaliert werden mit dem Wert „Pixel je Inch“, der für das Fenster `.sketchpad` gilt und mittels `winfo` abfragbar ist.

Hinweis: Andere nützliche X-Events sind z. B. Tastenbelegungen (%K). Das folgende Skript (Ousterhout) liefert eine nette Anzeige der symbolischen Namen aller – auch außergewöhnlicher – Tasten.

```
bind . <Any-KeyPress> {
  puts "Symbolischer Name (Keysym): %K"}
focus .
```

Die Ausgabe der Koordinaten wird formatiert mit zwei Nachkommastellen (`format`-Kommando).

Das eigentliche Malen wird durch zwei Ereignisse gesteuert. Wird zum ersten Mal die Taste gedrückt, muß ein Farbklecks auf die Stelle gesetzt werden, an der der Cursor steht. Ab dann wird bei gedrückter Taste mit jeder Bewegung an den kontinuierlich abgefragten Koordinaten ein weiterer Klecks gemalt.

```
bind .sketchpad <ButtonPress-1> {sketch_box_add %x %y}
bind .sketchpad <B1-Motion> {sketch_box_add %x %y}
```

Gemalt wird ein 6 x 6 Pixelfeld zentriert um die (*x*, *y*)-Koordinaten. Ferner setzen wir voraus, daß wir das für Abschnitt 8.2.3 [12] versprochene Abfragen der Stiftfarbe mittels `colormenu_get` bereits haben.

```
proc sketch_box_add {x y} {
  set x0 [expr $x-3]
  set x1 [expr $x+3]
  set y0 [expr $y-3]
```

```
    set y1 [expr $y+3]
    set color [colormenu_get .style.color]

    .sketchpad create rectangle $x0 $y0 $x1 $y1 \
      -outline "" -fill $color
}
```

Wichtig beim Anzeigen des Rechtecks ist, daß wir den sonst üblichen Rand weglassen (`outline` erhält den Nullstring).

Übung 6–1

In der Platin-Version des Malprogramms kann man die Stiftstärke variieren. Es soll drei Stärken (fein, mittel, breit) geben. Es ist jetzt 15 Uhr, der Vorstand tagt um 16:30 Uhr. Wenn Ihre Demo mit dem neuen Feature bis dahin steht, könnte Ihr nächster Geschäftswagen ein Sechszylinder sein!

6.2.6 Letzter Glattstrich

Jetzt sollte man schauen, wie man die Anwendung mit geringem Aufwand perfekt machen kann.

- Sind einige der Widgets zusammengeklemmt? Dann sollte man mit den Optionen `-padx` und `-pady` etwas Raum schaffen.
- Haben alle Fenster aussagekräftige Titel? Wenn nein, setzen des Titels mit
  `wm title . "Sketch Release 7.4 Gold"`
- Was ist fälschlicherweise festverdrahtet? Labeltexte und Menueinträge sind ok, Farben und Fonts eher nicht, da nicht über Rechnergrenzen hinweg portabel. Deshalb statt
  `canvas .sketchpad -background white`
  lieber die Zeichenfläche setzen mit
  `canvas .sketchpad`
  und eine Ressource in die Optionendatenbank stellen:
  `option add *sketchpad.background white startupFile`
  Damit erhält automatisch jedes Widget mit Namen `sketchpad` die Hintergrundfarbe weiß.

- Tastenkürzel; gerade für Aufgaben wie „Schneiden & Kleben“ benötigt man Tastaturabkürzungen.
- Hilfestellung, z. B. durch Hilfefenster (→ Abschnitt 5.3 [12]) oder mittels kontextsensitiver Hilfetexte (H+McL [12]: balloon help); Text erscheint etwas zeitverzögert, wenn man mit dem Cursor in die Nähe eines Widgets kommt. Im Beispiel lauten die Einbauten dafür:

```
balloonhelp_for .style.color {Pen Color:
   Selects the drawing color for the canvas}
balloonhelp_for .style.readout {Pen Location:
   Shows the location of the pointer on the
   drawing canvas (in inches)}
balloonhelp_for .sketchpad {Drawing Canvas:
   Click and drag with the left mouse button
   to draw in this area}
```

- Im **File** Menü einen **About...** Eintrag mit Verfasser, Datum, Copyright, Servicenummer und Gebühr/Minute für Hilfestellung.

6.2.7 Testen

- Fenster auf- und zuziehen; alles ok?
- Klappt das Highlighting, wenn der Focus wechselt? Stimmen die Tastaturkürzel noch, wenn der Anwender die Caps-Lock oder Num-Lock Taste gedrückt hat?
- Stürzt der Dialog ab, wenn man statt **OK** auf **Quit** oder **Cancel** drückt. Ist das Fenster hinterher blockiert?
- Wenn aus einer nichtexistierenden Datei gelesen wird, macht das Programm den Abflug? Sollte man nicht einige `catch`-Kommandos zum Abfangen von Fehlern einbauen?
- Wie sieht das ganze auf einer anderen Plattform aus? Wenn man sich auf UNIX-Kommandos wie `rm` und `lpr` verlassen hat, was sagt Windows dazu?[1]

6.2.8 Programmversand

Programmieren macht nur richtig Spaß, wenn andere Leute das Programm einsetzen. Das verlangt einiges:

- Anwender hat `wish` in der neusten Form. Dann schicken wir nur das Skript oder machen es per **ftp** verfügbar.
- Anwender hat keine `wish`, kennt sie auch nicht und will nichts damit zu tun haben. Dann müssen wir eine komplette Version mit dazugepackter `wish` verschicken, samt Installationsroutine (→ Abschnitt 8.3 [12]).[2]
- Anwender sind „naive“ Internet-Nutzer. Dann betten wir das Programm in eine Web-Seite ein. Der Browser auf Kundenseite (z. B. Netscape) lädt das Tcl/Tk-Skript (sofern er dafür vorbereitet ist, Stichwort Plug-in, Tcl/Tk-enabled, ...) und bringt es auf der Kundenmaschine zur Ausführung. Funktioniert am besten für kleine Anwendungen.

---

1. Als ob uns das irgendwie interessieren würde :-)
2. T-Online Anwender erhalten umsonst einen kleinen Kalender, der sich bei uns registriert und dabei gleich das Paßwort des Anwenders mitabliefert. Gibt’s nicht?

Zusammenfassung

Wir haben an einer kleinen Anwendung die Vorgehensmethodik für die Gestaltung von Programmen mit graphischen Oberflächen kennengelernt. In den folgenden Abschnitten werden die einzelnen Punkte vertieft und es werden Bibliotheksroutinen geschrieben, die in eigene Anwendungen übernommen werden können.

7 Packen, Rastern, Plazieren von Fenstern

Erzeugt man mit

```
button .b -text "Exit" -command exit
```

ein Widget, sieht man es erst, wenn es z. B. mit

```
pack .b -side bottom
```

unten in ein Fenster gelegt wird.

Alternativ könnte man das Widget auf ein virtuelles Raster mit

```
grid .b -row 0 -column 0
```

legen, oder es an spezielle Bildschirmkoordinaten plazieren:

```
place .b -x 10 -y 25
```

Die drei Kommandos `pack`, `grid` und `place` sind die drei Geometriemanager in Tk. Jeder davon unterstützt ein anderes Plazierungskonzept. Insbesondere `pack` und `grid` verwenden ein recht gutes Modell, das auch die automatische Anpassung bei Veränderungen der Fenstergröße vornimmt.

In diesem Kapitel sollen die drei Manager anhand einiger Anwendungen vorgestellt werden und es wird angedeutet, wie man daraus neue Geometriemanager entwickeln kann. Eine der Anwendungen wird ein Notizbuch sein, in dem geblättert wird.

7.1 Der Gebrauch des pack Kommandos

Die wesentliche Eigenschaft des `pack` Managers ist die Vermeidung absoluter Lageangaben. Dazu muß man eine Vorstellung von der allgemeinen Plazierungsstrategie haben.

7.1.1 Das Höhlenmodell (cavity-based model)

Betrachten wir den einfachen Dialog unten mit einem Fehlerbildchen, einer Textbotschaft, einer Trennlinie und einem „Weiter“-Knopf.

Der erste Entwurf des Programmtexts sieht wie folgt aus (`f21.tcl`[1]):

```
label .icon -bitmap error
label .mesg -text "Print job failed:
Printer is off line
or out of paper"
frame .sep -width 100 -height 2 -borderwidth 1 \
           -relief sunken
button .dismiss -text "Dismiss" -command exit
```

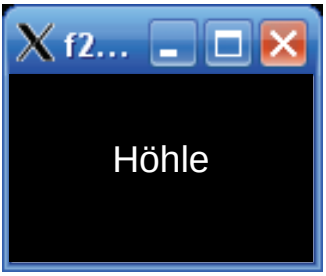
Da es kein Trenner-Widget gibt, verwenden wir einen leeren Rahmen mit Höhe 2 × Randbreite, so daß man nur den Rand sieht. Noch wird dieses Widgets nicht angezeigt, das Layout wird erst durch die folgenden `pack`-Kommandos bestimmt.

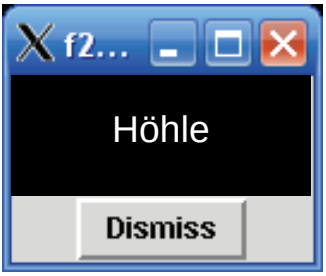
```
pack .dismiss -side bottom
pack .sep -side bottom
pack .icon -side left
pack .mesg -side right
```

1. Diese und alle weiteren Beispieldateien dieses Kapitels finden Sie auf dem Rechner „holle“ im Verzeichnis `/home/students/TCL_share/scriptSamples/kapitel7`.

Jedes `pack`-Kommando legt ein Widget innerhalb seines Elternwidgets ab. Das ganze Vaterfenster wird als leere Höhle betrachtet. Widgets werden entlang den Seiten der Höhle abgelegt: `top`, `bottom`, `left`, `right` – jeweils in der Reihenfolge der `pack`-Kommandos.

Im Falle des Beispiels oben sehen die Schritte wie folgt aus.

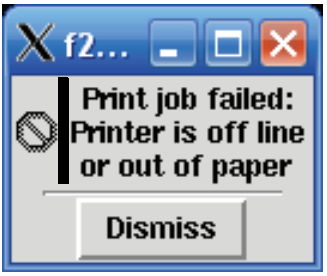





```
pack .dismiss -side bottom
```


```
pack .sep -side bottom
```


```
pack .icon -side left
```




```
pack .mesg -side right
```


```
verkleinern (anpassen)
auf natürliche (verlangte) Größe
```

Der letzte Schritt wird auch „vakuumverpacken“ (shrink-wrapping) genannt. Durch ihn erhält das Gesamtfenster die kleinste benötigte Fläche. Wie man aber sieht, ist das Resultat nicht ganz befriedigend: der Trenner sollte über die ganze Fensterbreite gehen, um den Knopf wünscht man sich etwas Platz und das Icon sollte in der linken oberen Ecke liegen.

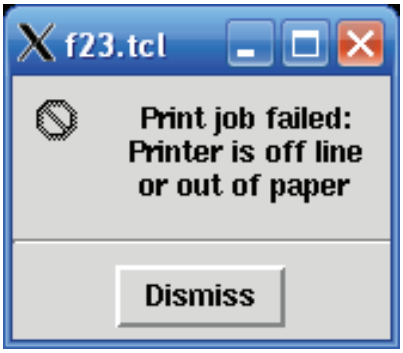
7.1.2 Optionen

Dies erreichen wir mit etwas Füllung (padding), wobei die Angaben sich auf Pixel beziehen (vgl. `f23.tcl`).

```
pack .dismiss -side bottom -pady 4
pack .sep -side bottom -fill x -pady 4
pack .icon -side left -anchor n -padx 8 -pady 8
pack .mesg -side right -padx 8 -pady 8
```

Beim Verankern des Icons ist zu beachten, daß dieses wie alle Objekte normalerweise in die Mitte der vom Packer zugeteilten Parzelle (engl. *parcel*) zu liegen kommt. In unserem Fall legen wir es nach „Norden“ (`anchor n`). Andere Richtungen sind `s`, `e`, `w`, `ne`, `nw`, `se` und `sw`.

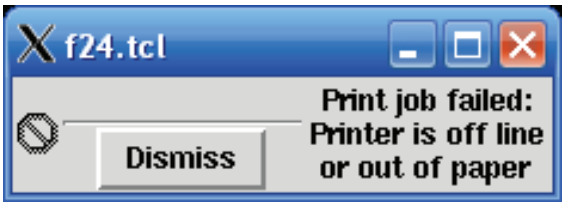
Das Aufziehen der Trennlinie geschieht mit `-fill x` (nur horizontal). Häufig füllt man mit `-fill both`, Vorgabe ist `-fill none`, d. h. Objekte behalten ihre ursprüngliche Größe. Das Ergebnis sieht wie folgt aus:



[12] weisen darauf hin, daß je nach Platzvorgabe und Plazierungsabfolge, die `fill` und `anchor` Optionen keinen sichtbaren Effekt haben. Meist läßt sich das durch genauere Analyse erklären, der erste Eindruck kann aber verwirrend sein.

7.1.3 Plazierungsreihenfolge

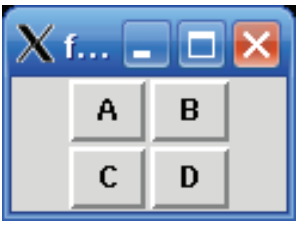
Die Plazierungsreihenfolge ist bei diesem Geometriemanager natürlich entscheidend. Am folgenden Beispiel erkennt man das deutlich (`f24.tcl`).




```
pack .icon -side left
pack .mesg -side right
pack .dismiss -side bottom
pack .sep -side bottom
```

7.1.4 Hierarchisches Packen

Die Neigung des Packers, die ganze Seite einer Höhle zu beanspruchen, kann zu Problemen führen, etwa wenn wir die folgenden vier Knöpfe symmetrisch unterbringen wollen (`f25.tcl`).

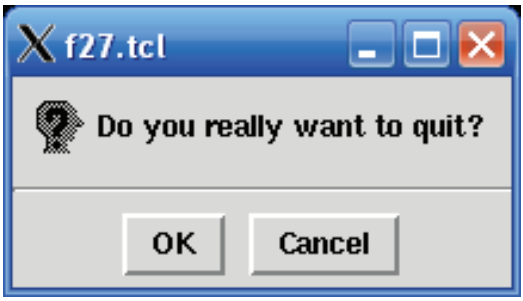


Würde man z. B. Knopf **A** zuerst mit `-side top` packen, würde er die ganze obere Zelle belegen, d. h. **B** käme in die Höhle darunter. Mit dem `grid` Manager ginge dies leichter, aber auch mit dem Packer kommt man ans Ziel, wenn man je zwei Knöpfe in ein Frame legt, wie unten gezeigt.

```
frame .f1
button .f1.a -text "A"
button .f1.b -text "B"
pack .f1.a -side left
pack .f1.b -side left

frame .f2
button .f2.c -text "C"
button .f2.d -text "D"
```

```
pack .f2.c -side left
pack .f2.d -side left

pack .f1 -side top
pack .f2 -side top
```

Die Technik des hierarchischen Aufbaus kann man mit dem Bestätigungsfenster unten nochmals üben.



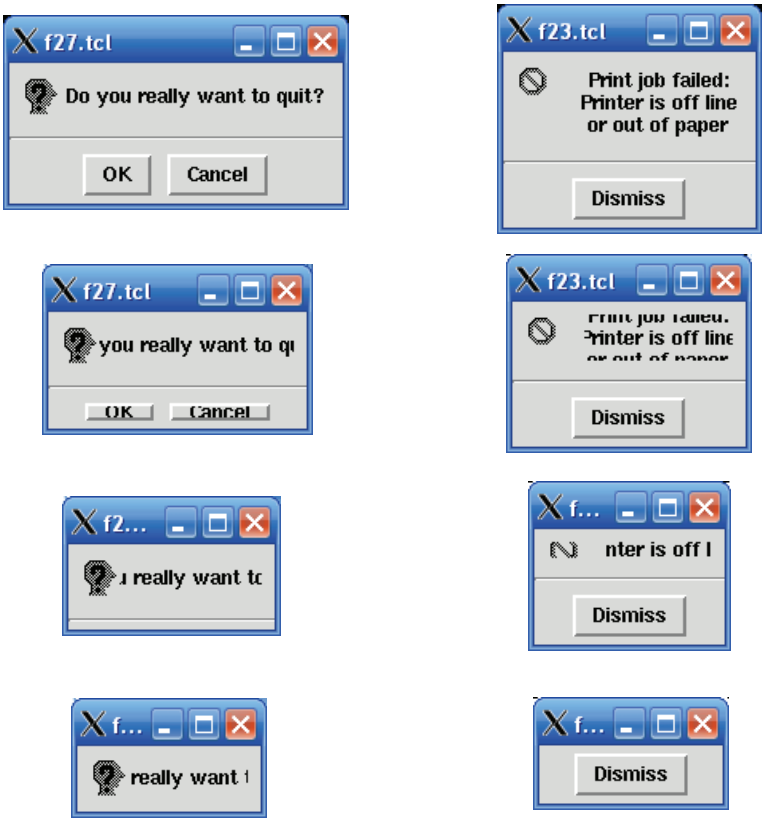
Der Code (`f27.tcl`) hierfür ist recht selbsterklärend.

```
frame .top
label .top.icon -bitmap questhead
label .top.mesg -text "Do you really want to quit?"
pack .top.icon -side left
pack .top.mesg -side right

frame .sep -height 2 -borderwidth 1 -relief sunken

frame .controls
button .controls.ok -text "OK" -command exit
button .controls.cancel -text "Cancel" -command exit
pack .controls.ok -side left -padx 4
pack .controls.cancel -side left -padx 4

pack .top -padx 8 -pady 8
pack .sep -fill x -pady 4
pack .controls -pady 4
```

Ungewohnt ist das Fehlen der `-side` Angaben für die letzten drei `pack` -Kommandos. Die Voreinstellung lautet `-side top` und man läßt sie weg, wenn die Plazierung „normal“ von oben nach unten erfolgt.

7.1.5 Fensterverkleinerungen

Ein umgebendes Fenster hat bei Verwendung des Packers minimale Größe und beim Aufziehen wird entsprechend Füllraum eingefügt. Was passiert beim Stauchen?

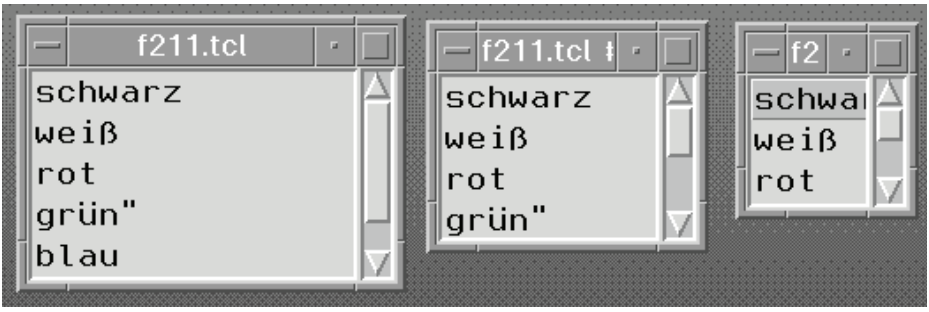


In der linken Spalte wird beim Verkleinern Text vom Label abgeschnitten und die Buttons wurden reduziert. Am Ende sind die Buttons sogar ganz verschwunden.

Macht man den selben Versuch mit dem „Dismiss“-Dialog von oben (siehe rechte Spalte), stellt man fest, daß der Label verschwindet, der Button jedoch bis zuletzt überlebt.

Die Erklärung lautet: die zuletzt gepackten Objekte verschwinden zuerst! Die Faustregel lautet demnach: Packe die wichtigen Dinge zuerst, sofern eine Wahlmöglichkeit besteht.

Bei Rollbalken ist das ganz wichtig, wie man leicht sieht. So ist das Verschwinden des Texts zu verschmerzen, ein Löschen des Rollbalkens wäre fatal. Generell kann man das Verkleinern von Fenstern ganz verhindern oder man kann Minimalgrößen vorschreiben.



7.1.6 Fenstervergrößerungen

Neben Fensterverkleinerungen müssen wir auch Fenstervergrößerungen beachten. Für das Beispiel oben (`f211.tcl`) sieht das Ergebnis des Aufziehens am Rand überraschenderweise sehr unschön aus.

Im rechten Bild oben haben wir auch die Höhle angedeutet, die zwischen den beiden Parzellen existiert. Zweitens bleibt die `listbox`, die ohne `fill`-Optionen plaziert wurde, in der Mitte ihrer Parzelle kleben. Dies ließe sich mit `fill -y` beheben. Das löst nicht (auch nicht mit `-fill both`) das Problem der Lücke rechts, denn `fill` füllt nur innerhalb der Parzelle. Was wir brauchen, ist ein Anwachsen des Widgets parallel mit der Vergrößerung der Parzelle. Dies macht die `-expand`-Option als Teil von `pack`.

Fügen wir nur `-expand yes` ein, wächst die Parzelle beim Aufziehen des Fensters, d. h. die *cavity* verschwindet, das Widget selbst vergrößert sich aber nicht. Erst mit `-exand yes -fill both` sieht das aufgezogene Fenster vernünftig aus.

Man beachte aber, daß nicht jedes Widget wachsen (`-fill` ...) soll. Rollbalken, die breiter werden, sehen ziemlich amateurhaft aus!

7.1.7 Entpacken von Widgets

Ein Widget, das zwar angelegt, aber nicht plaziert („gemanaged“) wurde, existiert sehr wohl, ist jedoch nicht sichtbar. Das kann man dahingehend ausnutzen, daß man Widgets kommen und gehen lassen kann, ohne sie zerstören und neuanlegen zu müssen; man packt und entpackt sie einfach.

Als Beispiel nennen [12] einen Hilfe-Knopf, den man für einige Dialogboxen nicht sehen soll, weil dort keine Hilfe angeboten wird.

Angelegt und angezeigt wird er mit

```
button .controls.help -text "Help"
pack .controls.help -side left -expand yes -padx 4
```

Er verschwindet mit

```
pack forget .controls.help
```

und wird mit erneutem

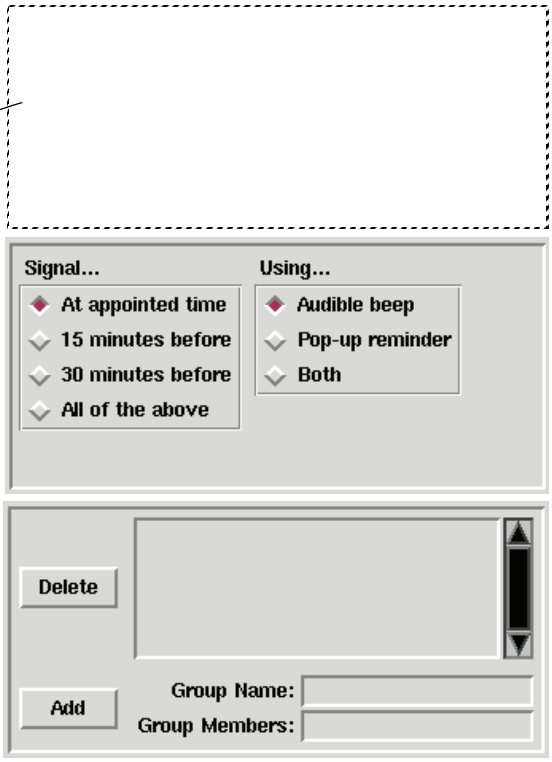
```
pack .controls.help -side left -expand yes -padx 4
```

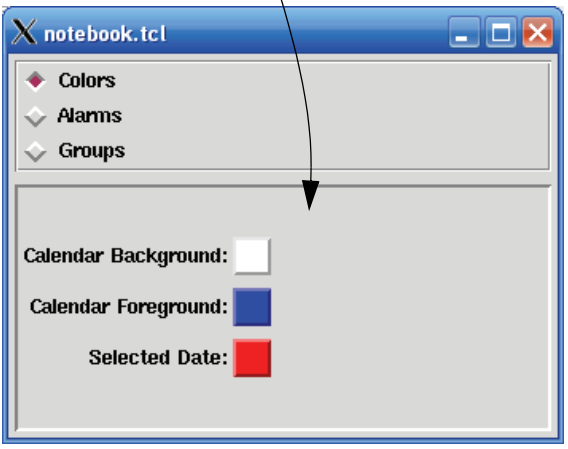
wieder angezeigt. Dabei kommt er ans Ende der Packliste. Möchte man das vermeiden, etwa weil der Knopf in der Mitte liegen soll, geht das mit der Option `-before` oder `-after`:

```
pack .controls.help -before .controls.cancel \
                    -side left -expand yes -padx 4
```

Dies läßt sich im großen Stil für Anwendungen nutzen, bei denen innerhalb eines Widgets große Teile systematisch (modal) ausgetauscht werden, analog zu Seiten eines Büchleins. Das sog. Notizbuch unten (`notebook.tcl`) unten, wird von [12] als Beispiel angeführt. Generell vermeidet man damit die Überladung des Bildschirms mit gestapelten Fenstern oder einem Widget mit einer Ziegelwand voller Knöpfe.

Seiten (Widgets außerhalb des Bildschirms)





Radiobox (Kontrollbereich)

Der Aufbau einer ähnlichen Anwendung gestaltet sich wie folgt (`f218.tcl`). Zuerst schaffen wir das Notebook.

```
notebook_create .nb
pack .nb -side bottom -expand yes -fill both -padx 4 \
         -pady 4
```

Damit ist das Notizbuch grundsätzlich angelegt. Seiten werden mit einer anderen Prozedur hinzugefügt, wobei die Seite zunächst ein leerer *Frame* ist, den man später füllt.

```
set p1 [notebook_page .nb "Page #1"]
```

Ein möglicher Inhalt wäre etwa

```
label $p1.icon -bitmap info
pack $p1.icon -side left -padx 8 -pady 8
label $p1.mesg -text "Ein Text\nauf\nSeite 1"
pack $p1.mesg -side left -expand yes -pady 8
```

Eine zweite Seite sieht vielleicht wie folgt aus:

```
set p2 [notebook_page .nb "Page #2"]
label $p2.mesg -text "Ein anderer Text auf Seite 2"
pack $p2.mesg -side left -expand yes -pady 8 -padx 8
```

Diese Seiten haben symbolische Namen erhalten, unter denen wir sie später aufrufen können, z. B. mit

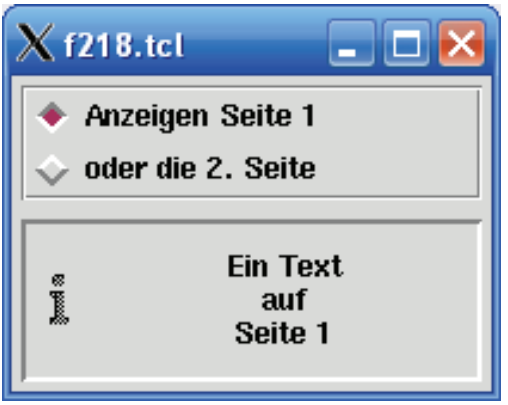
```
notebook_display .nb "Page #1"
```

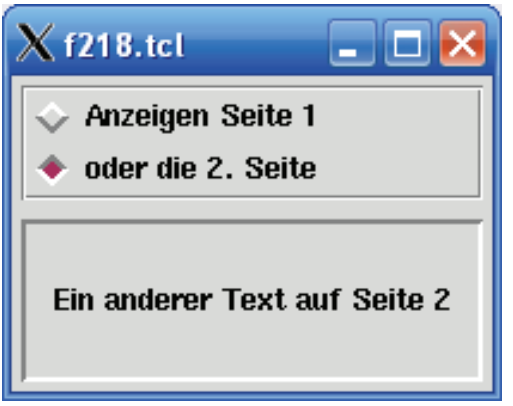
bzw. mit

```
notebook_display .nb "Page #2"
```

Diese beiden Kommandos stecken wir in die Radiobox, deren Entwicklung in [12] im Kapitel 8.2.1 beschrieben wird.

```
radiobox_create .controls
pack .controls -side top -fill x -padx 4 -pady 4
radiobox_add .controls "Anzeigen Seite 1" \
  {notebook_display .nb "Page #1"}
radiobox_add .controls "oder die 2. Seite" \
  {notebook_display .nb "Page #2"}
```





Damit ist das Grobkonzept des Notizbuchs (vgl. `f218.tcl`) klar und wir müssen die einzelnen Prozeduren vervollständigen. Zuerst hatten wir `notebook_create` für den äußeren Rahmen.

```
proc notebook_create {win} {
  global nbInfo
  frame $win -class Notebook
  pack propagate $win 0
  set nbInfo ($win-count) 0
  set nbInfo ($win-pages) ""
  set nbInfo ($win-current) ""
  return $win
}
```

Der Klassenname des Rahmens (`frame`) ist `Notebook`. Für diese Klasse können wir dann Optionen in die Ressoucendatei legen, z. B. diese Anweisungen für einen abgesenkten Rand.

```
option add *Notebook.borderWidth 2 WidgetDefault
option add *Notebook.relief sunken WidgetDefault
```

Das Kommando `pack propagate` hilft uns die Größe zu kontrollieren. Wir übergehen das hier momentan.

Die Information über den Zustand des Notizbuchs legen wir in einen globalen Vektor `nbInfo`, dessen Felder die Seitenanzahl, die Liste der Seiten und den Namen der gegenwärtigen Seite enthalten.

Die Prozedur `notebook_create` liefert den Namen des gerade geschaffenen Notizbuchs zurück.

Jetzt schauen wir uns die Prozedur zum Erzeugen von Seiten an. Sie verlangt den Namen des Notizbuchs und einen Namen für die Seite.

```
proc notebook_page {win name} {
  global nbInfo
  set page "$win.page[incr nbInfo($win-count)]"
  lappend nbInfo($win-pages) $page
  set nbInfo($win-page-$name) $page
  frame $page
  if {$nbInfo($win-count) == 1} {
     after idle [list notebook_display $win $name]
  }
  return page
}
```

Zunächst werden neue, eindeutige Namen durch Hochzählen der Seitennummer in `nbInfo` generiert. In unserem Notizbuch `.nb` entstehen so für die Seitenfenster die Bezeichner `.nb.page1`, `.nb.page2`, usw. Dieser neu generierte Name kommt in die Liste der Namen in `nbInfo` und wird in der Variablen `page` gespeichert. Es wird ein `frame` dafür erzeugt und der Name wird zurückgeliefert.

Ist es die erste Seite, wird sie angezeigt, sobald die Anwendung nichts mehr zu tun hat. Insbesondere sollten dann auch die anderen Seiten generiert worden sein, so daß deren Größe bekannt ist und damit das Notizbuch richtig eingestellt werden kann. Das `list`-Kommando darin bewirkt die richtige Aufbereitung der Argumente und wird in Abschnitt 3.1.5 [12] besprochen.

Jetzt kommt das eigentliche Anzeigen der Seite.[1]

```
proc notebook_display {win name} {
  global nbInfo

  set page ""
  if {[info exists nbInfo($win-page-$name)]} {
     set page $nbInfo($win-page-$name)
  } elseif {[winfo exists $win.page$name]} {
     set page $win.page$name
  }
  if {$page == ""} {
     error "bad notebook page \"$name\""
  }

  notebook_fix_size $win

  if {$nbInfo($win-current) != ""} {
     pack forget $nbInfo($win-current)
  }
  pack $page -expand yes -fill both
  set nbInfo($win-current) $page
}
```

Die Argumente sind der Name des Notizbuchs und der Name der Seite, die angezeigt werden soll. Wenn der Seitenname ein symbolischer Bezeichner ist, finden wir den Seitennamen im Slot `nbInfo($win-page-$name)`, den wir vorher erzeugt haben. Wir benutzen `info exists` um herauszubekommen, ob ein solcher Eintrag existiert. Wir lassen auch einfache Ziffern als Seitennamen zu, also z. B. 0, 1, 2, usw. In diesem Fall finden wir den Namen mit der 2. Alternative (`elseif`).

Danach zeigen wir diese Seite an, wobei zuerst die gegenwärtige verschwinden muß (`pack forget`).

Zuletzt müssen wir uns um die Größen des Notizbuchs kümmern. Die Seiten werden in der Regel unterschiedlich groß sein. Bei jedem Seiten-

1. Die folgenden Prozeduren sind nicht in der Datei `f218.tcl` enthalten. Sie wurden durch Einbinden des Paketes `Efftcl` durch das Kommando `package require Efftcl` verfügbar gemacht. Dies gilt ebenfalls für die Prozeduren zum Umgang mit den Radioboxen.

wechsel würde ohne Vorkehrungen das Notizbuch auf eine neue Größe springen, was wenig elegant wirkt. Man wird daher das Notizbuch so planen, daß die größte Seite hineinpaßt und die kleineren Seiten entsprechend strecken. Das Notizbuch selbst bleibt in der Größe fest, daher oben das Kommando

```
pack propagate $win 0
```

das mit dem Wert 0 die Anpassungsweitergabe ausschaltet.

Die Größenberechnung wollen wir nicht im Detail besprechen. Interessant ist das Kommando `update idletasks`, mit dem wir ausstehende `pack`-Kommandos einleiten, da die Größe nicht berechnet werden kann, sofern nicht alle Fenster zur Anzeige aufbereitet wurden.

```
proc notebook_fix_size {win} {
  global nbInfo
  update idletasks
  set maxw 0
  set maxh 0
  foreach page $nbInfo($win-pages) {
    set w [winfo reqwidth $page]
    if {$w > $maxw} {
      set maxw $w
    }
    set h [winfo reqheight $page]
    if {$h > $maxh} {
      set maxh $h
    }
  }
  set bd [$win cget -borderwidth]
  set maxw [expr $maxw+2*$bd]
  set maxh [expr $maxh+2*$bd]
  $win configure -width $maxw -height $maxh
}
```

7.2 Das `grid`-Kommando

Die zweite Möglichkeit des Geometriemanagements ist die Objekte entlang eines virtuellen Gitters auszulegen. Der `grid`-Manager ist einfacher, aber genauso mächtig wie `pack`.

7.2.1 Das Gittermodell

Ein Gitter kann eine beliebige Anzahl von Zeilen und Spalten besitzen. Diese können unterschiedliche Größen haben. Die Kombination von Zeilen- und Spaltennummer bezeichnet den Platz (die Parzelle), auf dem (auf der) ein Widget zu liegen kommt.

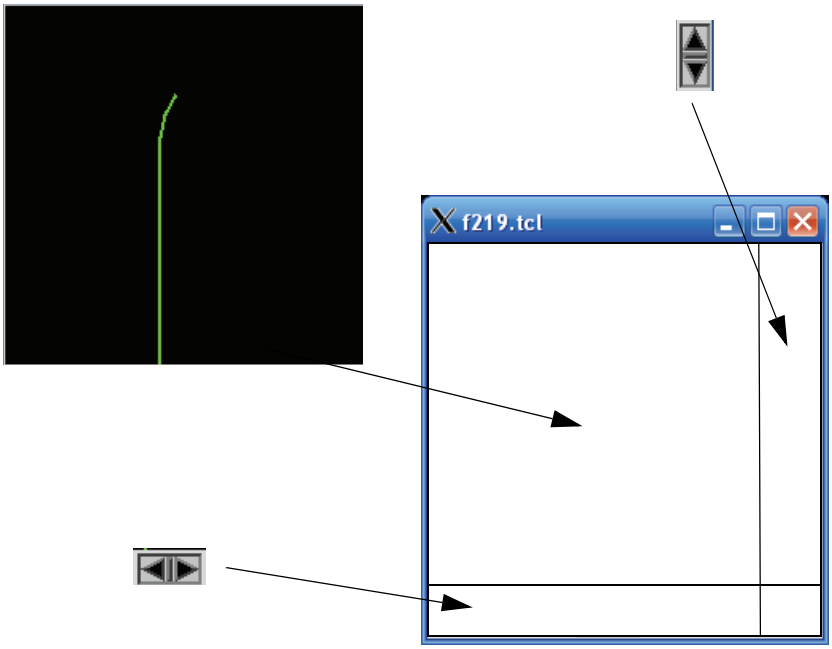
Damit lassen sich Widgets gut zueinander anordnen, z. B. eine Leinwand (canvas) zu den Rollbalken (`f219.tcl`).

```
canvas .display -width 3i -height 3i -background black \
  -xscrollcommand {.xsbar set} \
  -yscrollcommand {.ysbar set}
scrollbar .xsbar -orient horizontal \
  -command {.display xview}
scrollbar .ysbar -orient vertical \
  -command {.display yview}
.display create line 98.0 298.0 98.0 83.0 \
  -fill green -width 2
.display create line 98.0 83.0 101.0 69.0 \
  -fill green -width 2
 .display create line 101.0 69.0 108.0 56.0 \
  -fill green -width 2
...
```

Um jetzt den Rollbalken für die *y*-Richtung rechts von der Leinwand und den Balken für *x* unter die Leinwand zu legen, können wir `grid` verwenden. Werden in einem Aufruf mehrere Widgets aufgeführt, kommen sie alle in verschiedene Spalten der gleichen Fensterreihe.

```
grid .display .ysbar
grid .xsbar x
```

In der zweiten Fensterreihe (zweiter `grid`-Aufruf) steht `x` für eine leere Parzelle, wir hätten es auch weglassen dürfen.



Hätten wir dagegen weitere Widgets in der dritten, vierten Spalte, wären wir mit `x` als Platzhalter weitergesprungen.

Wie beim Packer wird das endgültige Fenster dicht um die erzeugten Widgets gelegt, wobei Zeile und Spalte 1 von der Größe der Leinwand dominiert wird, während die zweite Spalte durch den Rollbalken festgelegt wird.

Das fertige Fenster sieht wieder nicht so toll aus: die Rollbalken sitzen zwar an den richtigen Stellen, sind aber zu kurz (siehe Abbildung unten).

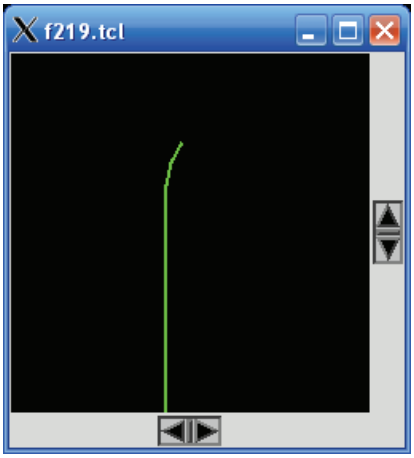
7.2.2 Gitteroptionen

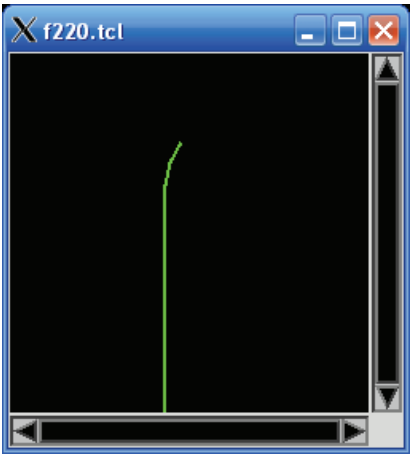
Im obigen Programmausschnitt war die Reihenfolge der `grid`-Kommandos relevant. Das läßt sich vermeiden durch Angabe der Koordinaten, also z. B.

```
grid .display -row 0 -column 0
grid .ysbar -row 0 -column 1
```

was zwar mehr Schreibarbeit bedeutet, aber den Text flexibler macht.

Bei den Rollbalken brauchen wir jetzt etwas in der Art von `-fill` beim Packer. Bei `grid` ist es `-sticky`, wobei man die Seite der Parzelle angeben muß, an der das Widget „kleben“ soll. Gibt man nur eine Seite an, wirkt `-sticky` wie `-anchor` bei `pack`. An zwei Seiten wirkt es wie `-fill x` bzw. `y`. An vier Seiten entspricht es dem `-fill both`.

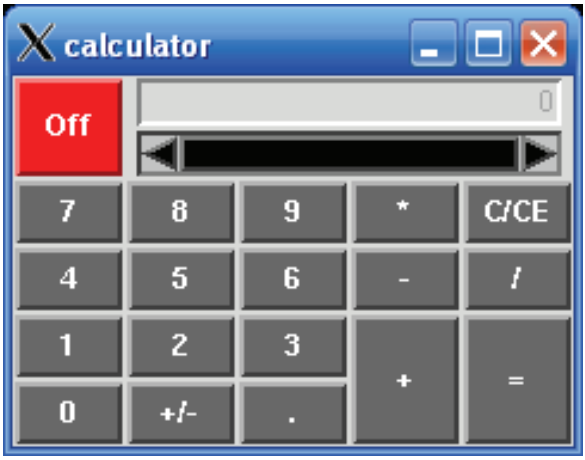





```
grid .display -row 0 -column 0 -sticky nsew
grid .ysbar -row 0 -column 1 -sticky ns
grid .xsbar -row 1 -column 0 -sticky ew
```

In unserem Beispiel (vgl. `f220.tcl`) wird die Leinwand an allen Seiten festgemacht, der *y*-Rollbalken aber nur oben und unten (`-sticky ns`).

Für `grid` gibt es eine Reihe weiterer Optionen, die wir z. T. an einem anderen Beispiel sehen werden. Dies ist ein kleiner Taschenrechner für beliebig große Zahlen, weshalb das Anzeigefeld auch einen horizontalen Rollbalken bekommt.



Die Widgets werden wie folgt erzeugt.

```
button .quit -text "Off" -command exit

entry .readout -state disabled -textvariable current \
   -xscrollcommand {.sbar set}
scrollbar .sbar -orient horizontal \
   -command {.readout xview}

button .key0 -text "0" -width 3 -command {keypress "0"}
button .key1 -text "1" -width 3 -command {keypress "1"}
button .key2 -text "2" -width 3 -command {keypress "2"}
button .key3 -text "3" -width 3 -command {keypress "3"}
button .key4 -text "4" -width 3 -command {keypress "4"}
button .key5 -text "5" -width 3 -command {keypress "5"}
button .key6 -text "6" -width 3 -command {keypress "6"}
button .key7 -text "7" -width 3 -command {keypress "7"}
button .key8 -text "8" -width 3 -command {keypress "8"}
button .key9 -text "9" -width 3 -command {keypress "9"}
button .point -text "." -width 3 -command {keypress "."}
button .plus -text "+" -width 3 -command {keypress "+"}
button .minus -text "-" -width 3 -command {keypress "-"}
button .times -text "*" -width 3 -command {keypress "*"}
button .div -text "/" -width 3 -command {keypress "/"}
button .equal -text "=" -width 3 -command {keypress "="}
button .sign -text "+/-" -width 3 \
                                -command {keypress "+/-"}
```

```
button .clear -text "C/CE" -width 3 \
                                -command {keypress "C/CE"}
```

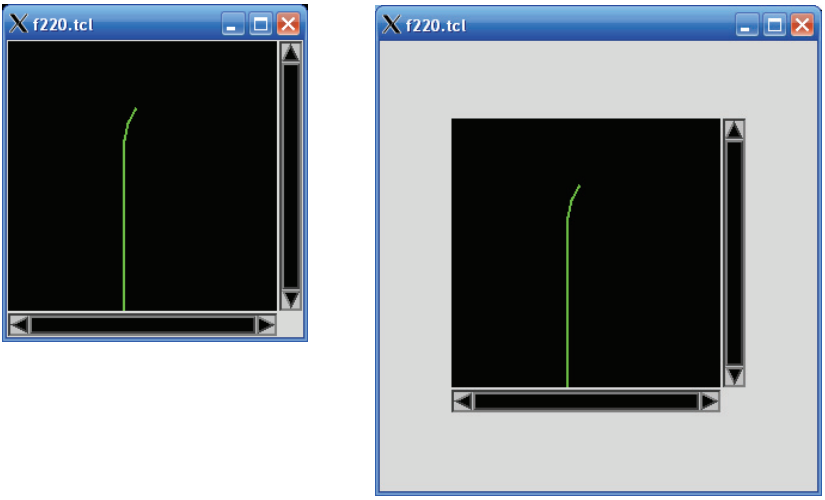
Diese werden dann durch `grid` wie folgt in das Gitter gelegt.

```
grid .quit .readout -sticky nsew
grid .sbar -column 1 -sticky nsew
grid .key7 .key8 .key9 .times .clear -sticky nsew
grid .key4 .key5 .key6 .minus .div -sticky nsew
grid .key1 .key2 .key3 .plus .equal -sticky nsew
grid .key0 .sign .point -sticky nsew

grid configure .quit -rowspan 2
grid configure .sbar -columnspan 4 -padx 4
grid configure .readout -columnspan 4 -padx 4
grid configure .plus -rowspan 2
grid configure .equal -rowspan 2
```

Interessant an diesem Beispiel ist das Strecken einiger Widgets über mehrere Spalten oder Zeilen. Dies betrifft den „Off“-Knopf, die Anzeige, sowie die Tasten für `+` und `-`. Im Beispiel wird dies nachträglich mit `configure` gemacht. Durch `-rowspan` bzw. `-columnspan` werden die angegebene Anzahl von Zellen nach rechts bzw. nach unten mitbelegt. Das Füllen dieser Zellen wird wiederum durch die `-sticky` Option bestimmt, in unserem Fall wird der ganze Raum ausgefüllt (`nsew`).

7.2.3 Größenanpassungen

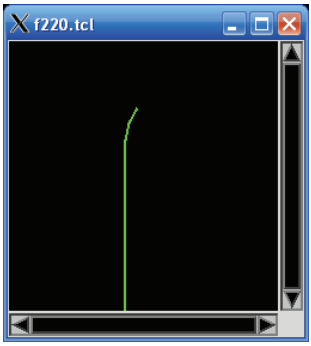
Wie schon beim Packer gibt es auch für `grid` eine generelle Größenanpassungsstrategie. Diese `grid`-Strategie ist aber weniger flexibel als bei `pack`, weil alle Objekte nach Voreinstellung ihre Größe behalten. Beim Zusammenschieben werden deshalb die äußeren Zellen nicht mehr angezeigt. Beim Auseinanderziehen schwimmt dagegen die Matrix in der Mitte des Fensters.

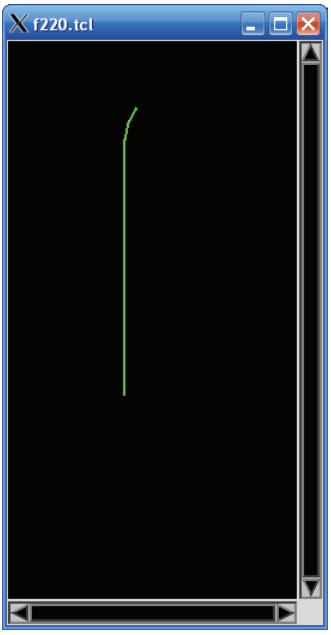


Dem muß man dadurch begegnen, daß man Gewichte für Größenänderungen festlegt. Per Voreinstellung haben alle Zellen Gewicht 0, d. h. sie ändern ihre Größe nicht bei Änderungen ihres Vaterwidgets. Mit einer ganzen Zahl größer 0 legt man fest, welche Zellen an Änderungen partizipieren und zwar im Verhältnis der angegebenen Gewichte. Durch

```
grid columnconfigure . 0 -weight 1
grid rowconfigure . 0 -weight 1
```

legen wir im Fenster namens `.` für die erste Zelle (die Leinwand) fest, daß sie sich mit Gewicht 1 ändert, d. h. sie macht alle Größenanpassungen voll mit. Die anderen Widgets bleiben auf Gewicht 0, d. h. die Rollbalken sollen sich nicht in ihrer Breite ändern.





Damit klappt die Größenveränderung jetzt gut. Auch beim Taschenrechner kann man dies beobachten. Generell gilt, daß gleiche Gewichte bei allen Widgets für gleiches proportionales Wachstum bzw. Verkleinerung, sorgen.

7.2.4 Die Manager `grid` und `pack` zusammen

In [12] folgt ein Beispiel, in dem `pack` und `grid` für eine E-Mail-Anwendung zusammen genutzt werden. Generell gilt, je *Frame* kann nur ein Manager die Kontrolle innehaben. Verschiedene Frames können aber von verschiedenen Managern kontrolliert werden.

Wir übergehen dieses Beispiel hier. Für die Vorlesung sollte man mit beiden und ggf. mit einer Mischung experimentieren.

7.3 Das `place`-Kommando

Mit `place` lassen sich Widgets an absolute oder fensterrelative Koordinaten legen.

7.3.1 Das Koordinatenmodell

Nehmen wir an, wir erzeugen einen „Exit“-Knopf wie folgt.

```
button .exit -text "Exit" -command exit
```

Mit dem Kommando

```
place .exit -x 0 -y 0 -anchor nw
```

legen wir ihn an die Koordinaten (0, 0), wobei durch `-anchor nw` der Knopf an seiner „nordwestlichen“ Ecke verankert wurde (Fall **(a)**).

Im Fall **(b)** legen wir den Ankerpunkt auf die halben relativen *x*- und *y*-Fensterkoordinaten, am Widget selbst liegt der Ankerpunkt in der „südöstlichen“ Ecke. Damit liegt der Knopf insgesamt seitlich versetzt im Fenster.

Hätten wir ihn direkt in der Mitte gewollt, wäre das mit `-anchor c` (centered) möglich gewesen, wie im Fall **(c)** zu sehen, wo gleichzeitig auch die Widgetgrößen relativ zur Fenstergröße angegeben werden. Absolute Größen lassen sich mit `-width` und `-height` angeben.

(a) **`place .exit -x 0 -y 0 -anchor nw`**

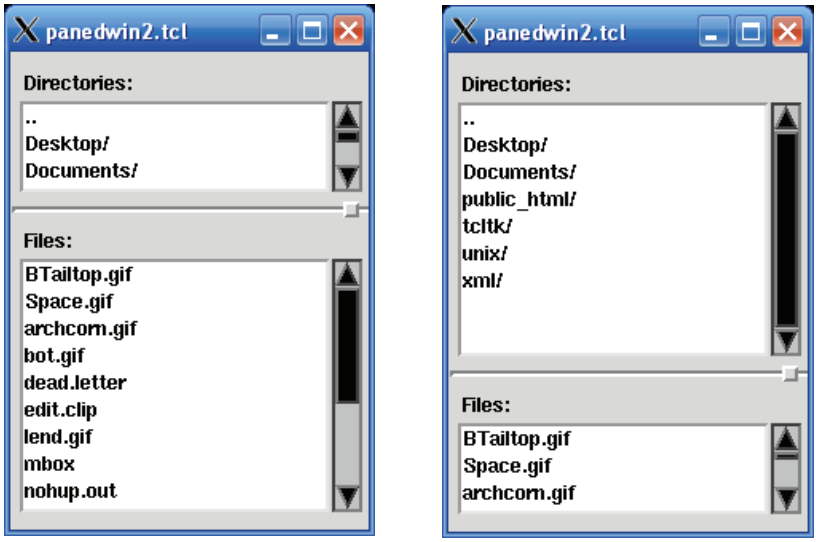
(b)
```
place .exit -relx 0.5 -rely 0.5 \
      -anchor se
```

(c)
```
place .exit -relx 0.5 -rely 0.5 \
    -anchor c -relwidth 0.8 -relheight 0.8
```

Der `place` Manager macht kein „shrink-wrapping“ um die Widgets herum, d. h. Objekte kleben wie auf einer Pinwand und liegen ggf. übereinander, wobei die Aufruffolge die Verdeckung (**stacking**) bestimmt, was sich durch `raise` und `lower` verändern läßt. Insgesamt macht dies den „Placer“ sehr unflexibel, z. B. wenn sich durch eine Schriftart- oder Schriftgrößenveränderung Widgets vergrößern, die sich dann ggf. überlappen.

7.3.2 Eigene Geometriemanager

Mit `place` lassen sich aber eigene Geometriealgorithmen realisieren, indem auf höherer Ebene eine Plazierungsstrategie das Layout berechnet und dieses dann über `place` absolut an das Fenster weitergibt. Dies wird an einem Beispiel demonstriert, dem sog. **paned window**, einer Art **Schiebetafel**, bei dem das Fenster in zwei koordinierte Bereiche aufgeteilt ist.



Die Bereiche in dieser Darstellung heißen panes und haben eine Trennlinie dazwischen, den sog. sash (Schieberahmen) mit einem Griff (grip). Wie man an den Bildern sieht (vgl. `panedwin2.tcl`), kann man den Verzeichnissen oben mehr Platz geben, dann aber zu Lasten der Normaldateien unten – und umgekehrt.

Wir besprechen jetzt einige Prozeduren einer vereinfachten Variante (`panedwin1.tcl`), mit denen wir solche Fenster anlegen können. Der Aufruf lautet

```
panedwindow_create .pw 3i 4i
pack .pw -expand yes -fill both
```

Verlangt wird ein Fenstername, die Größe wird auf 3×4 Zoll gesetzt. Oben kommt eine Tafel `pane1` rein (als `frame` realisiert), unten `pane2`, dazwischen die Schiebeleiste.

Den oberen Rahmen füllen wir z. B. durch

```
frame .pw.pane1.dirs
pack .pw.pane1.dirs -expand yes -fill both \
  -padx 4 -pady 10
label .pw.pane1.dirs.lab -text "Directories:"
pack .pw.pane1.dirs.lab -anchor w
scrollbar .pw.pane1.dirs.sbar \
  -command {.pw.pane1.dirs.list yview}
pack .pw.pane1.dirs.sbar -side right -fill y
listbox .pw.pane1.dirs.list -selectmode single \
  -yscrollcommand {.pw.pane1.dirs.sbar set}
pack .pw.pane1.dirs.list -side left \
  -expand yes -fill both
```

Ganz analog füllt man den unteren Rahmen. Beim ersten Aufruf sind beide gleich groß.

Die Prozedur zur Erzeugung solcher paned windows sieht wie folgt aus.

```
proc panedwindow_create {win width height} {
  global pwInfo

  frame $win -class Panedwindow \
    -width $width -height $height
  frame $win.pane1
  place $win.pane1 -relx 0.5 -rely 0 -anchor n \
    -relwidth 1.0 -relheight 0.5
  frame $win.pane2
  place $win.pane2 -relx 0.5 -rely 1.0 -anchor s \
    -relwidth 1.0 -relheight 0.5

  frame $win.sash -height 4 -borderwidth 2 \
    -relief sunken
  place $win.sash -relx 0.5 -rely 0.5 \
    -relwidth 1.0 -anchor c

  frame $win.grip -width 10 -height 10 \
    -borderwidth 2 -relief raised
  place $win.grip -relx 0.95 -rely 0.5 -anchor c

  bind $win.grip <ButtonPress-1> "panedwindow_grab $win"
  bind $win.grip <B1-Motion> "panedwindow_drag $win %Y"
  bind $win.grip <ButtonRelease-1> \
```

```
      "panedwindow_drop $win %Y"

  return $win
}
```

Hier wird der Fenstername und die gewünschten Maße übergeben. Der erzeugte Rahmen gehört zur Klasse `Panedwindow`, wodurch sich gewisse Optionen setzen lassen, etwa der Doppelpfeil für das Cursoraussehen auf dem Griff:

```
option add *Panedwindow.grip.cursor sb_v_double_arrow \
  WidgetDefault
```

Als nächstes werden die Rahmen `pane1` und `pane2` erzeugt und plaziert, ersterer oben aufgehangen und nach unten wachsend, letzterer unten (`-anchor s`) in der Mitte des Fensters und nach oben wachsend. Beide gehen über die volle Breite (`-relwidth 1.0`), belegen aber nur die halbe Fensterhöhe.

Dazwischen liegt der Schieberahmen mit dem Griff, der 10 × 10 Pixel groß ist und am rechten Rand befestigt ist. Dieser reagiert auf Mausklikken, ~bewegung und ~freigeben. Speziell „drückt“ sich der Knopf ein, wenn er angeklickt wird und zeigt damit, daß er reagiert hat. Hierfür gibt es eine Prozedur:

```
proc panedwindow_grab {win} {
  $win.grip configure -relief sunken
}
```

Für das Ziehen existiert eine weitere Prozedur, die relativ trickreich ist.

```
proc panedwindow_drag {win y} {
  set realY [expr $y-[winfo rooty $win]]
  set Ymax [winfo height $win]
  set frac [expr double($realY)/$Ymax]
  if {$frac < 0.05} {
    set frac 0.05
  }
  if {$frac > 0.95} {
    set frac 0.95
  }
  place $win.sash -rely $frac
  place $win.grip -rely $frac
```

```
  return $frac
}
```

Zunächst muß die vom Ereignis gelieferte absolute Bildschirmkoordinate `%Y` in eine fensterrelative umgerechnet werden, wozu man die Koordinate des linken oberen Fensterecke (`winfo rooty`) abzieht. Den resultierenden Abstand (in Pixeln) dividieren wir durch die Fensterhöhe, wobei wir Typwandlung nach `double` brauchen, weil `expr` mit `/` nur Integerdivision liefert.

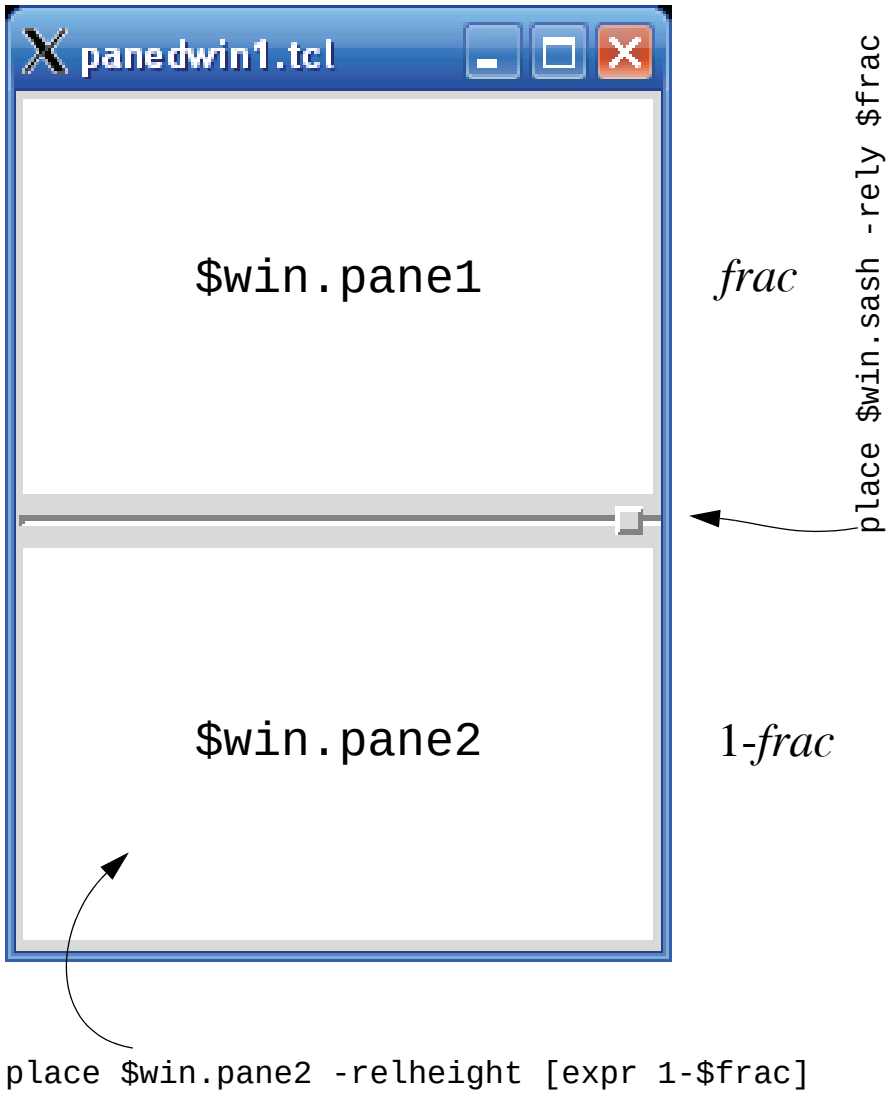
Vorsicht ist auch angebracht, wenn die Schiebeleiste oben über den Rahmen oder unten über den Rahmen hinausgeschoben würde. Der Bruchteil bekäme negativ oder `>1` und die Leiste wäre weg. Deshalb begrenzen wir den Bruchteil auf `>0.05` und `<0.95`.

Zuletzt läßt man den Griff los und jetzt kann die Anpassung der Tafeln erfolgen.

```
proc panedwindow_drop {win y} {
  set frac [panedwindow_drag $win $y]
  panedwindow_divide $win $frac
  $win.grip configure -relief raised
}
```

Mit dem von `panedwindow_drag` zurückgelieferten Bruch teilen wir das Fenster.

```
proc panedwindow_divide {win frac} {
  place $win.sash -rely $frac
  place $win.grip -rely $frac
  place $win.pane1 -relheight $frac
  place $win.pane2 -relheight [expr 1-$frac]
}
```



Zu beachten ist hier, daß die neuen `place` Aufrufe nicht nochmals `-relx`, `-relwidth`, usw. angeben. Die Regel lautet: erneute Aufrufe von `place` überlagern mit ihren neuen Werten bestehende Angaben, lassen aber die anderen unangetastet.

Möchte man jetzt das Fenster anlegen, z. B. mit Schiebefenstern im Verhältnis 1/3 zu 2/3, ginge dies mit

```
panedwindow_create .pw 3i 4i
panedwindow_divide .pw 0.3
pack .pw -expand yes -fill both
```

Damit hätten wir mit einigen neuen Prozeduren eine Art von neuem Manager geschaffen, der Schiebefenster behandeln kann.

8 Ereignisbehandlung

Wenn man einen Knopf drückt, wird der Code ausgeführt, der sich hinter der `-command` Option versteckt. Das sieht einfach aus, wirft aber subtile Fragen auf:

- Wie kann man verhindern, daß das Programm während der Ausführung des Knopfkommandos anhält?
- Wie kann man den Benutzer davon abhalten, mit dem Programm zu kommunizieren, während dieses beschäftigt ist?
- Wie kann ein Druck auf „RETURN“ behandelt werden, während man in einem Eingabefeld ist, obwohl RETURN sonst gar nichts macht?

Hierzu müssen wir uns Tk näher ansehen, spez. auch das `bind`-Kommando. Mit dem `bindtags`-Kommando können wir Gruppen von ähnlichen Kommandos bündeln und mit `after` lassen wir Dinge zeitabhängig ablaufen. Zuerst aber sollten wir uns ansehen, wie das `wish`-Programm funktioniert.

8.1 Die Ereignisschleife

Betrachten wir das „Hallo Welt!“ Skript.

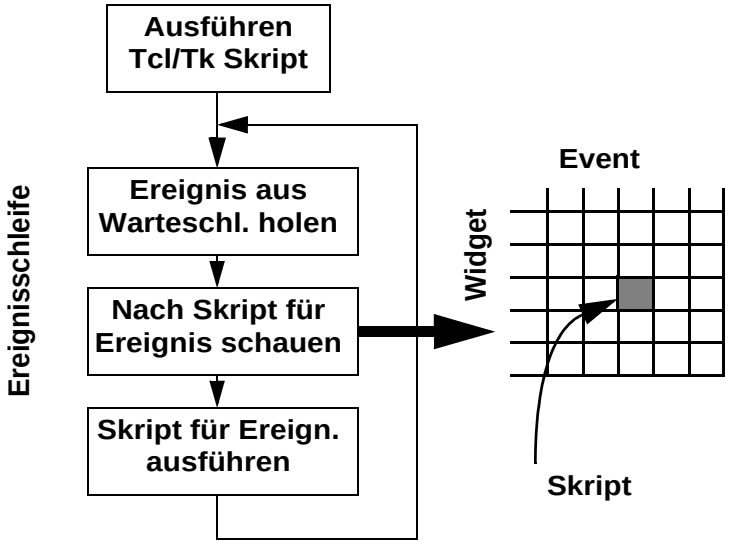
```
button .b -text "Hallo Welt!" -command exit
pack .b
```

Wenn die `wish` das Skript ausgeführt hat, wird das Hauptfenster erzeugt und die Shell wartet darauf, daß etwas passiert. Klickt ein Anwender auf

den Knopf, erhält der Knopf eine Benachrichtigung, die wir *Ereignis* (**event**) nennen, in diesem Fall das `ButtonPress`-Ereignis.

Der Knopf reagiert mit einem abgesenkten Relief. Läßt der Anwender jetzt die Maustaste wieder los, erhält der Knopf ein weiteres Ereignis, diesmal `ButtonRelease`. Der Knopf reagiert darauf mit der Ausführung seines Kommandos und wechselt das Aussehen zurück zum erhobenen Relief.

Die angesprochenen, hereinkommenden Ereignisse werden in einer *Ereigniswarteschlange* (**event queue**) zwischengespeichert und nacheinander durch eine Endlosschleife, die sog. *Ereignisschleife* (**event loop**), abgearbeitet.



Das zugebundene Skript, das für ein Widget ein Ereignis behandelt, heißt **binding**. Die Tk-Widgets haben automatisch zugebundene Skripten, z. B. für den Knopf das Anklicken und das Knopffreigeben, für Eingabefelder Tastendruck, usw.

Möchte man das Standardverhalten ändern oder zusätzliches Verhalten hinzufügen, kann man das mit dem `bind`-Kommando erreichen. Die Syntax ist recht kompliziert (siehe Abschnitt 8.3). Ein einfaches Beispiel für die Zubindung wäre etwa das Skript für unseren „Hallo Welt!“ Knopf:

```
entry .field
pack .field
bind .field <KeyPress-Return> {
  .b flash
  .b invoke
}
```

Offensichtlich wird ein Skript an die Eingabetaste gebunden, damit der Knopf blinkt, wenn man im Eingabefeld die ENTER-Taste drückt. Anschließend wird sein Kommando (also `exit`) aufgerufen.

Neben Widget-Ereignissen kann es auch andere Ereignisse geben, etwa ein Wecksignal durch das `after`-Kommando, das die Ausführung eines Skripts nach einer angegebenen Zeitspanne auslöst. Genauso kann ein zum Schreiben oder Lesen eröffneter Dateideskriptor ein Ereignis auslösen, wodurch Kommunikation mit anderen Programmen möglich wird.

8.1.1 Tastaturfokus

Wenn man auf ein Eingabefeld klickt, erhält man einen aktiven Mauszeiger. In UNIX-Systemen wird zugleich das Eingabefeld schwarz umrandet. Dadurch wird angezeigt, daß das Feld *fokussiert* ist, d. h. die Tastatureingaben gehen an die Anwendung, der dieses Feld gehört.

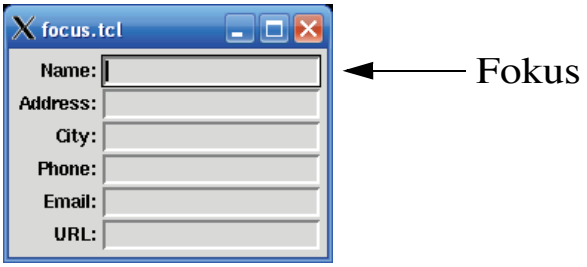
In jedem Hauptfenster kann es zu jedem Zeitpunkt nur ein Widget geben, das den Fokus besitzt. Zwar kann man in einem Dialogfenster mehrere Eingabewidgets haben, die alle nach KeyPress-Ereignissen ausschauen, aber nur eines davon hat den Fokus und erhält die Ereignisse. Im nächsten Beispiel sieht man das für ein Formular.

```
set x 1
foreach t {Name Address City Phone Email URL} {
  label .lab$x -text "$t:"
  entry .ent$x -background gray -width 25
  grid .lab$x -row $x -column 0 -sticky e
  grid .ent$x -row $x -column 1 -sticky ew
  incr x
}
```

Damit der Anwender beim Öffnen gleich ins erste Feld tippen kann, setzen wir den Fokus darauf.

```
focus .ent1
```

Die Ausgabe sieht wie folgt aus (`focus.tcl`):



Später kann der Anwender ein anderes Feld anklicken oder durch die Tabulatortaste den Fokus weiterschalten. Dies wird durch Tks eingebaute Ereignisbindungen erledigt.

8.1.2 Änderungen erzwingen

Wenn `wish` alle Ereignisse der Warteschlange erledigt hat und nichts besseres vorhat, frischt es (**updates**) die Widgets am Bildschirm auf. Dadurch kann Tk mehrere aufgelaufene Änderungen auf einmal erledigen.

Im Beispiel

```
label .x
pack .x
.x configure -text "Hallo Leute!"
.x configure -borderwidth 2 -relief groove
```

sieht man nicht nacheinander die Auswirkungen der Befehle, also zuerst den Knopf, dann den Text und danach die Randlinie. Vielmehr erscheint alles zusammen.

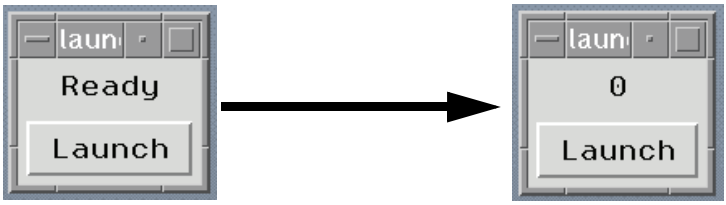
Manchmal möchte man aber Änderungsschritte nacheinander und sofort sehen. Etwa in dem Skript (`launch1.tcl`)

```
label .countdown -text "Ready"
pack .countdown -padx 4 -pady 4

button .launch -text "Launch" -command {
  for {set i 10} {$i >= 0} {incr i -1} {
```

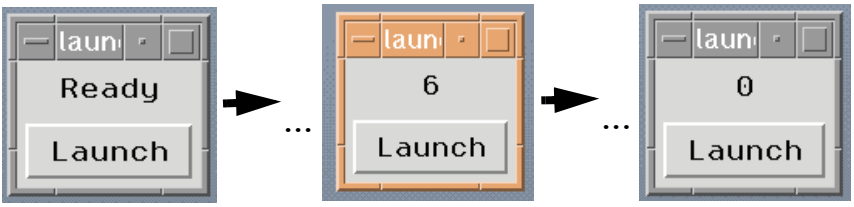
```
      .countdown configure -text $i
      after 1000
    }
  }
  pack .launch -padx 4 -pady 4
```

Mit dem Launch-Knopf kann die Schleife „abfeuern“, die von 10 auf 0 herunterzählt und dabei immer 1000 ms verzögert.

Wer hofft, daß man das Herunterzählen sieht, wird aber enttäuscht. Das Programm pausiert 11 Sekunden und zeigt dann die Null.



(a) Nur letzte Änderung ist sichtbar



(b) **`update`** nach jedem **`configure`** macht Änderungen sichtbar

Aus Sicht von Tk hängen die 11 `configure`-Befehle aneinander und erst nach der `for`-Schleife kehrt das Programm zur Ereigniswarteschlange zurück und erneuert – wenn dort nichts weiter vorliegt – die Anzeige .

Setzen wir einen `update`-Befehl nach jedem `configure` ein, kehrt das Programm mit jedem `update` zur Warteschlange zurück, arbeitet diese ggf. ab und frischt dann die Widgets auf (`launch2.tcl`).

```
button .launch -text "Launch" -command {
  for {set i 10} {$i >= 0} {incr i -1} {
    .countdown configure -text $i
    update
    after 1000
```

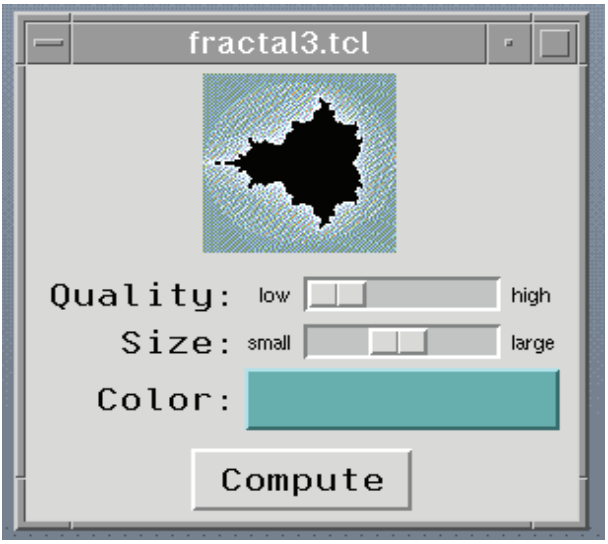
```
    }
  }
```

Da wir eigentlich nur das Nachfahren der Änderungen brauchen, geht es effizienter mit `update idletasks` statt `update` oben. Durch das Schlüsselwort `idletasks` ignoriert `update` die Ereigniswarteschlange und aktualisiert nur die Widgets (`launch3.tcl`).

8.1.3 Langlaufende Anwendungen

Tk verbringt die meiste Zeit in der Warteschleife. Kommt ein Ereignis, wird das zugebundene Kommando ausgeführt. Da diese meist kurz sind, kehrt Tk schnell zurück, erneuert die Ausgabe und somit erscheint das Programm „lebendig“.

Was passiert aber, wenn das angebundene Kommando lange läuft, wie z. B. die Berechnungen für das fraktale Bild im Programm `fractal3.tcl`?



Ohne entsprechende `update`-Vorkehrungen wird das Bild nicht richtig neu angezeigt, wenn das Fenster abgedeckt wurde, weil es nicht auf Ereignisse reagiert. Viele Anwender würden es daher „abschießen“, weil sie denken, die Anwendung sei „abgestürzt“.

Mit `update`, wie hier gekürzt gezeigt

```
$fractal(image) blank
for {set x 0} {$x < $size} {incr x} {
  set cr [expr double($x*$dr)/$size+$fractal(x0)]
  for {set y 0} {$y < $size} {incr y} {
    set ci [expr double($y*$di)/$size+$fractal(y0)]
    set zr 0.0; set zi 0.0
    for {set iter 0} {$iter < $maxiters} {incr iter} {
      set rsq [expr $zr*$zr]
      set isq [expr $zi*$zi]
      set zi [expr 2*$zr*$zi + $ci]
      set zr [expr $rsq - $isq + $cr]
      if {$rsq + $isq >= 4.0} {
        break
      }
    }
    $fractal(image) put $cmap($iter) -to $x $y
  }
  update
  }
}
```

reagiert das Programm auf Ereignisse. Die obige Routine berechnet für jedes Pixel die Farbe mittels der fraktalen Formel und setzt das Pixel mit `put`. Nach jeder Spalte wird `update` aufgerufen und diese angezeigt. Damit wird der Fortschritt angezeigt und der Aufwand hält sich trotzdem in Grenzen.

Generell sollten langlaufende Programme immer einen Fortschrittsanzeiger enthalten, wir kommen darauf in 9.3 zurück.

Es stellt sich die Frage, warum wir nicht wieder `update idletasks` verwenden. Wie oben angeführt, bewirkt `update idletasks` nur die Ausführung der allerletzten Änderungen, d. h. die Anzeige der Pixelspalte. Sonstige nötige Änderungen, etwa in Folge eines `Expose`-Ereignisses nach einer Abdeckung des Fensters, kämen nicht zur Ausführung. Deshalb sollten langlaufende Kommandobindungen immer mit dem vollen `update` arbeiten.

Allerdings hat `update` einen anderen Pferdefuß. Es verarbeitet **alle** Ereignisse, also auch Tastendrücke der Maus und Tastatureingaben. So

kann ein Anwender auf halber Strecke die Größenangaben ändern und „Compute“ drücken. Dann würde zunächst sofort mit dem Generieren des neuen Bildes begonnen, danach aber mit dem alten fortgefahren!

Daher sollte man bei längeren Aktivitäten manche der Anwenderinteraktionen ausschalten. Dies wäre möglich, indem wir den Knopf wie folgt modifizieren (`fractal2.tcl`):

```
button .compute -text "Compute" -command {
  set size [.controls.size get]
  .display configure -width $size -height $size
  set color [.controls.color cget -background]
  set maxiters [.controls.qual get]

  .compute configure -state disabled
  fractal_draw $color $maxiters $size
  .compute configure -state normal
}
```

Große Anwendungen haben aber viele Widgets, die während gewisser Phasen nicht auf Benutzereingaben reagieren sollten. Daher wäre es angebracht, die ganze Anwendung in einen „In-Betrieb-Zustand“ zu versetzen. Dabei ändert der Cursor sein Aussehen, z. B. in eine Sanduhr oder eine Uhr mit Zeigern. Alle Fenster reagieren weiterhin auf Ziehen des Rands und Ver-/Aufdecken, aber der Benutzer kann nicht mit ihnen kommunizieren.

Dazu legen wir einen neuen Rahmen `.busylock` an, den wir außerhalb des Bildschirms plazieren. Mit dem `grab`-Kommando schicken wir alle Maus- und mit `focus` alle Tastaturereignisse an diesen Rahmen, der sie einfach ignoriert.

Der „Compute“-Knopf erhält dann eine Prozedur `busy_eval`, die den Zustand „busy“ herstellt und das Fraktale-Skript ausführt (`fractal3.tcl`).

```
button .compute -text "Compute" -command {
  set size [.controls.size get]
  .display configure -width $size -height $size
  set color [.controls.color cget -background]
  set maxiters [.controls.qual get]
```

```
    busy_eval {
      fractal_draw $color $maxiters $size
    }
  }
```

Die Prozedur `busy_eval` wird wie folgt implementiert:

```
frame .busylock
bind .busylock <KeyPress> break
place .busylock -x -2 -y -2

proc busy_eval {script} {
  set fwin [focus]
  focus .busylock
  grab set .busylock

  set cursor [. cget -cursor]
  . configure -cursor watch
  update

  set status [catch {uplevel $script} result]

  . configure -cursor $cursor
  grab release .busylock
  focus $fwin

  return -code $status $result
}
```

Außerhalb der Prozedur legen wir den Rahmen `.busylock` an und schalten mit `bind ... break` die Standardbindings, wie z. B. „Alt“ oder „Tab“, aus. Die Details dieser Technik werden in 3.5.2 bei Harrison/McLennan [12] gezeigt. Damit `grab` auf allen Platformen richtig funktioniert und keinen „grab window not visible-Fehler“ anzeigt, muß der Rahmen explizit angelegt sein. Wir erreichen das mit `place` und Koordinaten links überhalb der oberen linken Bildschirmecke.

Innerhalb der Prozedur merken wir uns zunächst in der Variablen `fwin`, welche Anwendung den Fokus hatte. Dann ziehen wir Tastatur- und Mauseingabe auf `.busylock` und ändern den Mauszeiger, wobei wir das alte Aussehen wieder retten. Mit `update` werden die Änderungen angezeigt.

Die Änderung des Mauszeigers im Hauptfenster gilt im übrigen für alle abhängigen Unterfenster, mit Ausnahme gewisser Eingabe- und Textcursor. Es lohnt aber nicht, die ganze Widgethierarchie durchzuturnen, um überall `-cursor` zu ändern, nur damit auch dort die Sanduhr erscheint.

Jetzt können wir das als Argument übergebene Skript aufrufen. Statt mit `eval` tun wir dies mit dem `uplevel`-Kommando. Dadurch läuft das Skript in der Aufrufsumgebung ab, nicht im Kontext des Prozedurinneren! Dies bedeutet, daß alle Argumente des Aufrufs ihre Gültigkeit aus der Aufrufsumgebung beziehen. In unserem Fall

```
busy_eval {
  fractal_draw $color $maxiters $size
}
```

waren dies z. B. die Argumente `$color`, `$maxiters`, `$size`, die innerhalb von `busy_eval` überhaupt nicht bekannt sind. Deshalb soll das Skript eine Etage höher im Aufrufstack ablaufen.

Um etwaige Fehler in diesem Skript abzufangen, die sonst die ganze Anwendung im „busy-Zustand“ gelassen hätten, umgeben wir den Aufruf mit dem `catch`-Kommando, merken uns in `status` den Ergebniscode und in `result` das Skriptergebnis. Beide Werte geben wir mittels der Option `-code` im `return`-Kommando zurück an den Aufrufer von `busy_eval`.

8.1.4 Gültigkeitsbereich der Ausführung

Programmtext, der innerhalb der Ereignisschleife abgearbeitet wird, arbeitet im **globalen** Kontext. Speziell können sich Programme, die als Bindings an Ereignisse gebunden werden, nur auf globale Variablen beziehen. Das kann in Programmen wie dem folgenden zu Verwirrungen führen.

```
proc wait_for_click {win} { ;# BUG ALERT
  set x 0
  bind $win <ButtonPress-1> {set x 1}
  bind $win <ButtonPress-2> {set x 2}
  vwait x
```

```
  return $x
}
```

Zunächst sieht es so aus, als ob wir eine Variable `x` auf 0 setzen, danach Bindings zu den Maustasten herstellen und auf die Änderung warten, um das Ergebnis des Maustastendrucks abzuliefern.

Tatsächlich ist das erste `x` aber lokal, während das Kommando `set x 1` in `bind` ein bisher ungenanntes, globales `x` nennt, worauf auch mit `vwait` gewartet wird.

Trotzdem läuft das Programm und reagiert auf Tastaturklicks, wird aber immer 0 zurückliefern, da das lokale `x` nicht verändert wird.

Ähnliche Gültigkeitsregeln gelten für Skripte, die aus der Ereignisschleife heraus aufgerufen werden, wie z. B. solche, die `after` oder `fileevent` übergeben werden, oder solche, die nach der `-command` Option eines Widgets stehen.

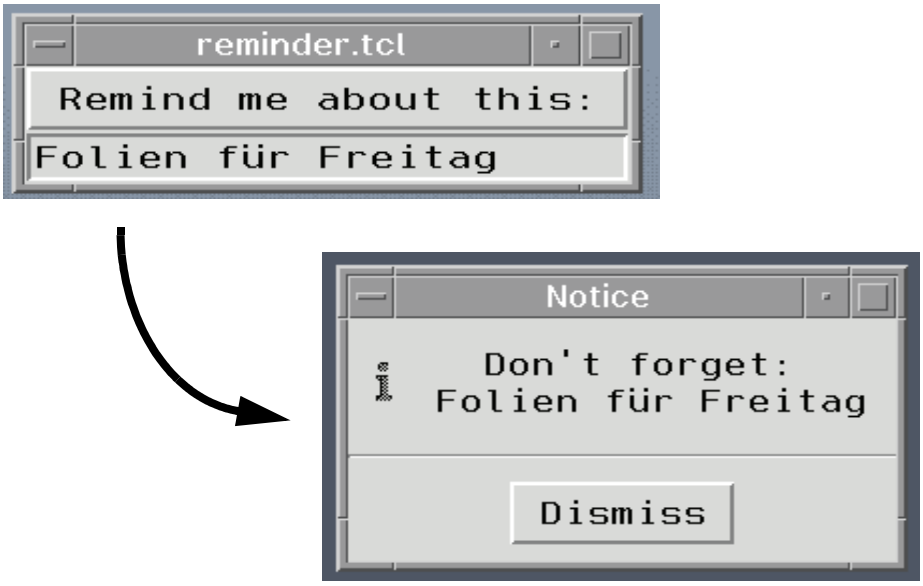
8.1.5 Ersetzungsregeln und die Ereignisschleife

Manchmal braucht man Variablenersetzungen in Skripten, die man den `bind`- oder `after`-Kommandos übergibt. Dabei ist Vorsicht geboten, was die Ersetzungsregeln anbetrifft!

Im folgenden Erinnerungsprogramm, das nach einer festverdrahteten Verzögerung von fünf Sekunden uns ein Memo schickt, wird eine Prozedur `reminder` aufgerufen, die einen vorher eingegebenen Text (in `mesg`) als Memo anzeigen soll.

```
button .remind -text "Remind me about this:" -command {
  reminder "Don't forget: \n[.mesg get]"
  .mesg delete 0 end
}
pack .remind -anchor w
entry .mesg
pack .mesg -fill x
```

Man könnte versucht sein, die `reminder`-Prozedur wie folgt zu implementieren, wobei ein im Abschnitt 6.4 [12] eingeführtes Benachrichtigungswidget `notice_show` benutzt wird.




```
proc reminder {mesg} {
  after 5000 {notice_show $mesg}
}
```

Leider liefert dies nur eine Fehlermeldung, wonach `$mesg` nicht bekannt sei, denn die geschwungenen Klammern haben die Ersetzung von `$mesg` verhindert. Somit erhält `notice_show` nicht die Botschaft, die in dieser Variablen gespeichert ist.

Andere Versuche, dies ohne Klammern oder mit `"..."` zu erreichen, schlagen auch fehl. Gehen würde es mit

```
proc reminder {mesg} {
  after 5000 "notice_show \"$mesg\""
}
```

solange die Botschaft nicht auch wieder doppelte Anführungszeichen enthält.

Die beste Lösung nach [12] ist

```
proc reminder {mesg} {
  after 5000 [list notice_show $mesg]
}
```

wobei das `list`-Kommando seine Argumente richtig formatiert, also mit Escape-Zeichen und Klammern, wie gebraucht.

8.2 Einfache Beispiele mit `bind`

Mit dem `bind`-Kommando kann man das Verhalten eines einzelnen Widgets oder ganzer Widget-Klassen ändern. Einfache Beispiele wären:

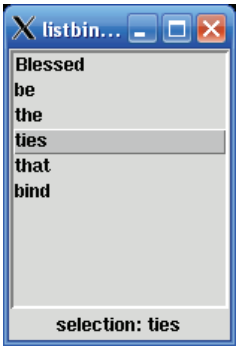
8.2.1 Auswahl eines Eintrags aus einer Liste

In vielen Programmpaketen kann man durch Doppelklicken einen Eintrag aus einer Liste auswählen. In der Listbox-Beschreibung findet sich diese Eigenschaft aber nicht.

Daher bauen wir hier eine Liste mit einigen Einträgen auf und fügen das gewünschte Verhalten mit `bind` hinzu.

```
listbox .lb
label .choice
pack .lb .choice
foreach i {Blessed be the ties that bind} {
  .lb insert end $i
}
bind .lb <Double-Button-1> {
  .choice configure -text "selection: [.lb get active]"
}
```

Wenn wir mit Maustaste 1 doppelklicken, wird im Feld unterhalb der Liste das aktive Element der Liste angezeigt (`listbind.tcl`).



8.2.2 Automatische Hilfetexte

In manchen Anwendungen erscheinen Hilfetexte, wenn man die Maus auf Knöpfe positioniert. In Tk kann man dieses Verhalten leicht einbauen mittels der `Enter` und `Leave` Ereignisse, die durch das Betreten und Verlassen des Mauszeigers für einen Bildschirmbereich ausgelöst werden.

Hier ein Beispiel (`enter.tcl`):

```
frame .cmd
button .cmd.back -bitmap \
  [file join @$env(EFFTCL_LIBRARY) images back.xbm]
button .cmd.fwd -bitmap \
  [file join @$env(EFFTCL_LIBRARY) images fwd.xbm]
canvas .c
label .help -relief sunken -borderwidth 2 -anchor w

pack .cmd .c .help -side top -fill x
pack .cmd.back .cmd.fwd -side left

bind .cmd.back <Enter> {
  .help configure -text "Page backward"}
bind .cmd.back <Leave> {.help configure -text ""}
bind .cmd.fwd <Enter> {
  .help configure -text "Page forward"}
bind .cmd.fwd <Leave> {.help configure -text ""}
```

Offensichtlich ist dieser Mechanismus sehr mächtig, weil das Verhalten isoliert eingestellt werden kann und dann autonom abläuft. In [12] wird die von den Autoren immer wieder gepriesene Ballonhilfe (freischwe-

bende Hilfestexte) als weiteres Beispiel für den Abschnitt 6.7.2 angekündigt.

8.2.3 Bindings für Klassen

Bis jetzt haben wir Verhalten für einzelne Widgets angegeben. Mit `bind` kann man aber das Verhalten einer ganzen Klasse von Widgets angeben. Die Klasse hat den selben Namen wie das Widget, nur mit großem Anfangsbuchstaben, also `Entry` statt `entry`, `Radiobutton` statt `radiobutton`, etc.

Hier z. B. ein `entry`-Widget, dessen Eingabefelder die Farbe wechseln sollen, wenn sie den Fokus empfangen, d. h. wenn das `FokusIn`-, bzw. das `FocusOut`-Ereignis eintritt (`entry.tcl`).

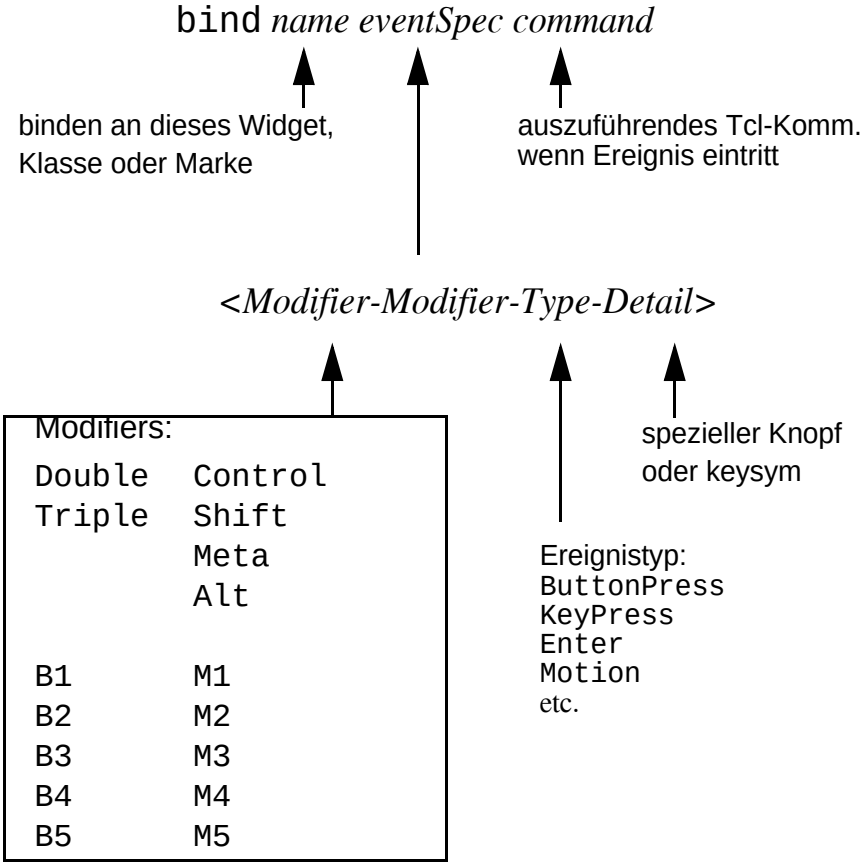
Die Bindung von `FocusIn` an die Klasse `Entry` geschieht wie folgt:

```
bind Entry <FocusIn> {%W configure -background white}
bind Entry <FocusOut> {%W configure -background gray}
```

Erhält in der Folge ein spezielles Feld den Fokus, wird durch Substitution von `%W` das aktuelle Widget, z. B. `.ent1`, eingesetzt.

8.3 Die Syntax des `bind`-Kommandos

Jetzt sind wir gerüstet, uns die Syntax von `bind` anzuschauen. Zusammengefaßt sieht sie wie folgt aus:



Ohne Angabe der Ereignisspezifikation und ohne Kommando erhalten wir die für das Widget gegenwärtig gesetzten Werte.

8.3.1 Die Ereignisspezifikation

Die Ereignisspezifikation anzugeben, ist trickreich. Es gibt aber ein Kochrezept, mit dem es immer klappt. Zunächst die Zutaten:

Der **Modifizierer** ist optional. Er kann z. B. eine Shift-Taste sein, die man gedrückt hält, während man gleichzeitig eine Maustaste betätigt. Es

kann auch eine Wiederholung sein, z. B. ein Doppel- oder Dreifachklick, sowie eine Kombination mehrerer Modifizierer, z. B. das beliebte Control + Shift bei Mausklick.

Der **Typus** ist das wichtigste Feld. Er gibt die Art des Ereignisses an, eine Mausbewegung, ein Tastendruck, eine Widgeteingabe, usw.

Die **Detail**angabe ist optional. Bei einem Mausereignis kann es die spezielle Maustaste sein, bei einem Tastaturereignis ist es die tatsächlich gedrückte Taste.

8.3.2 Ereignistypen

Die einzige unbedingt erforderliche Angabe ist der Ereignistyp, von dem es 22 verschiedene gibt.

In der Regel kommt man mit einem Dutzend aus, der Rest betrifft Feinheiten aus der Systemprogrammierung und kann im Tk-Handbuch nachgelesen werden.

In der folgenden Tabelle stellen die Angaben in Klammern Abkürzungen dar.

Ereignistyp	Beschreibung
ButtonPress (Button) ButtonRelease	Ausgelöst, wenn eine Maustaste gedrückt oder losgelassen wird.
KeyPress (Key) KeyRelease	Ausgelöst, wenn eine Tastaturtaste gedrückt oder losgelassen wird.
Motion	Ausgelöst, wenn die Maus über einem Widget bewegt wird; häufig in Verbindung mit `canvas`.
Enter Leave	Ausgelöst, wenn die Maus Widgetgrenzen überschreitet.
FocusIn FocusOut	Ausgelöst, wenn ein Widget den Tastaturfokus erhält oder verliert. Bestätigung von Benutzereingaben werden ggf. durch `FocusOut` behandelt.

Ereignistyp	Beschreibung
Map Unmap	Ausgelöst, wenn das Widget vom Bildschirm verschwindet oder angezeigt wird, z. B. nach Ikonisierung; kann verwandt werden, um ein Programm in den Wartezustand zu versetzen.
Configure	Ausgelöst von Fensterrekonfigurierungen, z. B. Fenstervergrößerungen (resize). Stößt z. B. das Neumalen von Leinwänden an.

Tab. 8–1 *Gebräuchliche Ereignistypen*

8.3.3 Modifizierer

Hiermit lassen sich Maus- und Tastaturereignisse weiter modifizieren, z. B. das elementare `Motion`-Ereignis

```
bind .x <Motion> {puts "moving"}
```

mit dem Modifizierer `Button1`, der jetzt nur auslöst, wenn gleichzeitig zur Bewegung noch die linke Maustaste gedrückt ist:

```
bind .x <Button1-Motion> {puts "moving, B1 pressed"}
```

Noch spezifischer wäre

```
bind .x <Shift-Button1-Motion> {
  puts "moving, Shifted and B1 pressed"}
```

Fährt man mit der Maus über das Widget, ohne Modifizierer zu drücken, erhält man die Ausgabe

```
moving
moving
moving
```

Drückt man jetzt die Maustaste oder Shift+Maustaste, dann wechselt die Ausgabe zu

```
moving, B1 pressed
moving, B1 pressed
moving, B1 pressed
```

bzw. zu der noch längeren Form. Damit ist sichergestellt, daß man mehrere Bindings an einem Widget anbringen kann und daß Tk immer das genau passende Verhalten auswählt.

Genauso können Mehrfachklicks eingesetzt werden, bei denen vorausgesetzt wird, daß die Mausklicks kurz hintereinander erfolgen und daß sich die Maus dabei nicht wesentlich bewegt.

Die verfügbaren Modifizierer lassen sich in einer Tabelle (siehe unten) zusammenfassen.

Modifizierer	Beschreibung
Control Shift Meta Alt	Gedrückte Modifiziertaste muß gedrückt sein, wenn Ereignis eintritt
Button1 (B1) ... Button5 (B5)	Angegebene Maustaste muß gedrückt sein, wenn Ereignis eintritt. Vorgesehen sind bis zu 5 Tasten
Double Triple	Ereignis muß wiederholt auftreten, üblicherweise bei Maustasten ohne Mausbewegung und in schneller Abfolge
Mod1 (M1) ... Mod5 (M5)	Deutet an, daß gewählte Modifiziertaste gedrückt sein muß, wenn Ereignis eintritt. Alte Notation für X-Terminals, besser genaue Beschreibung (`Alt` oder `Meta`) von oben verwenden

Tab. 8–2 *Modifizierer für bind*

8.3.4 Details

Die letzte Angabe betrifft spezielle Maus- und Tastaturereignisse, wie aus der Tabelle ersichtlich.

Ereignisdetail	Beschreibung
Maustastennr.	Wenn das Ereignis `ButtonPress` oder `ButtonRelease` ist, kann die Tastennummer (1-5) angegeben werden.
Key Symbol	Für ein `KeyPress` oder `KeyRelease` Ereignis, kann das Detail in Form eines X Window keysym angegeben werden.

Tab. 8–3 *Detailangabe für bind*

Das keysym für eine alphanumerische Taste ist der Buchstabe. Entsprechend wird ein kleines c mit

```
bind .x <KeyPress-c> {puts "gedrücktes c"}
```

angebunden, ein großes C mit

```
bind .x <KeyPress-C> {puts "gepreßtes hohes C"}
```

was wiederum gleichwertig ist mit

```
bind .x <Shift-KeyPress-c> {puts "großes C"}
```

Großbuchstaben sollte man vermeiden, sofern nicht ausdrücklich auf die `Shift`-Taste verwiesen wird. H+McL geben ein einleuchtendes Beispiel für diese Empfehlung:

```
bind .x <Control-KeyPress-C> {puts "gedrücktes ^C"}
```

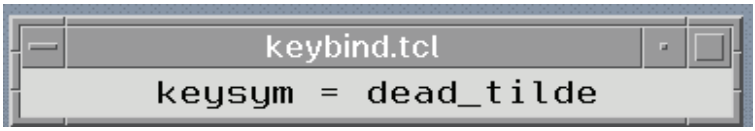
sieht aus wie das *Control-C Ereignis*, ist aber tatsächlich das *Shift-Control-C Ereignis*!

Viele Tasten werden durch symbolische Namen bezeichnet, z. B. +, -, <, >, die Funktionstasten, die Leertaste (space) und Eingabetaste (Return). Um diese herauszubekommen, läßt man sich die Bezeichner mit %K anzeigen (`keybind.tcl`).

```
label -msg -width 30 -text "Press any key"
pack .msg
```

```
bind .msg <Key> {.msg configure -text "keysym = %K"}
focus .msg
```

Im Bild unten haben wir auf der deutschen RS/6000-Tastatur Alt-Gr und die (*, +, ~)-Taste gedrückt, um das Tildezeichen – ein sog. Tottastenzeichen – zu produzieren.



8.3.5 Rezept zur Angabe der Ereignissequenz

Man überlegt sich zuerst den Ereignistyp. Wenn es kein `ButtonPress/ButtonRelease` oder `KeyPress/KeyRelease` Ereignis ist, ist man fertig, weil nur Maus- und Tastaturereignisse eine Erweiterung zulassen.

Jetzt füge man die Detailangaben hinzu, also ggf. eine Tastenbezeichnung oder die Nummer der Maustaste; möchte man, daß das Ereignis für beliebigen Tastendruck oder alle Maustasten gilt, läßt man das Detail weg. Schließlich gibt man die Modifizierer an, sofern gewünscht.

Im folgenden basteln wir uns einige Ereignisse.

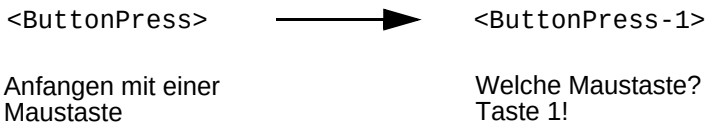
8.3.6 Unterschied Mausmodifizierer und Mausdetail

Betrachtet man eine Mausbewegung mit Maustaste-1 gedrückt und Shift gehalten, also

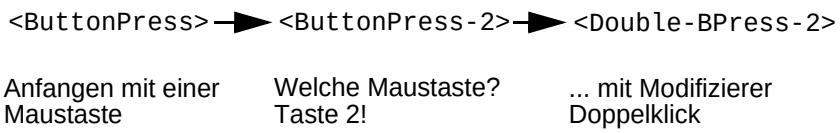
```
<Shift-Button1-Motion>
```

könnte man versucht sein, daraus ein `<Shift-Motion-1>` zu machen. Dies würde aber einen Fehler „specified button ‘1’ for non-button event“ liefern. Der Grund ist, daß das Detailfeld „-1“ nur in Verbindung steht mit einem `ButtonPress/ButtonRelease` bzw. `KeyPress/KeyRelease`, keinem Bewegungsereignis! Das verlangt auch, daß die Maustaste schon gedrückt ist, bevor das Ereignis (Mausbewegung) eintritt!

Klick mit Maustaste 1



Doppelklick mit Maustaste 2



Beliebige Taste an Tastatur gedrückt

8.3.7 Abkürzungen

Einige Bindingfolgen treten so häufig auf, daß Abkürzungen dafür vorgesehen sind, z. B. für `<ButtonPress-1>` bis `<ButtonPress-5>` schreibt man `<1>` bis `<5>`. Genauso kürzt man Tastatureingaben durch Angabe des Details ab, also für `<KeyPress-a>` nur `<a>` oder sogar nur `a`.

H+McL raten davon ab, weil es manche Leser verwirrt; Ausnahmen sehen sie z. B. für `<Control-c>` was eindeutig lesbarer ist als `<Control-KeyPress-c>`.

8.3.8 Ereigniskombinationen

Viele Programme, wie z. B. emacs, verwenden Ereignisfolgen um dadurch Befehle in Form von Tastaturkürzeln anzustoßen. Bekannt ist aus emacs z. B. Control-x Control-s zum Speichern einer Datei.

Dies läßt sich auch in Tcl/Tk machen, etwa mit

```
bind . <Control-x><Control-f> {load_file}
bind . <Control-x><Control-s> {saveFile}
```

Diese werden hier an das Hauptfenster „`.`“ gebunden. Dadurch können sie überall getippt werden.

H+McL sehen im Bau einer Geheimtür eine weitere Anwendung dieser Tastaturfolgen, um z. B. mit nichtdokumentierten Passwörtern Eigenschaften ein- und auszuschalten, etwa einen Powermodus in einem Spiel:

```
bind . <KeyPress-d><n><r><c> {set power_mode on}
```

Dies erfordert keine weiteren Vorkehrungen, um einen Sonderstatus herbeizuführen.

8.3.9 Die Prozentsubstitution

Bevor Tk das Kommando für ein angegebenes Verhalten ausführt, werden Informationen über das Ereignis in Felder, die mit % beginnen, substituiert. Es gibt eine ganze Reihe dieser Parameter (siehe `bind` Handbuchseite), aber nur wenige sind allgemein nützlich.

Z-kette	Ersetzung
%%	Ersetzt durch einzelnes Prozentzeichen
%W	Name des Fensters, das das Ereignis empfängt. Gültig für alle Ereignistypen.
%b	Die Nummer der Maustaste, die gedrückt oder losgelassen wurde. Nur gültig für `ButtonPress` und `ButtonRelease` Ereignisse.
%A %K	Die gedrückte oder losgelassene Tastaturtaste. %A ist das ASCII-Zeichen der Taste, eine leere Zeichenkette sonst (wenn kein ASCII-Zeichen). %K ist der symbolische Name. Gültig nur für `KeyPress` und `KeyRelease`.
%h %w	Höhe und Breite des Widgets. Nur gültig für `Configure` und `Expose` Ereignisse.
%x %y	Die *x* und *y* Koordinaten des Mauszeigers zum Zeitpunkt des Ereigniseintritts. Die Werte sind relativ zum Ursprung (0, 0) der linken, oberen Ecke des Widgets, das das Ereignis empfängt.

Z-kette	Ersetzung
%X %Y	Wie %x und %y, aber relativ zum Ursprung (0, 0) des Desktops. Nur gültig für `ButtonPress`, `ButtonRelease`, `KeyPress`, `KeyRelease` und `Motion` Ereignisse.

Tab. 8–4 *Prozentersetzungen*

Das folgende Beispiel zeigt eine Anwendung, die einen farbigen Text an die Position schreibt, an der gegenwärtig der Mauszeiger steht.

```
canvas .c -background white
pack .c
array set colors {1 red 2 green 3 blue}
bind .c <ButtonPress> {
 .c create text %x %y -text "click!" -fill $colors(%b)
}
```

Sorgfalt ist wieder angesagt in Zusammenhang mit Fluchtsymbolen. Will man etwa ein Zeichen in einem Textwidget hinten einfügen, könnte man schreiben:

```
bind .Text <KeyPress > {%W insert end %A}
```

woraus für ein spezielles Widget `.t` und ein eingegebenes `X` Tk das Kommando

```
.t insert end X
```

ausführt. Für ein exotischeres Zeichen, etwa die eckige Klammer `[`, setzt Tk ein Fluchtsymbol davor, damit es nicht versehentlich als Start eines eingebetteten Komandos interpretiert wird, d. h. Tk führt das Kommando

```
.t insert end \[
```

aus.

Hätte der Anwender aber im `bind`-Kommando geschweifte Klammern um den Parameter gesetzt, also geschrieben

```
bind .Text <KeyPress > {%W insert end {%A}}
```

hätte Tk

```
.t insert end {\[}
```

ausgeführt und damit \ und [eingefügt. Deshalb raten H+McL zur Verwendung von unmodifizierten bzw. nur mit doppelten Anführungszeichen geschützten Parametern.

8.4 Komplexere Ereignisse

Schauen wir uns jetzt einige komplexere Beispiele an. In ihnen kombinieren wir mehrere Bindings, um eine Aufgabe zu lösen.

8.4.1 Click, drag, drop

Eine Reihe bekannter Operationen kombiniert Mausklicks mit Ziehen der Maus.

- hervorheben von Text mit der Maus
- den Rollbalken ziehen
- ein graphisches Objekt durch Aufziehen erzeugen
- ein Programm steuern durch Ziehen und Ablegen von Bildschirmelementen

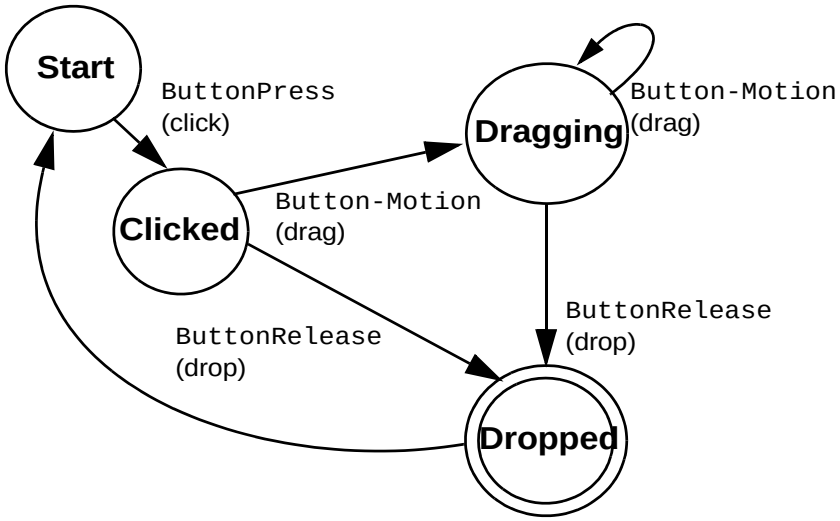
Einige dieser Operationen – wie die Hervorhebung von Text und das Ziehen am Rollbalken – sind bereits in Tk eingebaut. Andere müssen durch eine Folge von `bind`-Kommandos implementiert werden.

Obwohl sich die Operationen im Detail unterscheiden, liegt ihnen ein gemeinsames Modell zugrunde.

1. Man wählt ein Objekt aus oder startet eine Aktion durch Anklicken des Objekts und Festhalten der Taste. Dies löst ein Skript aus, das das Element auswählt oder ein neues Objekt schafft.
2. Man bewegt oder verändert das Objekt durch Ziehen der Maus bei gedrückter Taste. Gewöhnlich wird dabei die Anzeige stets neu gemalt, damit man sofortige Rückmeldung über die Aktion erhält. Zieht man z. B. ein Objekt folgt es dem Mauszeiger, d. h. die Neuanzeige erfolgt permanent.

3. Man beendet die Aktion durch Freigeben der Maustaste. Man kann sich dies als „Fallenlassen“ des Objekts (drop) vorstellen. Damit sind die Änderungen abgeschlossen. Wäre dies etwa die Auswahl von graphischen Objekten mit einem Gummiband, dann wären alle eingeschlossenen Objekte selektiert.

Die folgende Abbildung zeigt ein Zustandsdiagramm mit den möglichen Übergängen.



Man kann die Abfolge der Ereignisse leicht mit den folgenden Zeilen beobachten (`cdr1.tcl`).

```
pack [canvas .c]
bind .c <ButtonPress-1> { puts "click %x %y" }
bind .c <B1-Motion> { puts "drag %x %y" }
bind .c <ButtonRelease-1> { puts "drop %x %y" }
```

Ein Programmlauf erzeugt eine Ausgabe wie die folgende

```
click 104 91
drag 105 92
drag 121 99
drag 123 100
drag 126 101
drag 143 111
drag 145 112
```

```
drag 159 117
drag 159 118
drop 159 118
```

Man sieht daran, daß das Click- und das Drop-Ereignis genau einmal stattfindet. Dazwischen erfolgen mehrere Bewegungsereignisse, die aber auch fehlen können! Bei schnellem Ziehen bekommt man natürlich nicht ein Ereignis für jedes Pixel sondern nur die grobe Linie.

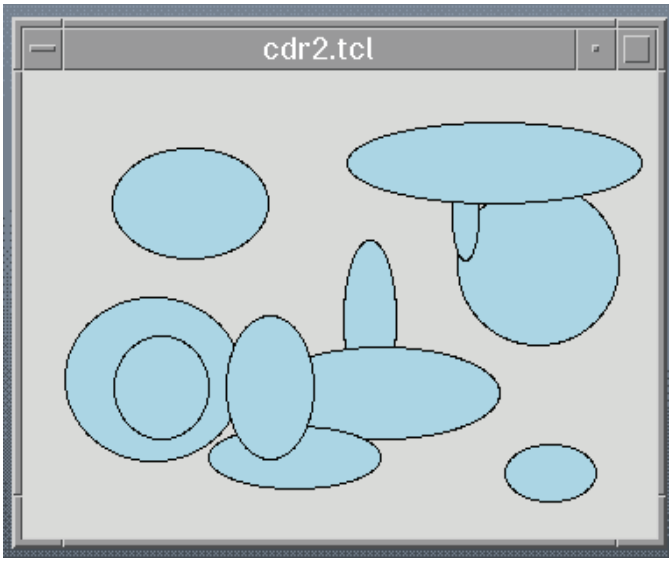
Wir wollen jetzt diese Bindings in einem einfachen Zeicheneditor verwenden, der Ovale auf einer Leinwand zeichnet. Der Code sieht wie folgt aus (`cdr2.tcl`).

```
pack [canvas .c]

bind .c <ButtonPress-1> {oval_create %W %x %y}
bind .c <B1-Motion> {oval_move %W %x %y}
bind .c <ButtonRelease-1> {oval_end %W %x %y}

proc oval_create {win x y} {
  global oval
  set oval(x0) $x
  set oval(y0) $y
  set oval(id) [$win create oval $x $y $x $y]
}

proc oval_move {win x y} {
  global oval
  $win coords $oval(id) $oval(x0) $oval(y0) $x $y
}

proc oval_end {win x y} {
  global oval
  oval_move $win $x $y
  $win itemconfigure $oval(id) -fill lightblue
}
```

Click ruft `oval_create` auf mit den Click-Koordinaten. Die Prozedur erzeugt an diesen Koordinaten ein Oval. Sie speichert die Werte und eine Identifikationsnummer in einem Vektor `oval`.

Drag ruft `oval_move` auf und adjustiert die Größe auf die durch Anfangskoordinaten und jetzige Mauskoordinaten gegebene Ausdehnung.

Drop ruft `oval_end` auf mit den Endkoordinaten. Mit diesen Werten wird das Oval ein letztes Mal adjustiert und bekommt eine neue Füllfarbe. Da sowohl die Drag- als auch die Dropbearbeitung die Größe verändern, ist es wohl besser, diese Tätigkeit in einer eigenen Prozedur, z. B. namens `oval_move`, zusammenzufassen.

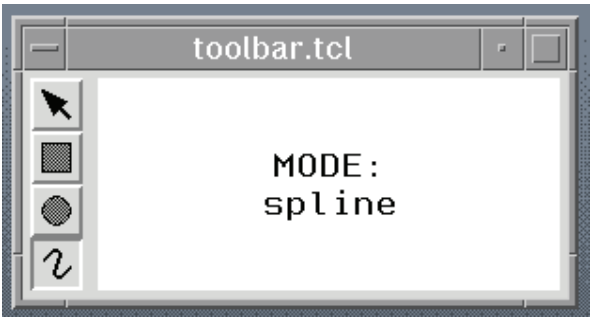


H+McL geben auch Gründe an, warum man die Bindings besser in Prozeduren verpackt und man diesen Argumente, also `%W`, `%x` und `%y`, übergibt, statt den Code direkt in das `bind`-Kommando einzufügen.

- Keine Gefahr des versehentlichen Überladens von globalen Variablen, z. B. durch `set x0 [expr %x - 4]`.
- Keine Probleme mit %-Substitution
- Portabilität des Codes; die Oval-Prozeduren können überall wiederverwendet werden
- bessere Lesbarkeit.

8.4.2 Anpassung von Widget-Verhalten

Das `bind`-Kommando macht Tk außerordentlich anpassungsfähig. So kann man einem **label** ein Verhalten verordnen, das ihn wie einen Menüknopf aussehen läßt mit entsprechenden Reaktionen. Dazu könnte man zwar gleich einen **button** verwenden, aber manchmal braucht man nur eine „Art von Knopf“ mit gewissen Anpassungen.



Die Werkzeugleiste in der Abbildung oben ist so ein Beispiel. Wählt man ein Werkzeug aus, erscheint es abgesenkt, die anderen erhaben. Man kann wie bei Radioknöpfen nur ein Element auswählen, aber die Leiste sieht nicht wie **radiobuttons** aus.

Das Programm dazu enthält eine Menge raffinierter Tricks mit generischen `toolbar_create`, `toolbar_add` und `toolbar_select` Prozeduren, die alle in der Efftcl-Library zu finden sind. Hier zunächst das Programm `toolbar.tcl`.

```
toolbar_create .tbar top {
 .show configure -text "MODE:\n%t"
}
pack .tbar -side left -fill y -padx 4 -pady 4

label .show -width 20 -background white
pack .show -side right -expand yes -fill both \
  -padx 4 -pady 4

toolbar_add .tbar select [image create bitmap \
  -file [file join $env(EFFTCL_LIBRARY) images \
  select.xbm]]

toolbar_add .tbar rect [image create bitmap \
```

```
    -file [file join $env(EFFTCL_LIBRARY) images \
    rect.xbm]]

toolbar_add .tbar oval [image create bitmap \
    -file [file join $env(EFFTCL_LIBRARY) images \
    oval.xbm]]

toolbar_add .tbar spline [image create bitmap \
    -file [file join $env(EFFTCL_LIBRARY) images \
    spline.xbm]]
```

Wir werden unten eine Prozedur `toolbar_create` einführen, die drei Argumente verlangt:

- den Namen der Werkzeugleiste, hier `.tbar`
- die Orientierung, wo die Leiste hinkommt, hier `top`,
- eine Callback-Routine, die in unserem Fall *`%t`* substituiert durch den Namen des gewählten Werkzeugs (`select`, `rect`, `oval`, `spline`) und ihn auf der Leinwand anzeigt.

Danach werden wir eine Prozedur `toolbar_add` entwerfen müssen, die neue Werkzeuge einfügt. Sie verlangt als Argumente den Namen der Leiste, also `.tbar`, den Namen des Werkzeugs, also z. B. `rect`, und ein Bitmap-Bild als Icon. Daraus macht die Prozedur ein Widget und packt es in die Leiste.

Nun zunächst `toolbar_create`.

```
proc toolbar_create {win {origin "top"} {command ""}} {
    global tbInfo

    frame $win -class Toolbar

    set tbInfo($win-current) ""
    set tbInfo($win-origin) $origin
    set tbInfo($win-command) $command
    return $win
}
```

Zunächst wird ein `frame` angelegt, dessen Optionen und Bindings sich aus der Klasse `Toolbar` in der Datenbank für Standardvorgaben ergeben. Die Datenstruktur für dieses Widget enthält drei Felder:

- `current` für den Namen des gerade gewählten Werkzeugs
- `origin` für die Orientierung beim Packen
- `command` für die Callback-Routine

Zunächst ist die Leiste leer und wir fügen nacheinander mittels der folgenden generischen Prozedur die Werkzeuge ein.

```
proc toolbar_add {win tool image} {
  global tbInfo
  set label "$win.tool-$tool"
  label $label -borderwidth 2 -relief raised \
               -image $image
  pack $label -side $tbInfo($win-origin) -fill both

  bind $label <ButtonPress-1> \
    [list toolbar_select $win $tool]

  if {[llength [pack slaves $win]] == 1} {
    after idle [list toolbar_select $win $tool]
  }
  return $label
}
```

Die Routine `toolbar_add` erzeugt einen `label`, dessen Name zusammengesetzt wird aus dem Leistenbezeichner (Argument `win`) und dem Werkzeugnamen (Argument `tool`). Sein Icon ist das dritte Argument (`image`). Seine Plazierung ergibt sich aus dem in `tbInfo` abgelegten `origin`-Wert.

Zunächst wäre jede der Marken inaktiv. Wir binden jetzt aber die Maustaste 1 an die Marke, wobei die gebundene Rountine `toolbar_select` ist, die wir noch angeben müssen.

Zunächst aber sorgen wir dafür, daß standardmäßig immer das erste Werkzeug ausgewählt ist. Das richten wir ein, wenn wir das erste Werkzeug anlegen. Um herauszubekommen, wieviele es gerade sind, rufen wir `pack slaves` auf, das die Liste der Unterwidgets eines angegebenen Oberwidgets liefert. Ist die Länge der Liste = 1, wählen wir es aus. Allerdings verzögern wir die Auswahl (`after idle`), bis das ganze Widget angelegt wurde. Der Grund ist, daß ggf. noch Teile des Widgets fehlen könnten, auf die in der `select`-Routine Bezug genommen wird. Dies ist

laut H+McL eine Grundregel für den Einsatz von Callbacks während der Konstruktion eines Widgets.

Vorläufiger krönender Abschluß ist die `toolbar_select`-Routine.

```
proc toolbar_select {win tool} {
  global tbInfo

  if {$tbInfo($win-current) != ""} {
    set label "$win.tool-$tbInfo($win-current)"
    $label configure -relief raised
  }
  set label "$win.tool-$tool"
  $label configure -relief sunken

  set tbInfo($win-current) $tool

  if {$tbInfo($win-command) != ""} {
    set cmd [percent_subst %t \
      $tbInfo($win-command) $tool]
    uplevel #0 $cmd
  }
}
```

Zunächst wird für ein bisher gesetztes Tool, ermittelbar aus `$tbInfo($win-current)`, dessen Aussehen wieder auf „nicht-ausgewählt“ (Relief oben) gesetzt. Das neu ausgewählte Werkzeug erhält dann das abgesenkte Aussehen und `$tool` wird als neue Wahl in `tbInfo` gesichert.

Danach soll die mit dem Auswahlvorgang verbundene Callback-Routine ausgeführt werden, wobei zunächst der `tool`-Wert für `%t` in der Routine `$tbInfo($win-command)` substituiert werden muß. Diese so aufbereitete Routine wird dann auf oberster Ebene (`uplevel #0`), also im für Bindings üblichen globalen Kontext, aufgerufen.

Damit haben wir gezeigt, wie man durch eigene Bindings ein neues Verhalten, hier das einer Radioleiste, erzeugen kann.

8.5 Bindemarken

Ein Widget kann auf ein einzelnes Ereignis mit mehr als einem Verhalten reagieren. Nehmen wir wish und eine Marke und binden wir zwei Kommandos daran für das selbe Ereignis.

```
$ wish
% pack [label .x -text "Ziel"]
% bind .x <Enter> {puts "entering %W (via %W)"}
% bind Label <Enter> {puts "entering %W (via Label)"}
```

Zieht man mit der Maus über das Widget wird die eine oder andere, oder beide, der zugebundenen Reaktionen gezeigt.

```
entering .x (via .x)
entering .x (via Label)
```

Will man sich anzeigen lassen, welche Bindings existieren und in welcher Reihenfolge diese zur Ausführung kommen, kann man sich dies mit dem Kommando `bindtags` ausgeben lassen.

```
% bindtags .x
.x Label . all
```

Demnach reagiert das Widget zuerst mit dem Binding für `.x`, danach mit dem Binding für die Klasse `Label`, danach ... für das globale Widget `.`, zuletzt ... für `all`(es).

Mit dem `bindtags`-Kommando lassen sich nicht nur die Bindings anzeigen, man kann die Bindemarken auch neu anordnen, z. B. durch

```
% bindtags .x {Label .x . all}
```

wobei jetzt die Klasse `Label` mit ihrem Binding nach vorne gerückt ist.

```
entering .x (via Label)
entering .x (via .x)
```

Die obige Reaktion auf eine Mausbewegung zeigt jetzt das veränderte Verhalten.

Auch möglich wäre das Weglassen einzelner Bindings oder das Ausschalten aller Bindings mittels

```
% bindtags .x { }
```

wobei das Leerzeichen innerhalb der Klammern wichtig ist. Ohne dieses Leerzeichen, also mit

```
% bindtags .x {}
```

haben wir den leeren String und dies setzt die Bindemarken wieder auf die Standardvorbesetzung zurück, also auf

```
.x Label . all
```

8.5.1 Vorbesetzung der Bindemarken

Gewöhnliche Widgets erhalten beim Anlegen eine Liste mit vier Elementen in der Reihenfolge:

- Widget-Bezeichner, hier `.x`
- Klassenbezeichner, hier `Label`
- Bezeichner des obersten Widgets, hier `.`
- Schlüsselwort `all`

Damit kann spezielles Widgetverhalten allgemeine Bindings überlagern, das einer Klasse wiederum das des obersten Widgets, d. h. gewöhnlich das eines Fensters.

Andererseits kann man leicht globales Verhalten, etwa ein Tastenkürzel für Abbruch, an oberste Widgets oder per `all` an die ganze Anwendung binden, z. B. mit

```
bind . <Control-z> do_undo
```

an das Hauptfenster und alle eingefügten Elemente

```
pack [entry .e1]
pack [entry .e2]
pack [entry .e3]
```

hätten `.` in ihrer Markenliste und würden daher das „undo“-Verhalten mittels Tastenkürzel `^z` erben.

Für Widgets auf oberster Ebene entfällt naturgemäß die Marke `.`, d. h. ein solches Widget, z. B. `.t`, mit Klasse `Toplevel`, hat nur die Liste

```
.t Toplevel all
```

d. h., der dritte Eintrag in den obigen Listen entfällt.

8.5.2 Der Gebrauch von `break` in der Ereignisbehandlung

Da der Widgetbezeichner in der Bindingsliste zuerst erscheint, kann man Widgets bequem ein spezielles Verhalten beibringen, das vom Klassenverhalten abweicht.

Im folgenden Beispiel wird ein Eingabewidget für Telefonnummern angegeben, das – abweichend vom üblichen Verhalten von Eingabewidgets – nur Ziffern, den Bindestrich und die Rücktaste akzeptiert. Im Fall einer anderen Eingabe ertönt ein Piepston.

Für dieses spezielle Verhalten binden wir das `KeyPress`-Ereignis an das Widget (`valid.tcl`):

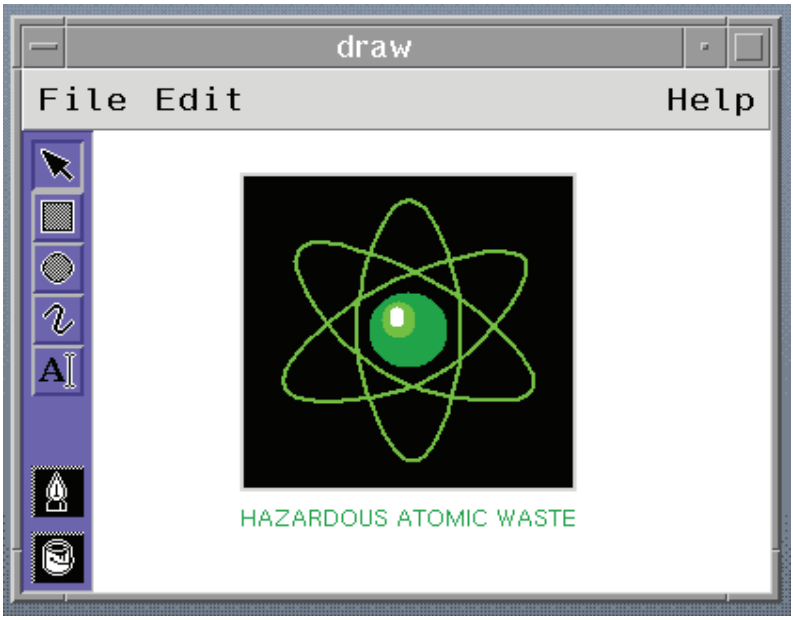
```
label .prompt -text "Phone Number:"
pack .prompt -side left
entry .phone
pack .phone -side left

bind .phone <KeyPress> {
  if {![regexp "\[0-9\]|-|\b" "%A"]} {
     bell
     break
  }
}
```

Wichtig daran ist, neben dem Gebrauch eines regulären Ausdrucks, das Auftreten des `break`-Kommandos im Binding. Normalerweise wird mit `break` eine Schleife verlassen, hier jedoch beendet es alle weiteren

Behandlungsversuche, speziell die der `Entry`-Klasse, wodurch das nicht gewünschte Zeichen aus %A sonst doch noch akzeptiert würde.

8.5.3 Hinzufügen neuer Verbindungsmarken

In dem Zeichenbrett unten können verschiedene Zeichenmodi mit der Werkzeugleiste links ausgewählt werden. Jeder der gewählten Modi reagiert anders auf Maus und Tastatur. Der „Oval-Modus“ zum Zeichnen oder Ziehen einer Ellipse kennt z. B. `<ButtonPress-1>`, `<B1-Motion>` und `<ButtonRelease-1>`..



Nun könnte man dem Widget `.canvas` durch individuelles Setzen von Bindings für die o. g. Ereignisse entsprechendes Verhalten beibringen, wenn das Oval-Werkzeug ausgewählt wurde. Wird ein anderes Werkzeug anschließend gewählt, müssen die Bindings wieder ausgeschaltet werden, z. B. durch leere Zeichenketten

```
bind .drawing <ButtonPress-1> { }
```

Das ist nicht gerade elegant. Besser geht es durch eine neue Bindemarke (**binding tag**), sagen wir, `oval`, der wir die drei Bindings zuordnen.

```
bind oval <ButtonPress-1> {
  canvas_shape_create %W oval %x %y
}

bind oval <B1-Motion> {
  canvas_shape_drag %W %x %y
}

bind oval <ButtonRelease-1> {
  canvas_shape_end %W %x %y
}
```

Jetzt schalten wir das Verhalten für den „Oval-Modus“ mittels `bindtags` ein:

```
bindtags .drawing {oval .drawing Canvas . all}
```

und wieder aus:

```
bindtags .drawing {.drawing Canvas . all}
```

Genauso können wir eine Bindemarke `text` einführen und über sie das Verhalten für die Texteingabe im Zeichenbrett definieren. Wieder setzen wir `text` an die erste Stelle der Marken für `.drawing`, wenn der Anwender Textmodus gewählt hat, und löschen `text` in der Liste, wenn ein anderer Modus gewählt wurde.

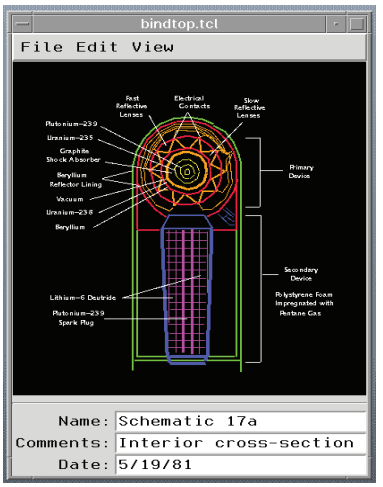
Neben den Widgetbezeichnern und den Klassennamen lassen sich demnach beliebige andere Marken einführen, für die Bindings definierbar sind und deren Verhalten über `bindtags` ein- und ausgeschaltet werden kann.

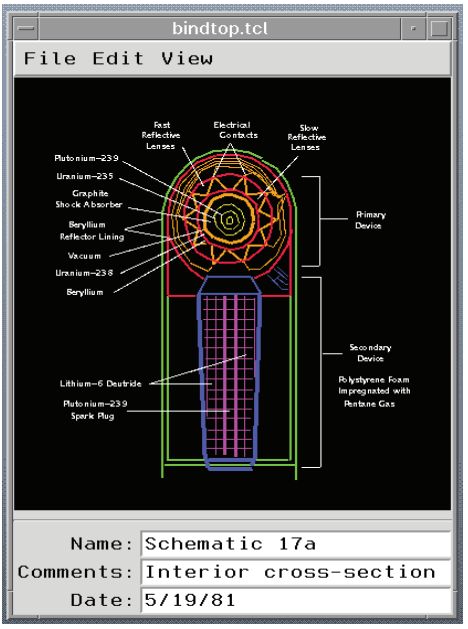
8.5.4 Verhalten am obersten Fenster anbinden

Oben haben wir vorgeschlagen, gewisse Tastenkürzel direkt am obersten Widget, dem Fenster „`.`“, festzumachen, damit alle Widgets darin das Verhalten erben. Das macht für Tastenkürzel Sinn, kann aber sonst unerwünscht sein.

Im Beispiel unten enthält die Leinwand eine größere Graphik. Wird das Fenster in der Größe verändert, ändert sich auch die Zeichnung. Das könnte man erreichen, indem man die Prozedur `resize_drawing` an

den <Configure>-Event im Hauptfenster bindet. Zunächst sieht das vernünftig aus.





Allerdings haben wir das Verhalten auch an alle Unterobjekte vererbt. Da diese alle auch das <Configure>-Ereignis erhalten, rufen sie auch `resize_drawing` auf.

Beim Laden des Widgets tritt dadurch eine merkbare Verzögerung ein. Bei genauem Hinsehen (`wish bindtop.tcl` aufrufen) wird die Leinwand rund 15 mal neu gemalt!

Eine Lösung wäre, das Neumalen der Leinwand nur an den `Configure`-Event der Leinwand zu binden:

```
bind .display <Configure> {
  resize_drawing .display
}
```

Falls man aber im Hauptfenster aus irgendwelchen Gründen das `Configure`-Ereignis erkennen möchte, würde diese Lösung nicht genügen. Dagegen kann man wie schon vorhin eine neue Marke einführen, sagen

wir diesmal `resizeDrawing`, und nur an diese das Neumalen der Leinwand binden.

```
bind resizeDrawing <Configure> {
  resize_drawing .display
}
set tags [bindtags .]
bindtags . [listinsert $tags 0 resizeDrawing]
```

Nur das oberste Fenster hat diese Bindungsmarke und nur `.` ruft daher `resize_drawing` auf. Viele andere Unterwidgets erhalten weiterhin das `<Configure>`-Ereignis und können entsprechend darauf reagieren, jedoch ohne Aufruf der Prozedur zum Neumalen der Leinwand.

8.6 Fehlersuche mit bindings

Oft wundert man sich bei einer Anwendung, welches Verhalten ein Widget hat und welche Ereignisse es erhält.

8.6.1 Bindings anzeigen

Falls sich ein Widget nicht wie gewünscht verhält, kann man die folgende Prozedur (aus dem Library Code von H+McL) verwenden, um sich eine Zusammenstellung der Bindings auflisten zu lassen.

```
proc bind_show {w {mode "-quiet"}} {
  puts "$w"
  foreach tag [bindtags $w] {
    puts "\t$tag"
    foreach spec [bind $tag] {
      puts "\t\t$spec"
      if {$mode == "-verbose"} {
        set cmd [bind $tag $spec]
        set cmd [string trim $cmd "\n"]
        regsub -all "\n" $cmd "\n\t\t\t" cmd
        puts "\t\t\t$cmd"
      }
    }
  }
}
```

Die Prozedur `bind_show` listet für ein Widget (Parameter `w`) den Namen und danach eingerückt seine Tags und je Tag deren definierte Events auf. So etwa für einen Knopf, dem wir ein einziges eigenes Binding für das `Enter`-Ereignis mitgeben:

```
% button .b
.b
% bind .b <Enter> { puts "jetzt in %W" }
% bind_show .b
invalid command name "bind_show"
% package require Efftcl
1.0
% bind_show .b
.b
  .b
    <Enter>
  Button
    <ButtonRelease-1>
    <Button-1>
    <Leave>
    <Enter>
    <Key-space>
  .
  all
    <Shift-Key-Tab>
    <Key-Tab>
    <Key-F10>
    <Alt-Key>
%
```

Wie man aber sieht, hat das Widget viele weitere Standardbindings. Mit der Option `-verbose` kann man sich auch die Kommandos dazu ansehen.

8.6.2 Ereignisse überwachen

Manchmal wundert man sich, wie der Strom der Ereignisse wirklich verarbeitet wird. Wenn man auf ein Widget mit der Maus klickt, die Maus nach außen zieht, die Maustaste losläßt, kommt dann wirklich das `Leave`- vor dem `ButtonRelease`-Ereignis?

Die folgende Prozedur erzeugt spezielle Bindings mit denen sich alle Ereignisse in einem Fenster überwachen lassen.

```
proc bind_debug {w on} {
  set events {
    {ButtonPress {W=%W #=%# x=%x y=%y b=%b s=%s }}
    {ButtonRelease {W=%W #=%# x=%x y=%y b=%b s=%s }}
    {Circulate {W=%W #=%# x=%x y=%y p=%p }}
    . . .
    {Visibility {W=%W #=%# x=%x y=%y s=%s }}
  }

  foreach e $events {
    set type [lindex $e 0]
    set fmt [lindex $e 1]
    bind BindDebugger <$type> "puts \"<$type> $fmt\""
  }

  set allwin [bind_debug_allwindows $w]
  foreach w $allwin {
    set tags [bindtags $w]
    set i [lsearch $tags BindDebugger]
    if {$on} {
      if {$i < 0} {
        set tags [linsert $tags 0 BindDebugger]
        bindtags $w $tags
      }
    } else {
      if {$i >= 0} {
        set tags [lreplace $tags $i $i]
        bindtags $w $tags
      }
    }
  }
}
```

Die Prozedur erzeugt eine Bindemarke `BindDebugger` und verbindet damit eine Folge von `puts`-Kommandos. Jedes Ereignis hat seine eigene Botschaft, die wichtige Daten für das Ereignis ausgibt.

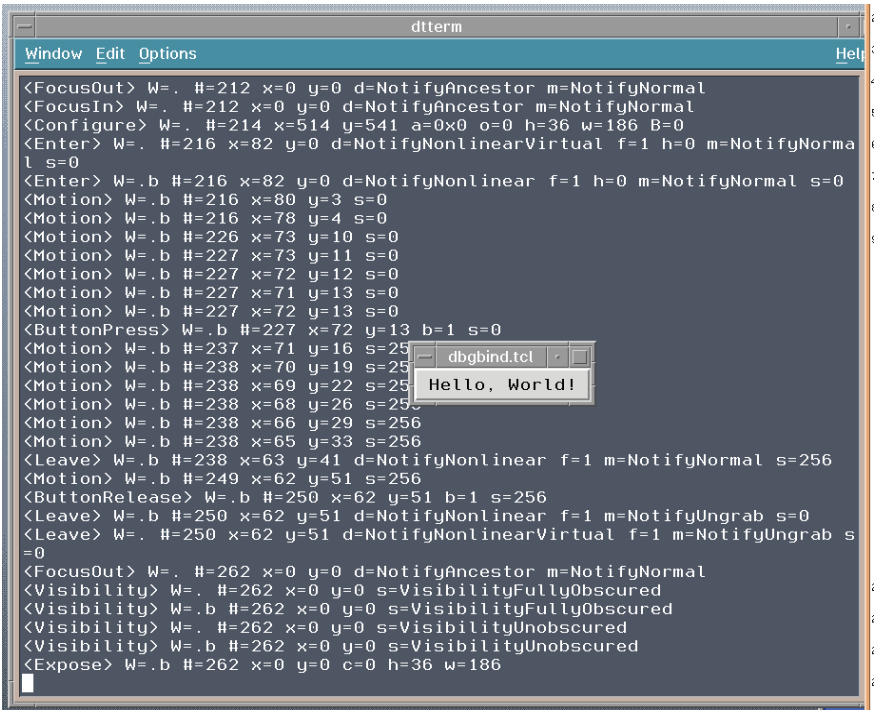
Die Bindings werden in einer Schleife erzeugt, wären sie von Hand codiert worden, sähen die Bindings so aus:

```
bind BindDebugger <ButtonPress> \
  "puts \"<ButtonPress> \
  {W=%W #=%# x=%x y=%y b=%b s=%s }\""
```

Die Prozedur schaltet jetzt Debugging an und aus, je nach Argument. Wenn das Argument "`on`" war, fügt sie `BindDebugger` überall ein, sonst entfernt sie die Marke aus allen bindings.

Mit dieser Prozedur können wir jetzt die Frage über das Verhalten beim Ziehen mit der Maus beantworten. In den Beispielen gibt es hierzu das `dbgbind.tcl` Skript.

```
package require Efftcl
button .b -text "Hello, World!" -command exit
pack .b

bind_debug . on
```



Wir sehen, daß es beim Verlassen des Widgets mit der Maus ein `Leave`-Ereignis gibt, danach mehre `Motion`-Ereignisse, dann nochmals ein `Leave`-Ereignis (zweifach gemeldet). Das allererste `Leave` noch vor

`ButtonRelease` ist das normale Verlassen der Parzelle. Zugleich erhält das Widget aber weiterhin Mausereignisse, weil es eine Art implizites `grab` auf dem Widget gesetzt hat. Dies meldet `.b` das Loslassen der Taste und danach ein virtuelles `Leave` (`m=NotifyUngrab`) für `.b` und `.`, womit die Maus den Widgetkontext endgültig verlassen hat.

8.7 Animation

Durch das `after`-Kommando kann zeitgesteuert ein Skript ausgeführt werden. So erzeugt das folgende Kommando die Botschaft `Hello, World!` nach einer Pause von 1 000 ms:

```
after 1000 {puts "Hello, World!"}
```

Durch Aneinanderreihen von `after`-Ereignissen kann man Dinge zeitabhängig ablaufen lassen, wie wenn sie im Hintergrund abgearbeitet würden.

Das Grundkonzept sieht wie folgt aus:

```
proc animate {} {
  ...                ;# mach was
  after 100 animate  ;# Neuansetzen der Aktion
}
animate              ;# Start der Aktion
```

Die Prozedur `animate` erledigt einige Aufgaben und beendigt sich, nicht jedoch, ohne sich selbst in 100 ms wieder neuangesetzt zu haben. Nach der Deklaration der Prozedur muß sie einmal aufgerufen werden, danach läuft sie immer wieder von selbst ab.

Bei flüchtigem Hinsehen könnte man `animate` für eine rekursive Prozedur halten – was sie nicht ist! Im Rumpf wird nicht wieder `animate` aufgerufen, sondern `after`. Dieses Kommando sorgt dafür, das `animate` in die Liste der in 100 ms abzuarbeitenden Kommandos kommt. Danach kehrt `after` zurück und damit wird diese Inkarnation von `animate` abgeschlossen. Etwas später läuft die Zeituhr für die eingetragene Prozedur ab und `animate` wird erneut aufgerufen und erledigt die vorgesehenen Arbeiten, usw.

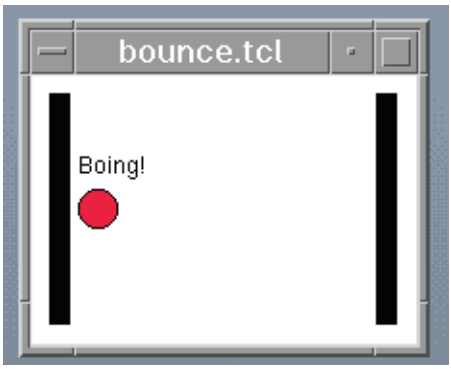
Welches die richtige Zeitspanne für `after` ist, hängt von den Umständen ab. Gibt es größere Berechnungen, muß das `update`-Kommando von Zeit zu Zeit aufgerufen werden.

8.7.1 Animation von Dingen auf der Leinwand

Im folgenden Beispiel lassen wir einen Ball zwischen zwei Wänden springen. Wenn er von der Wand abspringt, zeigen wir eine `Boing!` Botschaft (`bounce.tcl`).

```
pack [canvas .c -width 200 -height 140 \
  -background white]

.c create rectangle 10 10 20 130 -fill black
.c create rectangle 180 10 190 130 -fill black

proc new_ball {xpos ypos speed color} {
  set x0 [expr $xpos-10]
  set x1 [expr $xpos+10]
  set y1 [expr $ypos+20]
  .c create oval $x0 $ypos $x1 $y1 \
    -fill $color -tag "ball-$ypos"
  bounce $xpos $ypos $speed
}

proc bounce {xpos ypos speed} {
  .c move "ball-$ypos" $speed 0
  set xpos [expr $xpos+$speed]
  if {$speed > 0 && $xpos >= 170} {
    set speed [expr -$speed]
    .c create text 175 [expr $ypos-5] -text "Boing!" \
      -anchor se -tag "boing-$ypos"
    after 300 [list .c delete boing-$ypos]
  } elseif {$speed < 0 && $xpos <= 30} {
    set speed [expr -$speed]
    .c create text 25 [expr $ypos-5] -text "Boing!" \
      -anchor sw -tag "boing-$ypos"
    after 300 [list .c delete boing-$ypos]
  }
  after 50 [list bounce $xpos $ypos $speed]
}

new_ball 100 60 5 red
```

Wir erzeugen eine Leinwand und bauen darauf zwei Rechtecke als Wände auf.



Die Prozedur `new_ball` erzeugt einen neuen fliegenden Ball. Die Argumente sind die *x*- und *y*-Koordinaten, die Geschwindigkeit in *x*-Richtung und die Farbe des Balls. Die Prozedur erzeugt ein Oval und ruft danach `bounce` auf, das die eigentliche Animation übernimmt.

Die Prozedur `bounce` hat drei Parameter: die *x*- und *y*-Koordinaten für den Ball und die Geschwindigkeit. Diese gibt an, wieviele Pixel in 50 ms zurückgelegt werden. Ist `speed` positiv, bewegen wir uns nach rechts, sonst nach links. An der Wand angekommen, ändern wir das Vorzeichen. Dort zeigen wir auch unseren `Boing`-Text an, der nach 300 ms wieder verschwinden soll. Am Ende von `bounce` setzen wir den erneuten Aufruf mittels `after` an, wobei die neuen Koordinaten übergeben werden.

8.7.2 Fehlersuche bei after-Kommandos

Hätten wir im obigen Beispiel drei Bälle gleichzeitig in der Luft halten wollen, hätten wir `new_ball` dreimal aufrufen können (`bounce2.tcl`):

```
new_ball 100 60 10 red
new_ball 100 90 5 blue
new_ball 100 30 -3 green
```

Damit wären dann drei separate Ketten von `after`-Ereignissen geschaffen worden, die jeweils das Verhalten eines Balles einer Farbe kontrolliert hätten.

Will man jetzt in einem solchen Programm mit mehreren `after`-Ereignisketten Fehler suchen, kann man `after info` ohne weitere Argumente aufrufen, das einem eine Liste an Tokens mit noch anstehenden Ereignissen liefert.

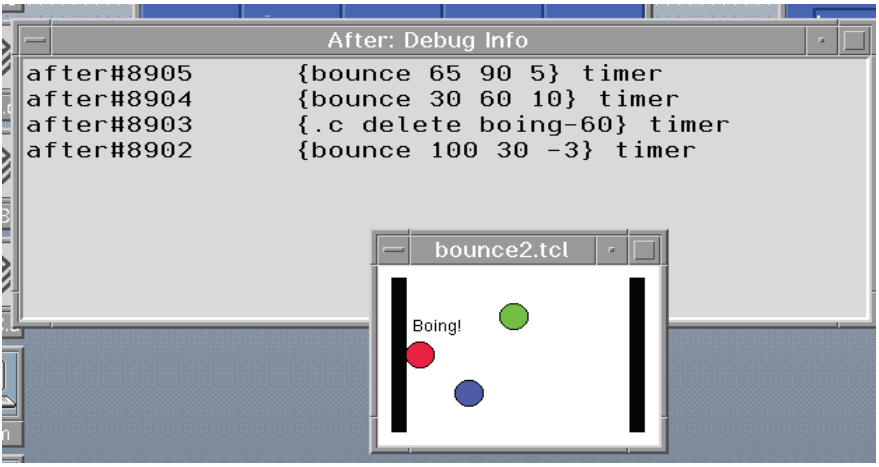
Ruft man `after info` mit einem dieser Tokens auf, erhält man Information zu diesem Ereignis einschließlich dem Kommandoskript in der Aufrufform mit ersetzten Argumenten.

Damit können wir uns ein Beobachtungsfenster anlegen:

```
proc after_debug {} {
  if {![winfo exists .afterdb.t]} {
     toplevel .afterdb
     wm title .afterdb "After: Debug Info"
     text .afterdb.t -width 50 -height 10 -wrap none
     pack .afterdb.t -fill both -expand yes
     after_debug_update
  }
}

proc after_debug_update {} {
  if {[winfo exists .afterdb.t]} {
     .afterdb.t delete 1.0 end
     foreach t [after info] {
        .afterdb.t insert end "$t\t[after info $t]\n"
     }
     after 100 after_debug_update
  }
}
```

In dem Fenster wird `after_debug_update` aufgerufen, das das Fenster leert und dann eine Liste der anstehenden `after`-Events ausgibt. Danach setzt es sich selbst nach 100 ms zum Wiederaufruf an.

Zu beachten ist, daß immer eine Prüfung stattfindet, ob das Fenster noch da ist (der Anwender könnte es geschlossen haben). Wenn nicht, hört `after_debug_update` auf.



Wie man sieht, gibt es immer drei bounce-Ereignisse und gelegentlich ein oder zwei `.c delete` Ereignisse, die das Abräumen des `Boing!`-Texts besorgen.

Übung 8–1

Wenn man in `after_debug` zwei Aufrufe von `after_debug_update` einbauen würde, z. B.

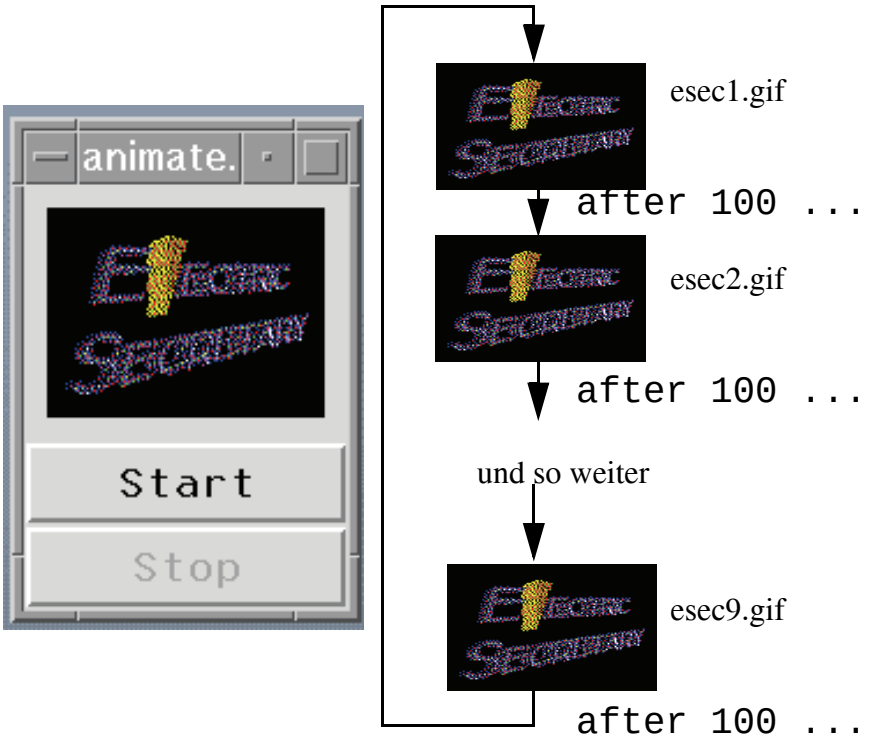
```
after_debug_update
after 50 after_debug_update
```

würde man dann auch diesen Event im Monitorfenster sehen?

8.7.3 Bibliotheksroutinen für die Animation

Animationen treten häufig genug auf, um es lohnenswert erscheinen zu lassen, eigene generische Routinen, `animate_start` und `animate_stop` dafür zu schreiben.

Als Beispiel verwenden wir das amimierte Logo aus der Abbildung unten.



Es besteht aus neun gif-Bildern, die wir zunächst laden (`animate.tcl`).

```
set images {}
set counter 0
for {set i 1} {$i <= 9} {incr i} {
  set file [file join $env(EFFTCL_LIBRARY) images \
    esec$i.gif]
  set imh [image create photo -file $file]
  label .l$i -image $imh
  lappend images $imh
}
```

Diese werden in einer Variablen `images` als Liste von Bildnamen abgespeichert. Zugleich erzeugen wir einen Hilfslabel für jedes Bild. Die Labels packen wir aber nicht, d. h. sie erscheinen nicht. Allerdings wird dadurch das Bild geladen und kann schnell zur Anzeige gebracht werden.

Als nächstes erzeugen wir den Label, der das Bild anzeigt; er beginnt mit dem ersten Bild, wenn wir den Startknopf drücken.

```
label .show -image [lindex $images 0]
pack .show -padx 8 -pady 8
```

Der Startknopf wird wie folgt angelegt.

```
button .start -text "Start" -state normal -command {
  set pending [animate_start 100 $images {
     .show configure -image %v
  }]
  .stop configure -state normal
  .start configure -state disabled
}
pack .start -fill x
```

Beim Anklicken wird die Prozedur `animate_start` ausgelöst. Diese erhält als Argumente einen Zeitverzögerungswert, eine Liste von Bildern und das Kommando

```
.show configure -image %v
```

Die Animationsroutine wird diesen Befehl alle 100 ms ausführen und dabei für `%v` immer ein neues Bild substituieren.

Die Prozedur `animate_start` liefert ein Token zurück, das die Animation identifiziert. Mit diesem Token, das wir in der Variablen `pending` gespeichert haben, können wir die Animation wieder anhalten, wenn wir `animate_stop` über den Stop-Knopf aufrufen.

```
button .stop -text "Stop" -state disabled -command {
  animate_stop $pending
  .start configure -state normal
  .stop configure -state disabled
}
pack .stop -fill x
```

Jetzt müssen wir noch die generischen Prozeduren `animate_stop` und `animate_start` implementieren.

```
set anInfo(counter) 0

proc animate_start {delay vlist command} {
  global anInfo
```

```
    set id "animate[incr anInfo(counter)]"
    set anInfo($id-delay) $delay
    set anInfo($id-vlist) $vlist
    set anInfo($id-command) $command
    set anInfo($id-pos) 0
    set anInfo($id-pending) \
      [after $delay "animate_handle $id"]

    return $id
  }
```

Zunächst wird das Token, das die Animationsfolge identifiziert, geschaffen, indem eine Zahl aus dem `anInfo`-Zähler an das Wort `animate` angehangen wird.

Unter diesem Schlüssel wird die Zeitverzögerung, die Bildliste, das Kommando, die Nummer des nächsten Bilds (`pos`, hier 0 für das Startbild) und das Token für das anstehende `after`-Ereignis abgespeichert.

Nach Einrichten dieser Werte kann die folgende Prozedur in den festgelegten Zeitintervallen aufgerufen werden.

```
proc animate_handle {id} {
  global anInfo

  if {[info exists anInfo($id-pending)]} {
    set pos $anInfo($id-pos)
    set val [lindex $anInfo($id-vlist) $pos]
    set cmd [percent_subst %v \
      $anInfo($id-command) $val]
    uplevel #0 $cmd

    if {[incr pos] >= [llength $anInfo($id-vlist)]} {
      set pos 0
    }
    set anInfo($id-pos) $pos
    set anInfo($id-pending) [after $anInfo($id-delay) \
      "animate_handle $id"]
  }
}
```

Die Routine verwendet wieder die geliebte Prozentsubstitution, die in H+McL im Abschnitt 7.6.7.3 beschrieben wird. Ferner wird das Kommando, also die eigentliche Anzeige, im globalen Kontext (`uplevel`

#0) ausgeführt, da es Referenzen auf Variablen enthalten könnte, die nur dort definiert sind.

Zuletzt wird in der Routine die nächste Anzeige angemeldet und der zurückgelieferte Token in der pending-Komponente des Records eingetragen. Über diesen Identifier können wir die Animation abbrechen. Dies passiert, wenn wir `animate_stop` via den Stop-Knopf aufrufen.

```
proc animate_stop {id} {
  global anInfo

  if {[info exists anInfo($id-pending)]} {
     after cancel $anInfo($id-pending)

     unset anInfo($id-delay)
     unset anInfo($id-vlist)
     unset anInfo($id-command)
     unset anInfo($id-pos)
     unset anInfo($id-pending)
  }
}
```

Dies beendet die Animationsaufrufe. Zugleich sind wir auch hier mit der Behandlung von Animationen fertig, wobei H+McL darauf hinweisen, daß obige Bibliotheksroutinen später für weitere Anwendungen, z. B. „schimmernde Auswahlrahmen“, blinkende Texte usw. verwendet werden.

9 Der Gebrauch der Leinwand

Die Autoren Harrison und McLennan weisen zu Recht darauf hin, daß eine interaktive graphische Ausgabe meist aussagekräftiger ist, als eine lange Liste von Meldungen (Informatikerspruch: ein Bild ersetzt 1024 Worte). H+McL erwähnen einen Maschinensaal, bei dem Probleme in einzelnen Bereichen durch blinkende rote Flächen in der graphischen Darstellung der Maschinenplans angezeigt werden.

Hinweis: Man sollte aber zwischen der Information unterscheiden, die die Existenz eines Faktums anzeigt (es gibt einen Zug von KS nach Frankfurt) und der Instanzinformation (es ist ein ICE, Abfahrt 6:12 Uhr, Gleis 2, verkehrt nicht ...). HCI-Gurus reiten gern auf ersterem rum und ignorieren das echte Problem der zweiten Art.

In jedem Fall ist aber das Leinwandwidget das geeignete Werkzeug, um nahezu beliebig gestaltete graphische Information darzustellen. Als Beispiele werden im Kapitel 4 des Buches von H+McL genannt und angekündigt:

- eine Fortschrittsanzeige
- ein Farbrad
- ein Notizbuch mit Griffleiste
- ein Kalender
- ein Zeicheneditor

9.1 Das Leinwandwidget verstehen

Eine leere Leinwand ohne Standardverhalten, hier zwei Zoll breit und einen hoch, erzeugt man mit

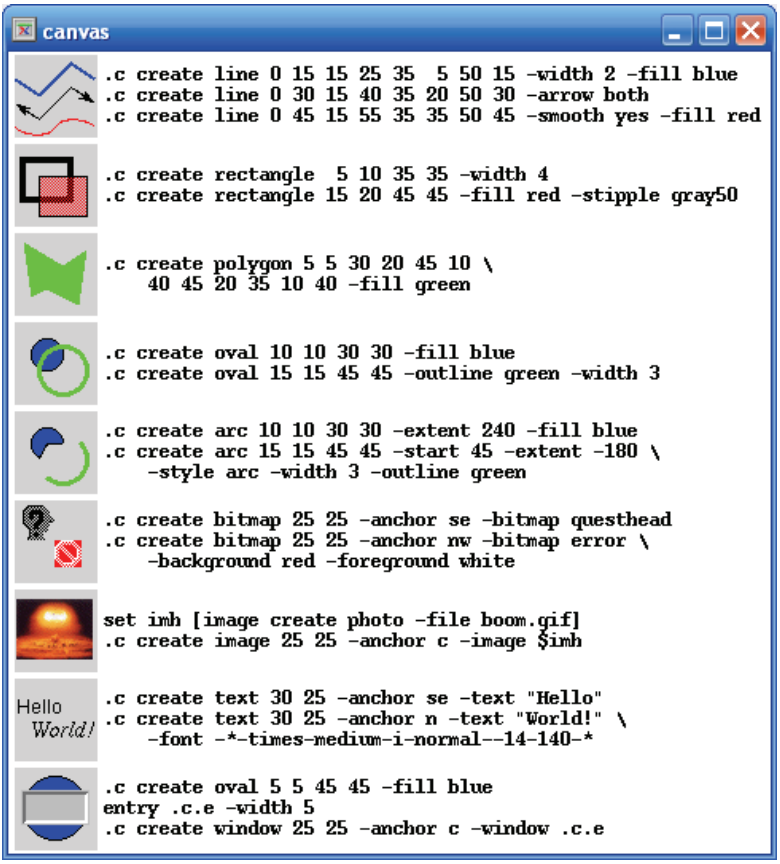
```
canvas .c -width 2i -height 1i
```

Auf dieser leeren Leinwand lassen sich sog. *items* unterbringen, z. B. eine Linie mittels

```
.c create line 0 15 15 25 35 5 50 15 -width 2 -fill blue
```

Jedes Zahlenpärchen gibt eine (x, y)-Koordinate an. Ohne Buchstabenmodifikation und als ganze Zahl bezeichnen die Werte Pixelkoordinaten, die links oben mit (0, 0) starten.

Jedes Item hat ein Standardaussehen. Linien sind z. B. schwarz, im Beispiel oben aber als blau und zwei Pixel stark neudefiniert.



Die obige Abbildung (Programm `items.tcl`) zeigt einige Items, die Details muß man dem Handbuch entnehmen. Eine Aufzählung der wesentlichen Arten folgt.

- **line**
  Linie mit zwei oder mehr Koordinaten. Pfeilenden können angebracht werden. Ist `smooth` angegeben, wird die Linie als Bézierkurve gezeichnet.
- **rectangle**
  Ein Rechteck hat zwei Koordinaten, die gegenüberliegende Ecken markieren. Man kann eine Umrandungsfarbe und ~stärke sowie eine separate Füllfarbe angeben.
- **polygon**
  Ein Polygon (geschlossener Kantenzug) hat drei oder mehr Punkte. Wie beim Rechteck hat es Linienfarbe, ~stärke und Füllfarbe für das Innere. Wie bei der Linie kann man einen Kurvenzug daraus machen.
- **oval**
  Oval mit zwei Koordinaten, die dem Rechteck entsprechen, das es umhüllt. Wie beim Rechteck ...
- **arc**
  Wie ein Oval aber mit einem Startwinkel und einer Ausdehnung (`extend`), der angibt, wieviel des Ovals gezeichnet werden. Der Bogen kann als Bogenlinie, Kuchenstück oder Schnur (`chord`) zwischen den Endpunkten erscheinen.
- **bitmap**
  Eine Pixelgraphik hat eine Koordinate, die seinen Ankerpunkt angibt. Sie wird gemäß der `-anchor` Option dort angelegt. Sie hat zwei Farben: Vordergrund (Standard schwarz) und Hintergrund (Leinwandfarbe).
- **image**
  Ebenfalls mit einem Ankerpunkt, aber anders als bitmap mit beliebiger Anzahl von Farben, die auf Monochrombildschirmen in Graustufen verwandelt werden (dithering).

- **text**
  Ein Textitem hat eine Koordinate für den Ankerpunkt. Ein einzelnes Textitem kann aus mehreren Zeilen bestehen, und man kann die Ausrichtung bestimmen. Ferner können Farbe und Font angegeben werden.

- **window**
  Ein Fenster hat einen Ankerpunkt. Es dient als Platzhalter für ein Widget, das in der Leinwand eingebettet wird und ggf. über anderen Items liegt.

Viele Items erlauben als Option ein Füllmuster (`-stipple`). Die Vorgabe ist eine dichte 100%-Füllung. Mit einem Bitmuster wie etwa `gray50` ergibt sich ein durchbrochener Effekt, bei dem die schwarzen Bitmap-Punkte in der Füllfarbe erscheinen, die anderen durchsichtig (vgl. das Rechteck oben mit der durchscheinenden Linie des Rechtecks darunter).

9.1.1 Rollen

Jede Leinwand hat eine unendliche Ausdehnung. Das Leinwandwidget kann aber nur einen bestimmten Ausschnitt zeigen, startend mit der Koordinate (0, 0) in der linken oberen Ecke. Der sichtbare Bereich heißt *viewport* (Guckloch, Sehschlitz). Man kann Dinge außerhalb des viewports anlegen, aber man kann sie nicht sehen, sofern man nicht den sichtbaren Ausschnitt dorthin bewegt.

Hierzu kann man Rollbalken an dem Fenster anbringen. Wir hatten dies im Abschnitt 7.2.1 bereits gesehen.

```
canvas .display -width 3i -height 3i -background black \
  -xscrollcommand {.xsbar set} \
  -yscrollcommand {.ysbar set}
scrollbar .xsbar -orient horizontal \
  -command {.display xview}
scrollbar .ysbar -orient vertical \
  -command {.display yview}
.display create line 98.0 298.0 98.0 83.0 \
  -fill green -width 2
.display create line 98.0 83.0 101.0 69.0 \
```

```
    -fill green -width 2
  .display create line 101.0 69.0 108.0 56.0 \
    -fill green -width 2
  ...
```

Jeder Rollbalken hat eine `-command` Option mit einem Kommando, das die Leinwandsicht beeinflußt. In unserem Fall bewirkt das Ziehen am horizontalen Balken die Ausführung des Kommandos `.display xview` mit einigen weiteren Argumenten. Analog geht dies für den vertikalen Rollbalken.

Die Leinwand selbst hat ähnliche `-xscrollcommand` und `-yscrollcommand` Optionen, die wiederum das Aussehen der Rollbalken bestimmen (wo die Griffe stehen und wie lang sie sind). In unserem Beispiel bewirken Änderungen die Ausführung von `.xsbar set`, bzw. `.ysbar set` mit passenden Argumenten. Damit können Leinwand und Rollbalken sich gegenseitig synchronisieren.

Allerdings rollt dadurch die Leinwand immer noch nicht richtig. Man muß der Leinwand noch die *Rollregion* (`-scrollregion`) angeben, womit man die Grenzen des Bereichs bestimmt, innerhalb dessen der Viewport sich bewegen kann. Wird nichts angegeben, ist der Bereich genauso groß wie der Viewport. Das bedeutet, daß ohne gesonderte Einstellung, Tk auch bei Leinwänden, die sehr viel größer sind als der sichtbare Bereich, zunächst nicht rollen will.

Deshalb muß man zunächst die Ausdehnung der Leinwand angeben. Dies könnte z. B. durch

```
.display configure -scrollregion {0 0 250 375}
```

geschehen. Innerhalb dieses rechteckigen Bereichs kann das Guckloch bewegt werden.

Oft kennt man die Größe nicht. Macht man den Rollbereich zu klein, kann man nicht alles sehen. Macht man ihn zu groß, sieht man viel leere Fläche.

Ein Trick besteht darin, sich von Tk ein umhüllendes Rechteck (*bounding box*, bbox) für alle Leinwanditems berechnen zu lassen und dessen Ausdehnung dann in `-scrollregion` einzugeben.

```
. display configure -scrollregion [.display bbox all] \
  -xscrollincrement 0.1i -yscrollincrement 0.1i
```

Im Beispiel oben wird das gezeigt, wobei zu bedenken ist, daß dies nach Erzeugung aller Items zu geschehen hat und wiederholt werden muß, wenn durch Einfügen oder Entfernen von Items sich die Größenverhältnisse geändert haben.

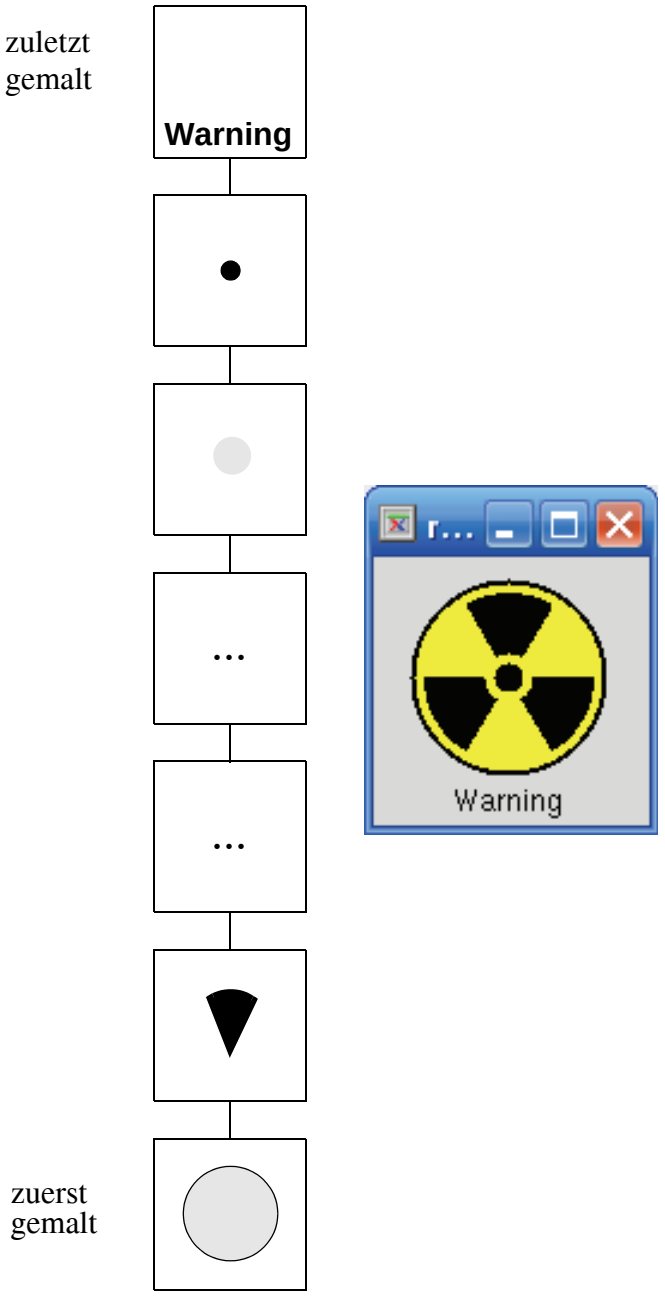
Wie man am Beispiel auch sieht, gibt es `-xscrollincrement` (`-y`...) Optionen um durch Anklicken der Pfeilspitzen das Sichtfenster schrittweise, hier z. B. jeweils um 0,1 Zoll, zu bewegen.

9.1.2 Die Anzeigeliste

Die Leinwand merkt sich alle Items, die erzeugt wurden. Zu jeder Zeit kann man Dinge darauf bewegen, die Items neu anlegen oder vergrößern. Die Zeichnung wird dann automatisch neu erzeugt. Im Beispiel haben wir den folgenden Code (`rad1.tcl`).

```
canvas .c -width 100 -height 110
pack .c

.c create oval 10 10 90 90 -fill yellow -width 2
.c create arc 15 15 85 85 -start 60 -extent 60 \
  -fill black
.c create arc 15 15 85 85 -start 180 -extent 60 \
  -fill black
.c create arc 15 15 85 85 -start 300 -extent 60 \
  -fill black
.c create oval 40 40 60 60 -outline "" -fill yellow
.c create oval 44 44 56 56 -outline "" -fill black
.c create text 50 95 -anchor n -text "Warning"
```

Wenn jedes der Items erzeugt wird, kommt es in eine interne Liste, die man *Display List* (Anzeigeliste) nennt. Muß neu gemalt werden, wird die Liste von unten nach oben neu zur Anzeige gebracht.


zuletzt
gemalt
Warning
...
...
zuerst
gemalt
r...
Warning

Nehmen wir an, der Reaktor wird kritisch und wir ändern die Botschaft von „Warning“ in „RED ALERT“. Dazu müssen wir die Elemente der Leinwand benennen können.

Wird ein Item erzeugt, vergibt die Leinwand dafür eine eindeutige Nummer, den *Itemidentifikator*. Diesen kann man festhalten, wenn das Item erzeugt wird:

```
set id [.c create text 50 95 -anchor n -text "Warning"]
```

Später können wir mit diesem Wert das Item rekonfigurieren:

```
.c itemconfigure $id -text "RED ALERT"
```

Bei diesen Änderungen stellt die Leinwand fest, welche Teile betroffen sind und besorgt das partielle Neumalen sehr effizient, so daß Änderungen sofort angezeigt werden.

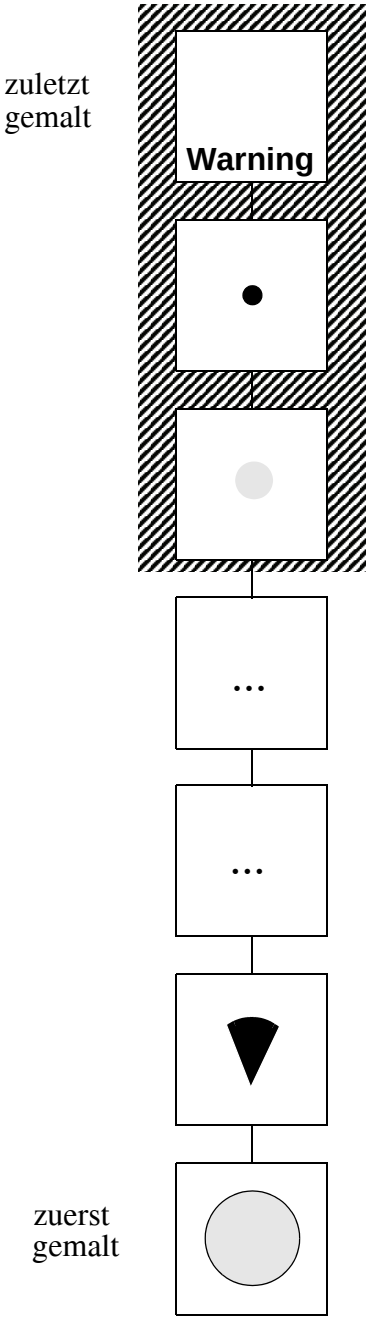
Die Leinwand besitzt auch Suchfunktionen, mit denen sich z. B. alle Items auswählen lassen, die innerhalb eines umschließenden Rechtecks liegen:

```
.c find enclosed 20 20 80 110
```

Wir könnten damit alle Items im Rechteck (20, 20) bis (80, 110) auf rote Farbe umschalten (Programm `rad2.tcl`).

```
foreach id [.c find enclosed 20 20 80 110] {
  .c itemconfigure $id -fill red
}
```

Neben `find enclosed` gibt es noch weitere Suchoperationen, die zum Teil aufeinander aufbauen.


zuletzt
gemalt
Warning
...
...
zuerst
gemalt


r...
Warning


r...
Warning

9.1.3 Der Gebrauch von Marken (tags)

Itemidentifikatoren sind nützlich, aber nicht sehr einprägsam. Leinwandprogramme werden übersichtlicher, wenn wir den Items symbolische Bezeichner mit der `-tags` Option zuordnen, wie im folgenden Beispiel (`rad3.tcl`).

```
canvas .c -width 100 -height 110
pack .c

.c create oval 10 10 90 90 -fill yellow -width 2
.c create arc 15 15 85 85 -start 60 -extent 60 \
  -fill black -tags sign
.c create arc 15 15 85 85 -start 180 -extent 60 \
  -fill black -tags sign
.c create arc 15 15 85 85 -start 300 -extent 60 \
  -fill black -tags sign
.c create oval 40 40 60 60 -outline "" -fill yellow
.c create oval 44 44 56 56 -outline "" \
  -fill black -tags sign
.c create text 50 95 -anchor n -text "Warning" \
  -tags message
```

Die Warnbotschaft hat jetzt den symbolischen Namen `message`. Bei Änderungen können wir über ihn auf das Item zugreifen:

```
.c itemconfigure message -text "RED ALERT" -fill red
```

Das ist klarer als den Itemdeskriptor in einer Variablen abzuspeichern und sich diese zu merken.

Genauso kann ein Tag einer Gruppe von Items zugewiesen werden. Im Beispiel geschah dies für die schwarzen Teile des Strahlungssymbols, die wir so auf einen Schlag rot färben können:

```
.c itemconfigure sign -fill red
```

H+McL weisen darauf hin, daß diese Operationen, welche die Leinwand betreffen, in übersetztem Code vorliegen, was ihre Ausführung viel schneller macht als ein zu interpretierendes Tcl-Skript, das z. B. durch eine Liste von Itemidentifikatoren iteriert.

Ein Item kann auch mehrere Marken (tags) haben und damit zu mehreren Gruppen von Items gehören. Beispielsweise könnte man eine Gruppe `hilite` bilden, deren Items alle gleichzeitig auf rot gesetzt werden. Dazu gehören sollen das `message`-Item und die `sign`-Items. Zusammen werden sie mit

```
.c itemconfigure hilite -fill red
```

umgefärbt.

Das verlangt, daß wir den Tagnamen `hilite` allen Items hinzufügen, die wir in der Gruppe haben wollen. Eine Möglichkeit, die den Gebrauch von Listen für Tags zeigt, wäre

```
.c create text 50 95 -anchor n -text "Warning" \
   -tags {message hilite}
```

Eine andere Möglichkeit wäre `addtag`, mit dem sich alle Items finden lassen, die z. B. die Marke `sign` haben und denen dann zusätzlich die Marke `hilite` eingetragen wird.

```
.c addtag "hilite" withtag "sign"
```

Damit markiert `hilite` jetzt neben dem Text „Warning“ auch die vier Graphikteile, die das Strahlungssymbol ausmachen.

9.1.4 Leinwandverhalten

Über die gerade besprochenen Marken läßt sich das Verhalten von Leinwanditems steuern. Zunächst noch ganz konventionell, z. B. wenn wir an das Leinwandwidget `.c` und das `<Enter>`-Ereignis ein Kommando binden (`rad4.tcl`).

```
bind .c <Enter> {
  .c itemconfigure hilite -fill red
}
bind .c <Leave> {
  .c itemconfigure hilite -fill black
}
```

Jetzt wechseln der Warntext und das Strahlensymbol, die unter der Marke `hilite` angesprochen werden können, beim Betreten des Fensters ihre Farbe von schwarz in rot.

Man kann aber noch weiter gehen, und Bindings direkt an Items knüpfen, so daß etwa nur beim Überstreichen der Items die Farbe wechselt.

```
.c bind hilite <Enter> {
  .c itemconfigure hilite -fill red
}
.c bind hilite <Leave> {
  .c itemconfigure hilite -fill black
}
```

Wir binden das Ereignis und die Reaktion an das `canvas`-Widget, das wiederum Ereignis und Reaktion an die in `bind` angegebenen Items bindet.

H+McL betonen, daß diese Ereignisbehandlung separat von der übrigen abläuft, d. h. sie wird nicht von den Bindemarkenordnung, wie oben diskutiert, berührt und auch ein `break` hat keinen Einfluß auf die Weitergabe innerhalb der Widget-Hierarchie.

```
.c bind message <ButtonPress-1> {
  if {[.c itemcget message -text] == "Warning"} {
     .c itemconfigure message -text "RED ALERT"
  } else {
     .c itemconfigure message -text "Warning"
  }
}
```

Ganz analog haben wir oben an das Textitem einen Mausklick gebunden, der das Hin- und Herschalten zwischen den zwei Warntexten ermöglicht (`rad5.tcl`).

Eine weitere raffinierte Möglichkeit besteht darin, eine Reaktion in Abhängigkeit vom aktuellen überstrichenen Item hervorzurufen. Dazu erhält das Item, das gerade von der Maus berührt wird, temporär die Marke `current`. Ändert es etwa daraufhin seine Farbe, Kontur oder Linienform, erhält der Anwender eine sofortige Rückmeldung über die Position. H+McL nennen diese Technik *brushing* (Anmerkung: in der Computergraphik bezeichnet man mit der brush-Technik das Malen gan-

zer Bildschirmbereiche mit einem Pixelmuster – ähnlich im Aussehen, aber eigentlich von der Intention etwas anderes).

```
.c bind hilite <Enter> {
  .c itemconfigure current -fill red
}
.c bind hilite <Leave> {
  .c itemconfigure current -fill black
}
```

Hier wird jetzt nur jedes einzelne Item (das in der `hilite`-Gruppe ist) in der Farbe geändert, wenn die Maus es überstreicht (`rad6.tcl`).

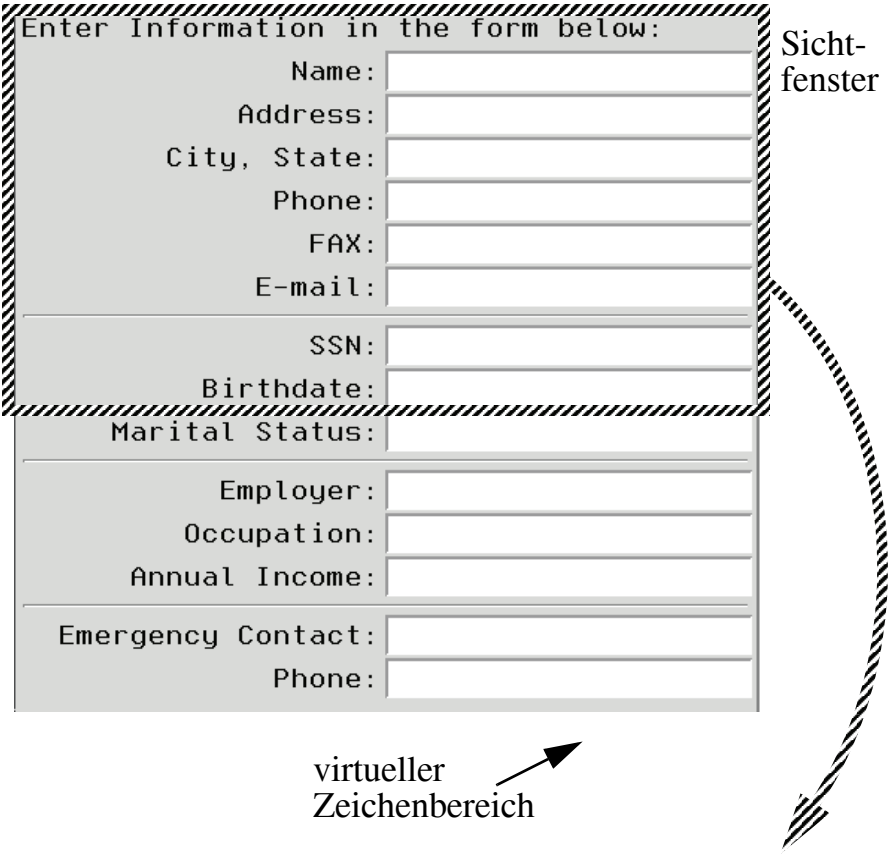
H+McL betonen, daß Tags die Schlüssel zum Canvas-Widget sind, weil sich über sie aktive Bereiche auf der Leinwand markieren lassen und geschlossene Reaktionen für ganze Untergruppen herstellbar sind.

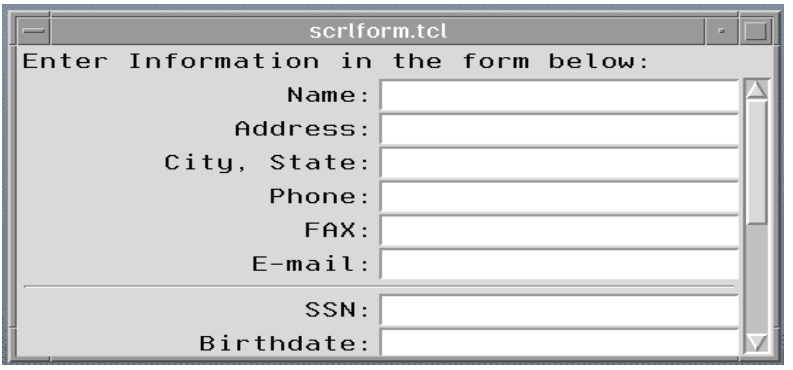
9.2 Rollbare Formulare

Wenn eine Leinwand zu klein ist, um eine ganze Graphik auf einmal darzustellen, sieht man nur einen kleineren Ausschnitt, der mit Rollbalken über die Graphik bewegt werden kann. Das funktioniert auch für Widgets, die in die Leinwand eingelassen sind, solange die Leinwand die Größe des Rollbereich kennt.

Demnach lassen sich Widgets und Graphiken mischen, wodurch anspruchsvolle Anzeigen gestaltet werden können. Im Beispiel hier geht es jedoch einfach nur um ein besonders langes Formular, durch das wir hindurchblättern wollen.

Wir gehen die Sache so an, daß wir mehrere kleinere Widgets mittels `pack` oder `grid` in einen Frame plazieren, den wir dann auf der Leinwand ablegen. Diese wird dann Rollbalken haben und schon ist das rollbare Formular fertig (`scrlform.tcl`).


Enter Information in the form below:
Name:
Address:
City, State:
Phone:
FAX:
E-mail:
SSN:
Birthdate:
Marital Status:
Employer:
Occupation:
Annual Income:
Emergency Contact:
Phone:
Sicht-
fenster
virtueller
Zeichenbereich


scrlform.tcl
Enter Information in the form below:
Name:
Address:
City, State:
Phone:
FAX:
E-mail:
SSN:
Birthdate:

Wir fangen mit einer generischen Routine an, die eine Leinwand erzeugt, diese mit Rollbalken versieht, und in der Leinwand einen leeren Rahmen (frame) ablegt.

```
proc scrollform_create {win} {

  frame $win -class Scrollform

  scrollbar $win.sbar -command "$win.vport yview"
  pack $win.sbar -side right -fill y

  canvas $win.vport -yscrollcommand "$win.sbar set"
  pack $win.vport -side left -fill both -expand true

  frame $win.vport.form
  $win.vport create window 0 0 -anchor nw -window \
    $win.vport.form

  bind $win.vport.form <Configure> \
    "scrollform_resize $win"

  return $win
}
```

Die Zusammensetzung erscheint zunächst verwirrend.

Ganz außen haben wir einen Rahmen, der die Leinwand und die Rollbalken umfaßt. Sein Name ist Argument der `scrollform_create` Prozedur. In dem Rahmen legen wir Leinwand und Balken an und verbinden sie, wie in 9.1.1 besprochen.

Dann legen wir einen zunächst leeren Rahmen an, der später die Labels und Entry-Widgets enthalten soll, die zum Formular gehören. Wichtig ist sein Name – `$win.vport.form` – der ihn als Unterwidget der Leinwand `$win.vport` ausweist. Damit sitzt er **in** der Leinwand und nicht – wie für `$win.form` – gleichberechtigt **neben** der Leinwand, die er samt Rollbalken ggf. überdecken könnte.

Den Rahmen legen wir in die obere Ecke, indem wir ihm ein Fensteritem auf der Leinwand erzeugen.

Noch haben wir nicht die Größe des Rollbereichs angegeben und in der Tat können wir das erst, wenn wir die Einträge in dem zunächst noch leeren Rahmen plaziert haben. Deshalb setzen wir die Größe nicht direkt, sondern binden die Zuweisung an das `<Configure>`-Ereignis. Die folgende Routine setzt die Rollbereichsgröße und wird aufgerufen, wenn sich die Größe des Formulars geändert hat.

```
proc scrollform_resize {win} {
  set bbox [$win.vport bbox all]
  set wid [winfo width $win.vport.form]
  $win.vport configure -width $wid \
     -scrollregion $bbox -yscrollincrement 0.1i
}
```

Die Routine ermittelt die Größe des Formulars durch die `bbox`-Leinwandoperation und diese Größe erhält die Leinwand später im `configure`-Befehl, um den Bereich festzulegen, in dem der Viewport sich bewegen darf. Die Breite der Leinwand setzen wir als festen Wert, ermittelt aus der Breite des Formulars. Hier sehen wir keine Rollbalken vor.

Jetzt ist das Rollformular gebrauchsfertig, hat aber natürlich noch keine Einträge. Diese werden in einer Schleife „automatisch“ nach gegebenen Eintragsnamen erzeugt, wobei der spezielle Name des Rahmens (`$win.vport.form`) hinter einer Namensmethode `scrollform_interior` versteckt wird.

```
proc scrollform_interior {win} {
  return "$win.vport.form"
}
```

Ganz oben hin kommt noch ein Text, darunter packen wir den zunächst leeren Rahmen wie oben besprochen.

```
label .title \
  -text "Enter Information in the form below:"
pack .title -anchor w

scrollform_create .sform
pack .sform -expand yes -fill both
```

Dann werden die kleineren Widgets in den Formularrahmen eingefügt, wozu auch Trennlinien (Feld „-“)gehören. Das Füllen beginnt mit der Ermittlung des Rahmennamens.

```
set form [scrollform_interior .sform]
```

Danach kommt die Schleife, wobei Widgetnamen, hier `line1`, `line2`, usw., automatisch erzeugt werden müssen. Diese werden intern dann durch das Präfix `$form`, z. B. zu `.sform.vport.form.line17`, erweitert und können noch weitere Verlängerungen für die Marke und das Eintragefeld (`label` bzw. `entry`) erhalten.

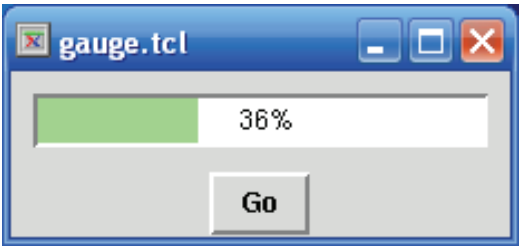
```
set counter 0
foreach field {
  "Name:" "Address:" "City, State:"
  "Phone:" "FAX:" "E-mail:"
  "-"
  "SSN:" "Birthdate:" "Marital Status:"
  "-"
  "Employer:" "Occupation:" "Annual Income:"
  "-"
  "Emergency Contact:" "Phone:"
} {
  set line "$form.line[incr counter]"
  if {$field == "-"} {
     frame $line -height 2 -borderwidth 1 -relief sunken
     pack $line -fill x -padx 4 -pady 4
  } else {
     frame $line
     label $line.label -text $field -width 20 -anchor e
     pack $line.label -side left
     entry $line.info
     pack $line.info -fill x
     pack $line -side top -fill x
  }
}
```

Ist der Rahmen komplett, wird sich der Formularrahmen dicht darumlegen und seine Größenveränderung wird den `<Configure>`-Event auslösen, der `scrollform_resize` aufruft. Das passiert auch, falls wir später weitere Zeilen in das Formular einfügen.

Anmerkung: Das Beispiel ist zwar ein rollbares Formular, aber kein „blätterbares“. Es reagiert nicht auf die „Bild-hoch“/„Bild-runter“ (PageUp/PageDown)-Tasten. Zusätzlich würde man sich dann auch eine Anzeige „Seite 3 von 7“ wünschen, ggf. spezielle „nächste Seite“/„Seite zurück“-Knöpfe am Fenster.

9.3 Eine Fortschrittsanzeige

Eine Fortschrittsanzeige (process gauge) soll verdeutlichen, wie man partiell Anzeigeteile verändert und andere intakt läßt. Man kann sie verwenden, wenn Web-Seiten geladen werden oder der Fortschritt rechenintensiver Abläufe signalisiert werden soll.



Dazu legen wir eine Leinwand an mit zwei Items: ein Textitem für die Prozentzahl (zwischen 0 und 100), versehen mit Marke `value`, und ein Rechteck für den Balken mit Marke `bar`.

Für die Erzeugung rufen wir `gauge_create` mit einem Namen für das ganze Gebilde und einer optionalen Farbe für den Balken auf.

```
proc gauge_create {win {color ""}} {
  frame $win -class Gauge

  set len [option get $win length Length]

  canvas $win.display -borderwidth 0 -background white \
     -highlightthickness 0 -width $len -height 20
  pack $win.display -expand yes

  if {$color == ""} {
     set color [option get $win color Color]
  }
  $win.display create rectangle 0 0 0 20 \
```

```
      -outline "" -fill $color -tags bar
    $win.display create text [expr 0.5*$len] 10 \
      -anchor c -text "0%" -tags value

    return $win
  }
```

Dem äußeren Rahmen, der als Hülle dient, geben wir den Klassennamen `Gauge`, damit wir Standardvorgaben für die Klasse einsetzen können. Im Beispiel sind dies

```
option add *Gauge.borderWidth 2 widgetDefault
option add *Gauge.relief sunken widgetDefault
option add *Gauge.length 200 widgetDefault
option add *Gauge.color gray widgetDefault
```

Man beachte, daß `borderWidth` und `relief` Standardoptionen für Rahmen sind, `length` und `color` aber speziell für das Meßgerät (`Gauge`) eingeführt wurden.

Diese Vorgabe fragen wir auch ab, wenn wir die Länge der Leinwand `$win.display` setzen. Deren Höhe kodieren wir hart mit 20 Pixeln, den Rand schalten wir aus.

Die Farbe fragen wir nur ab, wenn keine Vorgabe aus dem Aufruf vorliegt. Auch der Balken bekommt keine Randlinie und ist zunächst ein „Strich“ der Breite 0 und Höhe 20. Den Prozentwert setzen wir in die Mitte und lassen ihn `0%` anzeigen.

Um jetzt Meßwerte anzuzeigen, übergeben wir einer generischen Routine den Namen der Anzeige und den aktuellen Wert.

```
proc gauge_value {win val} {
  if {$val < 0 || $val > 100} {
    error "bad value '$val': should be 0-100"
  }
  set msg [format "%3.0f%%" $val]
  $win.display itemconfigure value -text $msg

  set w [expr 0.01*$val*[winfo width $win.display]]
  set h [winfo height $win.display]
  $win.display coords bar 0 0 $w $h
```

```
    update
  }
```

Beachtenswert ist die Wandlung der Gleitkommazahl `$val` in eine ganze Zahl mit höchstens drei Stellen (`format "%3.0f"`). Delikat ist auch das doppelte %%, damit ein % am Bildschirm erscheint. Die neue Längenberechnung ist offensichtlich, die eigentliche Änderung macht die Leinwandoperation `coords`. Mit `update` müssen wir dafür sorgen, daß die Änderungen auch wirklich sofort angezeigt werden.

```
gauge_create .g PaleGreen
pack .g -expand yes -fill both -padx 10 -pady 10

button .go -text "Go" -command {
  for {set i 0} {$i <= 100} {incr i} {
    after 100
    gauge_value .g $i
  }
}
pack .go
```

Zum Testen rufen wir dann nach dem Anlegen der Anzeige die Wertübermittlung hundertmal auf (`gauge.tcl`). In einer wirklichen Anwendung wäre das `after`-Kommando durch wirklichen Code zu ersetzen, der von Zeit zu Zeit einen Meßwert über seinen Fortschritt abliefert.

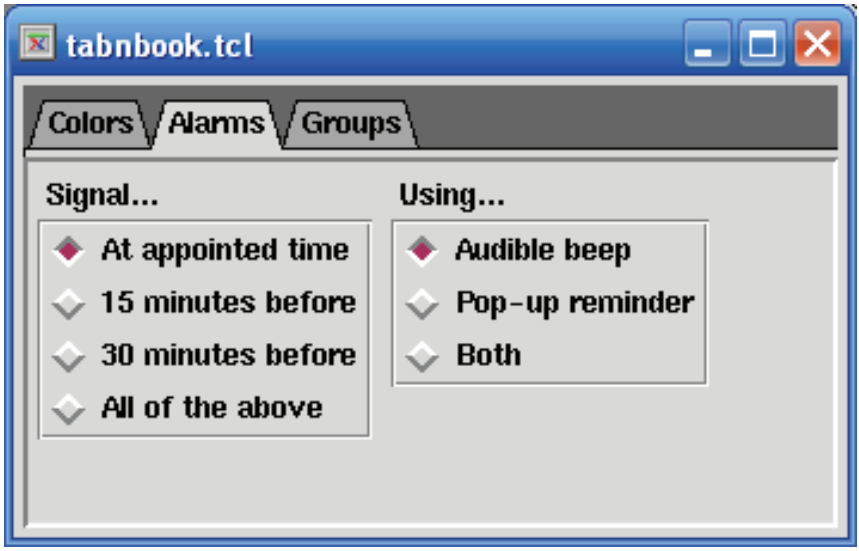
9.4 Ein HSB Farbeditor

In H+McL folgt jetzt ein Programm (`clrdial.tcl`) für einen Farbeditor, mit dem sich Farben einstellen lassen und zwar deren Farbton (*hue*), Sättigung (*saturation*) und Helligkeit (*brightness*).

Das Beispiel zeigt Techniken, wie sich bestimmte Aktionen an Items einer Leinwand und deren Bewegungen binden lassen. Wegen der Umrechnungen in einen Farbwert ist das Programm technisch aufwendig. Wir übergehen es hier.

9.5 Ein Notizbuch mit Griffleiste

Im Abschnitt 7.1.7 wurde ein Notizbuch eingeführt, daß einzelne Seiten zur Anzeige brachte. H+McL bauen darauf auf und fügen oben eine auf der Leinwand „gemalte“ Griffleiste (Liste von Reitern) an, um ein sog. *tabbed notebook* zu produzieren (`tabnbook.tcl`).



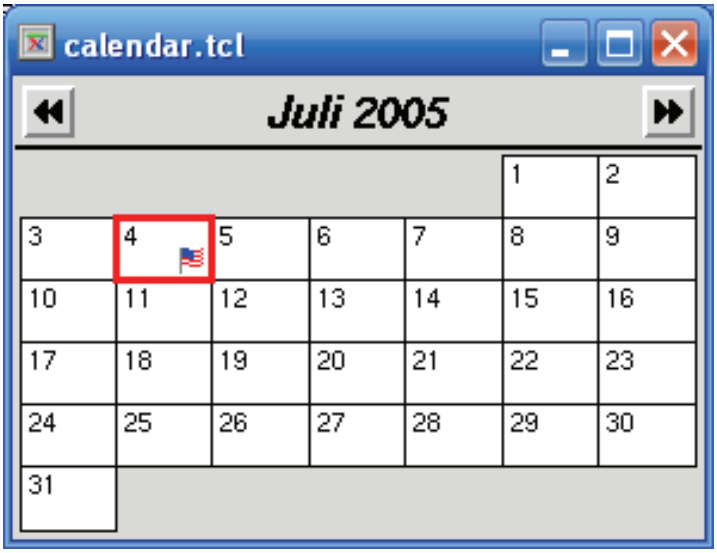
Die Schwierigkeit hier liegt in der variablen Länge der Texte für die Reiter, die von links nach rechts gemalt werden. Die dazu benötigten Polygone müssen ihre Koordinaten also zur Laufzeit bestimmen. Weitere Punkte betreffen das Umschalten der Farben für die gerade gewählte Seite.

Wir übergehen auch dieses Beispiel.

9.6 Ein Kalender

Mit diesem Beispiel sollen Größenveränderungen von Leinwänden behandelt werden. Eingeführt werden auch neue Techniken für die Auswahl einzelner Items und das Auffrischen komplexer Anzeigen.

Der Kalender zeigt jeweils einen Monat an. Für jeden Tag gibt es ein Rechteck, das den Tag im Monat links oben enthält und auch ein Bilditem vorsieht, um Feiertage oder Geburtstage zu markieren (`calendar.tcl`).



Oberhalb der Tagematrix wird der Monat als Textitem angezeigt zusammen mit zwei Window-Items für Knöpfe, damit man von Monat zu Monat blättern kann.

9.6.1 Größenänderungen behandeln

Wie üblich schreiben wir eine Prozedur, genannt `calendar_create`, um den Kalender anzulegen. Sie wird aufgerufen mit z. B.

```
calendar_create .cal 7/4/98
pack .cal -expand yes -fill both
```

d. h. wir erzeugen einen Kalender `.cal` und geben ein optionales Datum mit, das den Monat Juli auswählt. Eine alternative Datumsangabe wäre `now`, die dazu führt, daß das Programm sich den Monat über die Systemzeit (`clock seconds`) holt.

Wichtig ist, daß wir `.cal` mit `pack`-Optionen `-expand` und `-fill` zur Anzeige gebracht haben. Dadurch wird auch die Leinwand gestreckt, wenn das Fenster sich vergrößert. Dies müssen wir durch Neuzeichnen des Kalenders unterstützen.

```
proc calendar_create {win {date "now"}} {
  global calInfo env

  if {$date == "now"} {
     set time [clock seconds]
  } else {
     set time [clock scan $date]
  }
  set calInfo($win-time) $time
  set calInfo($win-selected) ""
  set calInfo($win-selectCmd) ""
  set calInfo($win-decorateVar) ""

  frame $win -class Calendar
  canvas $win.cal -width 3i -height 2i
  pack $win.cal -expand yes -fill both

  button $win.cal.back \
     -bitmap @[file join $env(EFFTCL_LIBRARY) \
     images back.xbm] -command "calendar_change $win -1"

  button $win.cal.fwd \
     -bitmap @[file join $env(EFFTCL_LIBRARY) \
     images fwd.xbm] -command "calendar_change $win +1"

  bind $win.cal <Configure> "calendar_redraw $win"

  return $win
}
```

Zu beachten ist momentan, daß H+McL einen globalen Array `calInfo` einführen mit den üblichen assoziativen Schlüsseln.

Zweitens wird wieder eine Klasse `Calendar` definiert, für die globale Optionen gesetzt werden können.

In die Hülle kommt eine Leinwand, deren `pack`-Optionen `-expand yes` und `-fill both` sind. Die Knöpfe zum Vor- und Zurückblättern der Monate erhalten Bitmaps. Die Knöpfe selbst werden aber nicht angelegt, sie kommen in Fenster, wenn der Kalender gezeichnet wird.

Dies wird ausgelöst durch das `<Configure>`-Ereignis, das an die Leinwand gebunden wird. Dieses wird auch beim erstmaligen Erzeugen automatisch ausgelöst, obwohl die Leinwandgröße mit 3"×2" zunächst fest vorgegeben ist.

Neugemalt werden muß auch, wenn der Monat wechselt, ausgelöst durch die `calendar_change`-Routine.

```
proc calendar_change {win delta} {
  global calInfo
  set dir [expr ($delta > 0) ? 1 : -1]
  set month \
    [clock format $calInfo($win-time) -format "%m"]
  set month [string trimleft $month 0]
  set year \
    [clock format $calInfo($win-time) -format "%Y"]

  for {set i 0} {$i < abs($delta)} {incr i} {
    incr month $dir
    if {$month < 1} {
      set month 12
      incr year -1
    } elseif {$month > 12} {
      set month 1
      incr year 1
    }
  }
  set calInfo($win-time) [clock scan "$month/1/$year"]
  calendar_redraw $win
}
```

Diese Routine verwendet die ziemlich aufwendigen `clock`-Kommandos und muß wegen der modulo 12 Rechnung bei Monaten ggf. auch das Jahr herauf- oder herunterzählen.

Die eigentliche Malroutine für den Monat ist natürlich ziemlich aufwendig und wird hier nur gekürzt wiedergegeben.

```
proc calendar_redraw {win} {
  global calInfo
  ...
  $win.cal delete all

  set time $calInfo($win-time)
  set wmax [winfo width $win.cal]
  set hmax [winfo height $win.cal]

  $win.cal create window 3 3 -anchor nw \
    -window $win.cal.back
  $win.cal create window [expr $wmax-3] 3 -anchor ne \
    -window $win.cal.fwd
  set bottom [lindex [$win.cal bbox all] 3]

  set font [option get $win titleFont Font]
  set title [clock format $time -format "%B %Y"]
  $win.cal create text [expr $wmax/2] $bottom -anchor s \
    -text $title -font $font

  incr bottom 3
  $win.cal create line 0 $bottom $wmax $bottom -width 2
  incr bottom 3

  set font [option get $win dateFont Font]
  set bg [option get $win dateBackground Background]
  set fg [option get $win dateForeground Foreground]
  ...

  set layout [calendar_layout $time]
  set weeks [expr [lindex $layout end]+1]

  foreach {day date dcol wrow} $layout {
    set x0 [expr $dcol*($wmax-7)/7+3]
    set y0 [expr $wrow*($hmax-$bottom-4)/ \
      $weeks+$bottom]
    set x1 [expr ($dcol+1)*($wmax-7)/7+3]
    set y1 [expr ($wrow+1)*($hmax-$bottom-4)/ \
      $weeks+$bottom]
    ...
    $win.cal create rectangle $x0 $y0 $x1 $y1 \
      -outline $fg -fill $bg
```

```
      $win.cal create text [expr $x0+4] [expr $y0+2] \
         -anchor nw -text "$day" -fill $fg -font $font

      $win.cal create image [expr $x1-2] [expr $y1-2] \
         -anchor se -tags [list $date-image]
      ...
    }
    ...
}
```

Wir übergehen die Details dieser Routine.

9.6.2 Sensoren und gebundene Kommandos

Viele Anwendungen haben sog. *hot spots*, d. h. Bereiche von Fenstern die man anklicken und ggf. selektieren kann. In unserem Beispiel will man z. B. einen Tag auswählen und sich die eingetragenen Termine anzeigen lassen.

Dazu könnte man die linke Maustaste an das Hintergrundrechteck für den Tag, seine Nummer und sein Icon binden. Man könne auch alle drei Items mit der selben Marke (tag) versehen und `<ButtonPress-1>` an dieses Tag binden.

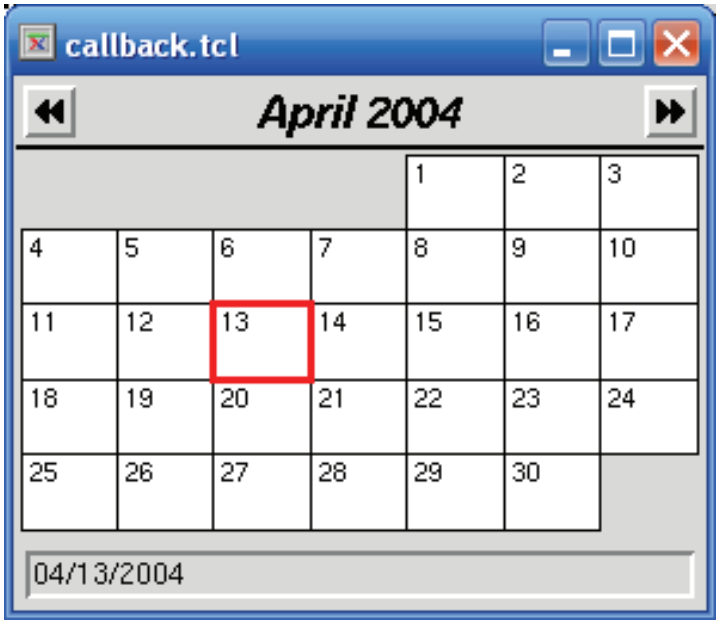
Einfacher ist es aber manchmal, über gewisse Bereiche ein unsichtbares Item zu legen und an dieses das Ereignis zu binden. Unsichtbare Rechtecke lassen sich mit `""` als Angabe für Rand- und Füllfarbe leicht erzeugen. Man sieht es zwar nicht auf dem Bildschirm, es reagiert aber durchaus auf Ereignisse. H+McL sprechen dann von einem Sensor.

In unserem Beispiel ließe sich so ein Sensor in der `redraw`-Routine innerhalb der `foreach`-Schleife mittels folgendem `create` und `bind` einbauen.

```
$win.cal create rectangle $x0 $y0 $x1 $y1 \
  -outline "" -fill "" \
  -tags [list $date-sensor all-sensor]

$win.cal bind $date-sensor <ButtonPress-1> \
  [list calendar_select $win $date]
```

Wichtig hierfür ist, daß das Rechteck **nach** dem Hintergrundrechteck, dessen Zahl und Bild, erzeugt wird, da es sonst von diesen verdeckt würde, was man dann durch `raise` oder `lower` korrigieren müßte.

Etwas undurchsichtig wieder die Namenserzeugung für den Tag. Der Sensor über dem Feld für den 1. Juli 1997 hätte demnach die Marke `07/01/1997-sensor`. An diese Marke wird `<ButtonPress-1>` gebunden, das `calendar_select` aufruft zum Auswählen des Tages, was ein Hin-/Herschalten *(toggle)* der Umrandung und anderer Merkmale zur Folge hat. Wir übergehen auch diese Routine, zumal der Callback, der an die Selektion gebunden ist, nur ein „*template*“ ist für das in `calInfo` gespeicherte Kommando und die geliebte Prozentsubstitution macht.



Im Quellcode zu Kapitel 4 [12] ist mit `callback.tcl` ein hübsches Skript abgelegt, das ein Entry-Widget mit dem ausgewählten Datum füllt. Es ließe sich gut in Anwendungen einbauen, die eine Datumsangabe verlangen, etwa wenn Anwender das Datum ihres Reisebeginns oder ein Abholdatum usw. angeben müssen. Mit der kalenderartigen Anordnung dürften deutlich weniger Fehler zu erwarten sein als mit den sonst üblichen linearen Ziehlisten.

9.6.3 Variablen überwachen

H+McL führen in diesem Unterabschnitt drei Techniken vor:

- das `trace`-Kommando
- die Namenssubstitution mittels `upvar`
- das Tcl-Pointerkonzept.

Als Anwendungsbeispiel wird die Eintragung von Bildern für Feiertage (Flagge für 4th of July, Glocke oder Weihnachtsbaum für 25. Dezember) und Geburtstage genommen. Diese Tage werden in Vektoren festgehalten, die zu konsultieren sind, wenn ein Tagrähmchen gemalt wird. Wird nachträglich ein Geburtstag eingefügt, sollte dies ein Neumalen des Kalendermonats auslösen.

Die Änderung einzelner Variablen läßt sich mit `trace variable` überwachen, einer Technik die aus Debuggern bekannt ist. Wichtig dabei ist, daß die überwachte Variable global ist, da sonst die Überwachung mit dem Ausstieg aus der Prozedur, in der `trace` aufgerufen wurde, endet. Was getan werden soll bei einer schreibenden Änderung (`w`) oder bei einem `unset` (`u`) wird – wie bei `bind` für ein Ereignis – in `trace` mit einer Callbackroutine angegeben.

H+McL verkomplizieren die Erläuterung noch damit, daß sie nicht `trace variable x wu "..."` aufrufen, sondern `trace variable $x wu "..."`.

Es wird somit der Wert von `x` als Variablenbezeichner genommen, d. h. `$` wirkt wie ein Dereferenzierungsoperator bei Zeigern. Ob das in Tcl eine sehr durchsichtige Strategie ist, kann diskutiert werden, da jede Variable potentiell als Pointer verstanden werden kann.

Auch `upvar` ist zunächst undurchsichtig. Es substituiert einen lokalen Namen für einen nicht-lokalen Bezeichner. Damit lassen sich Effekte wie bei einer Parameterübergabe „Call-by-Reference“ erzielen. Die gebräuchlichste Anwendung ist für den Zugriff auf Vektoren in einer Prozedur, da Vektoren (arrays) nicht als Argumente übergeben werden können.

Wie in C wird ein Zeiger auf den Vektor übergeben, der dann in der Prozedur dereferenziert werden muß.

```
proc druckmich name {
  upvar $name a
  foreach el [lsort [array names a]] {
    puts "$el = $a($el)"
  }
}

set Info(Alter) 37
set Info(Position) "Assistent C1"
druckmich Info
Alter = 37
Position = Assistent C1
```

Die Variable `a` steht also für den Wert des Parameters `name`, hier `Info`, und hat eine Wirkung, wie wenn sie eine Ebene höher im Aufrufstapel referenziert würde. Mit `upvar #0` wird sogar die globale Umgebung zum Gültigkeitsbereich.

Wir verzichten auf die Besprechung des Beispiels, da es einen komplizierten Anwendungsfall lösen will (Bilder für Feiertage und Geburtstage) und durch die gleichzeitige Einführung aller drei Lösungskonzepte dafür nicht gerade durchsichtiger wird.

9.7 Ein einfaches Malpaket

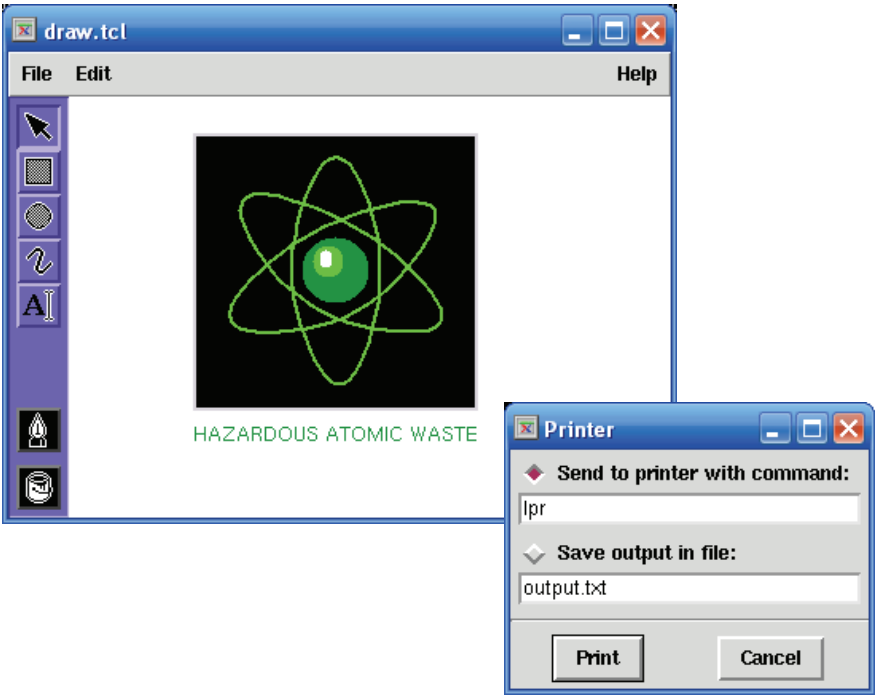
In H+McL folgt dann das Zeichenbrett (`draw.tcl`) mit den verschiedenen Malwerkzeugen, wie schon früher angerissen. Themen sind:

- die Gummibandtechnik zum Aufziehen und Selektieren von Items
- das Bewegen von Items
- das Skalieren von Items
- Farb- und Zeichenstiländerungen von Items
- Texteingabe und Texteditieren auf der Leinwand
- Postscriptausgabe
- Speichern und Laden einer Zeichnung

Interessant, weil einfach, ist die Postscriptdruckausgabe. Heißt z. B. die Leinwand `.drawing` dann kann mit

```
.drawing postscript
```

eine sehr lange Zeichenkette produziert werden, die sich als Postscript-Code der Zeichnung entpuppt.



Den Aufruf kann man in einer `draw_print` Prozedur verstecken, die dafür sorgt, daß – unabhängig vom sichtbaren Ausschnitt – die ganze Zeichnung ausgegeben wird.

```
proc draw_print {} {
  set x0 0
  set y0 0
  set x1 [winfo width .drawing]
  set y1 [winfo height .drawing]

  foreach {x0 y0 x1 y1} [.drawing bbox all] {}
  set w [expr $x1-$x0]
  set h [expr $y1-$y0]
```

```
    return [.drawing postscript -x $x0 -y $y0 -width $w \
      -height $h]
}
```

Den Aufruf nutzen wir als Kommando für einen Menüpunkt im Editor, dessen Aufruf und Druckerangaben etwa wie oben aussehen.

```
printer_create .print
.mbar.file.m add command -label "Print..." -command {
  printer_print .print draw_print
}
```

Die Ausgabe von `draw_print` wird dadurch an den richtigen Druckerdaemon weitergeleitet.

Sehr eindrucksvoll ist auch das Speichern und Laden von Zeichnungen, das eine Übersetzung aller Items in Tcl-Code vornimmt, wobei spezifische Widgetnamen ersetzt werden, damit die Zeichnungen auch in andere Umgebungen geladen werden können.

Somit wird z. B. aus

```
.drawing create oval 145.4 87.45 188.6 129.75 \
  -outline black -fill #23b397c5291d -width 2
```

dann in der Datei der Eintrag

```
draw oval 145.4 ... -width 2
```

Das Grundprinzip ist wie bei Postscript: wir produzieren als Ausgabe ein Programm, hier jetzt in Tcl.

Der Ladevorgang verwendet dann eine Zeichenroutine, z. B.

```
proc draw {args} {
  eval .drawing create $args
}
source $file
```

Damit wird aus

```
draw rectangle 79.0 ...
```

wieder ein

```
.drawing create rectangle 79.0 ...
```

Diese Methode birgt allerdings eine Gefahr. Ein übelwollender Zeitgenosse könnte uns in einer Zeichnung ein gefährliches Kommando, etwa ein `exec rm *`, unterjubeln, das bei Ausführung als Tcl-Zeichnung den Interpreter dazu bringt, unsere Dateien zu löschen.

Die Autoren gehen auf dieses Problem im Zusammenhang mit sicheren Interpretern im Abschnitt 7.6.6 [12] ein.

10 Das Textwidget

Die nächsten drei Kapitel aus H+McL werden hier nicht im Detail behandelt. Die folgenden Seiten dieses Skripts mit Anmerkungen zum Inhalt dienen deshalb hauptsächlich als Platzhalter, um die Numerierung konsistent mit der Vorlage des Buches zu halten (Kapitelnummer H+McL = Kapitelnummer Skript – 5).

Viele Anwendungen benötigen einen Texteditor, mit dem beliebige Texte eingegeben, angeschaut und verändert werden können. Textformate sollten einstellbar sein, Textteile sollten hervorgehoben und ausgeschnitten werden.

Das Textwidget `text` liefert diese Funktionalität und im Kapitel 5 [12] werden als Beispiele ein Email-Editor, ein Read-only Display für Fehlermeldungen, ein Editor für den Terminkalender und ein hierarchischer Dateibrowser gezeigt.

Wichtig für das Verständnis des Textwidgets ist die Adressierung des Textpuffers, der als Matrix von Zeilen (1, 2, ...) und Zeichen (0, 1, ...) einer Zeile indiziert werden kann.

Zweitens kann relativ zum Einfügezeiger (cursor) mit symbolischen Start- und Endadressen gearbeitet werden, etwa wenn das Wort oder die Zeile ausgewählt werden soll, in der der Zeiger momentan steht:

```
% .t get "insert wordstart" "insert wordend"
...
% .t get "insert linestart" "insert lineend"
```

Scrolling ist natürlich für lange Textpuffer notwendig, vieles davon ähnelt dem Leinwandwidget. Auch Tags sind wieder möglich für Ausschnitte des Texts, etwa eine Überschrift, wodurch Verhalten und Aussehen unabhängig von der konkreten Position des Textteils beeinflußt werden können. Mit Bindings lassen sich wieder Mausklicks an Tags binden, etwa in einem Web-Browser, bei dem ein Link die Farbe wechselt, wenn er angeklickt wird.

Größeren Raum nimmt auch die Diskussion über im Text eingebettete Fenster ein. Damit lassen sich Knöpfe, etwa wieder für eine Webseitenanzeige, oder auch Bilder in Texten darstellen. Natürlich fließen diese Widgets mit Texteinfügungen und -kürzungen.

11 Toplevel-Fenster

Wird die `wish` gestartet, legt sie ein Hauptfenster für die Anwendung an, in das andere Widgets gepackt werden. Daneben kann man aber in Tk mit

```
toplevel .popup
```

ein anderes Hauptfenster, genannt **toplevel widget**, erzeugen.

Damit lassen sich

- ein Warnfenster (sog. *alert*)
- ein Bestätigungsdialog
- eine Druckersteuerung (In Datei drucken? Auf welchen Drucker drucken?)
- eine Sprechblasenhilfe

bauen.

Unter anderem wird damit auch ein sog. modaler Dialog geübt (Beispiel `confirm.tcl`), bei dem eine Anwendung blockiert ist, bis der Anwender mit OK oder CANCEL den Eintritt eines Ereignisses bestätigt und eine Entscheidung getroffen hat.

Dazu gehört auch die Entwicklung einer `create`/`destroy`- und `show`/`hide`-Strategie, mit der diese vom unter Tk liegenden Windowmanager erzeugt, vernichtet, angezeigt und versteckt werden können.

Ganz nett sind auch die sog. **unmanaged windows**, d. h. Anzeigen ohne Rand und Ikonisierknöpfe, die nicht in der Größe verändert, verschoben oder ikonifiziert werden können. Man kennt sie als Werbebanner

beim Laden einer kommerziellen Applikation, spez. im Zusammenhang mit einem unaussprechlichen Betriebssystem. H+McL bezeichnen sie als **placards** (wörtlich: Plakate).

Schließlich tritt noch die von H+McL heißgeliebte Sprechblasenhilfe auf, bei der Hilfstexte mit einer gewissen Verzögerung erscheinen, wenn der Mauszeiger längere Zeit auf einem Widget verharrt. Zu Recht weisen die Autoren darauf hin, daß sich diese Art der Hilfestellung für geübte Anwender ausschalten läßt.

12 Zusammenwirken mit anderen Programmen

H+McL beschreiben in diesem ca. 50 Seiten starken Kapitel das Zusammenspiel von Tcl/Tk mit anderen Programmen. Zurecht weisen sie darauf hin, daß man das GUI als Frontend von der Applikationslogik trennen sollte, um auch bei der Fehlersuche klare Zuständigkeiten zu schaffen.

Glücklicherweise ist dies in Tcl/Tk in Verbindung mit UNIX gut möglich, da z. B. die Beschränkung auf einen einzigen Datentyp in Tcl, die Zeichenketten, exakt der UNIX-Philosophie der unstrukturierten Byteströme als Austauschmedium entspricht. Möglich wird damit der Aufruf eines UNIX-Kommandos innerhalb eines Tcl-Skripts mittels `exec`.. Das UNIX-Kommando lenkt seine Ausgabe in das Skript, z. B. in die Variable `info` im Codefragment unten, das ein Hilfefenster aus einer komprimierten Datei füttert.

```
set help [textdisplay_create "On-line Help"]
set info [exec zcat overview.Z]
textdisplay_append $help $info
```

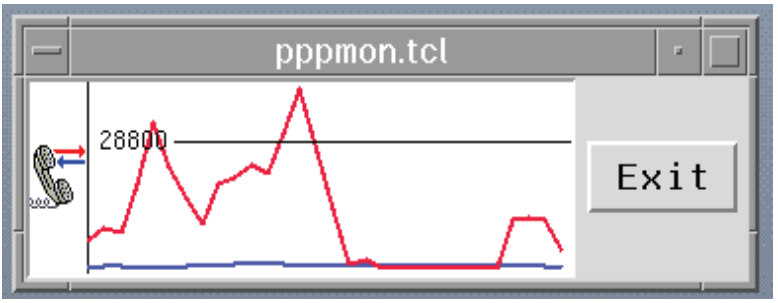
Wir können das Beispiel sogar dahin erweitern, daß wir eine Pipeline in das `exec`-Kommando stecken um etwa aus einer man-Pageausgabe Text für ein Hilfefenster zu erlangen.

```
set help [textdisplay_create "On-line Help"]
set info [exec zcat eccsman.Z | nroff -man | colcrt]
textdisplay_append $help $info "typewriter"
```

Dieses Thema wird in H+McL vertieft, auch in Zusammenhang mit dem `catch`-Kommando, um auftretende Fehler der angestoßenen UNIX-Programme zu fangen und anzuzeigen.

Ein Nachteil der obigen `exec`-Technik ist, daß das umgebende Tcl-Skript in seiner Abarbeitung blockiert (angehalten) wird, bis das mit `exec` aufgerufene Programm seine – vielleicht riesige – Ausgabe geliefert hat.

Für langlaufende Programme, die periodisch Daten abliefern – H+McL nennen einen PPP-Monitor als Beispiel – gibt es eine andere Möglichkeit: mit `open` kann eine Pipe genau wie eine Datei geöffnet werden und es kann aus dem zurückgelieferten Deskriptor mit `gets` gelesen werden.



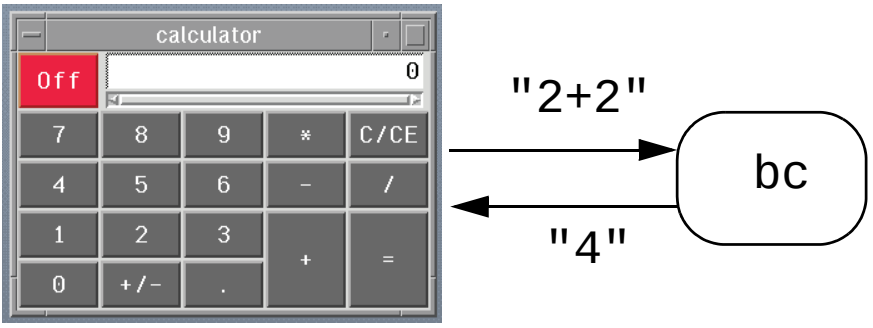
Über Fileevents kann deren periodische Datenanlieferung behandelt werden. Genauso kann Tcl auch Daten in eine Pipeline schreiben.

Ein weiteres Thema ist das Problem des verzögerten Schreibens. C-Programme unter UNIX, die ihre Standardausgabe nicht an den Bildschirm oder ein ähnliches Medium abliefern, puffern die Ausgabe aus Effizienzgründen zwischen, bis die Puffer (z. B. 4, 8 oder 32 KB) voll sind. Steckt in der GUI-Datenanlieferung eine Pipe, kann die Ausgabe am Bildschirm somit verzögert sein. H+McL betonen, daß Tcl immer seine Ausgabe sofort produziert. Trotzdem kann, durch Einsatz einer Pipe in der Ausgabe, die Datenanlieferung für eine interaktive Umgebung in unerwünschter Weise gepuffert sein. In diesen Fällen kann man mit dem `flush`-Kommando einzelne Werte sofort ausgeben oder noch besser mit `fconfigure` die Einstellung für einen Deskriptor generell ändern:

```
set fid [open "| cat" "w"]
fconfigure $fid -buffering line
puts $fid "Diese Zeile wird sofort ausgegeben"
puts $fid "Und diese auch."
```

Ein weiteres Hinweis aus dem sog. `Expect`-Paket, das eine Tcl-Version zur Unterstützung interaktiver Anwendungen darstellt, empfiehlt die Verwendung des `unbuffer`-Kommandos, das ein als Argument aufgeführtes anderes Kommando ausführt und diesem vorspiegelt, es schreibe auf den Bildschirm und müsse daher nicht puffern.

Das Pufferproblem taucht auch wieder auf, wenn Tcl in einer bidirektionalen Pipe mit einer anderen Anwendung kommuniziert, z. B. mit dem `bc` Rechenprogramm.



Hier muß man Wege finden, z. B. mit dem oben genannten Expect-Trick, die Ausgabe der Anwendung zu „entpuffern“.

Generell führt uns dieser Ansatz zu dem Thema **Client/Server**, wobei Tcl/Tk die Rolle einer GUI-Frontends übernehmen kann. Genauso kann Tcl wegen seiner Parsing-Fähigkeiten als Server eingesetzt werden, der Eingaben als Serverkommandos mit aktuellen Parametern interpretiert und zur Ausführung bringt.

Ein letztes, schwieriges Thema sind die sicheren Interpreter, eine Problematik, die auch von Java her bekannt ist. Wie schon früher ausgeführt, kann es gefährlich sein, auf dem eigenen Rechner einen beliebigen Tcl-Text zur Ausführung anzunehmen. Ist darin z. B. `exec rm -r .` enthalten, verliert man das Arbeitsverzeichnis und alle Unterverzeichnisse.

Im wesentlichen sind Dateioperationen (`open`) und das `exec`-Kommando gefährlich. Wir können aber in Tcl sog. sichere Interpreter anlegen, die diese Kommandos nicht kennen oder geeignet überlagern. Dies geht mit

```
set parser [interp create -safe]
set num [$parser eval {expr 2+2}]
```

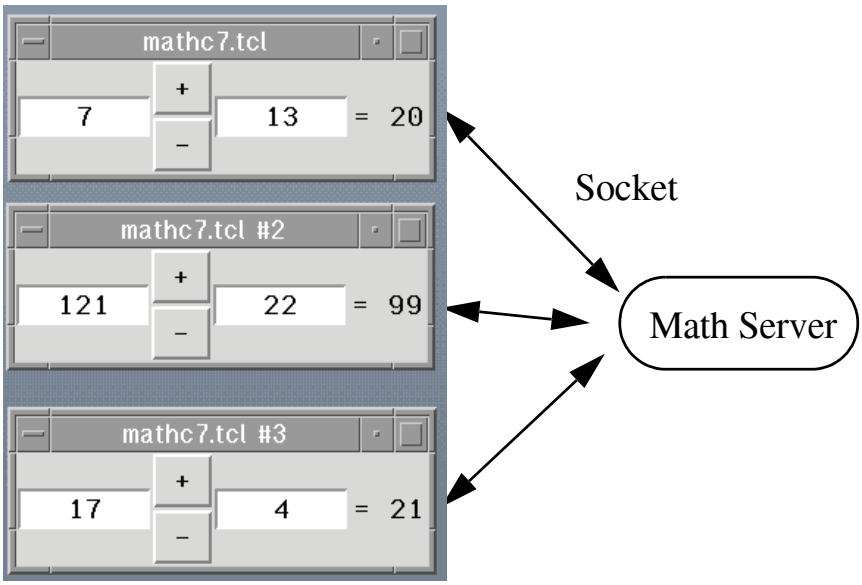
wobei wir hier in der Variablen `parser` einen neuen, sicheren Interpreter haben, dem wir den Ausdruck `2+2` zur Auswertung (`eval`) übergeben. Anwendungen, die mit diesem Umweg arbeiten, werden aber dadurch nicht gerade durchsichtiger.

Die Autoren wenden sich dann der asynchronen Kommunikation zu, die mit `fileevent`, `catch`, der Prozentsubstitution und der Argumentaufbereitung mit `list` arbeitet. Entsprechend anspruchsvoll, wenn auch nicht lang, sind die Beispiele.

Zuletzt gehen H+McL im Abschnitt 7.7 auf Netzwerkprogrammierung mit **Sockets** ein. Sockets sind ein UNIX IPC-Konzept, das sowohl Kommunikation zwischen Prozessen innerhalb eines UNIX-Systems (*UNIX domain*) als auch zwischen entfernten Rechnern (*Internet domain*) erlaubt. Die Kommunikation kann *verbindungsorientiert* (stream oriented, meist TCP/IP) oder *verbindungslos* (datagram service, meist UDP) gestaltet werden.

Das in H+McL besprochene Math-Server Beispiel verwendet die momentan von Tcl unterstützte Variante der Internet Stream Sockets (TCP/IP), bei der ein Server auf einem Rechner im Netz (host) einen Dienst unter einer bekanntgemachten Portnummer (für diese Art der Anwendung z. B. im Bereich 8100 - 9999) anbietet. Clients stellen über die IP-Adresse oder den Hostnamen unter Angabe der Portnummer eine Verbindung zum Server her (*connect*), der diese akzeptieren muß (*accept*).

Danach wird der Server in der Regel einen Unterprozeß oder einen Thread generieren, um den Client zu bedienen. Server und Client reden miteinander ähnlich wie in einer Pipeverbindung. Nach Erledigung der Aufgaben wird der Client die Verbindung schließen, worauf der Server dies auch tun und angelegte Ressourcen, z. B. einen Fileidentifier, freigeben sollte.



Das folgende Programm stellt den Math-Server vor.

```
set parser [interp create -safe]
$parser eval {
  proc percent_subst {percent string subst} {
    if {![string match %* $percent]} {
      error "bad pattern \"$percent\": \
        should be %something"
    }
    regsub -all {\\|&} $subst {\\\0} subst
    regsub -all $percent $string $subst string
    return $string
  }
}

$parser eval {
  proc add {x y cmd} {
    set num [expr $x+$y]
    set response [percent_subst %v $cmd $num]
    return $response
  }
  proc subtract {x y cmd} {
    set num [expr $x-$y]
    set response [percent_subst %v $cmd $num]
```

```
      return $response
    }
  }

  proc server_accept {cid addr port} {
    fileevent $cid readable "server_handle $cid"
    fconfigure $cid -buffering line
  }

  proc server_handle {cid} {
    global parser buffer

    if {[gets $cid request] < 0} {
      close $cid
    } else {
      append buffer $request "\n"
      if {[info complete $buffer]} {
        set request $buffer
        set buffer ""
        if {[catch {$parser eval $request} result] == 0} {
          puts $cid $result
        } else {
          puts $cid [list error_result $result]
        }
      }
    }
  }

  socket -server server_accept 9001
  vwait enter-mainloop
```

Im Gegensatz zu der undurchsichtigen Pointerlösung in C ist die Tcl-Lösung bedingt durch den reinen Austausch von Zeichenketten recht einfach. Der Aufruf zur Einrichtung des Sockets beim Server lautet

```
socket -server Kommando Port
```

bei uns also

```
socket -server server_accept 9001
```

Dem Kommando, hier `server_accept`, werden automatisch drei Argumente übergeben, den Filehandle für die Verbindung zum Client, die IP-Adresse des Clients und die Client-Portnummer.

Über das `fileevent`-Kommando bekommt der Server mit, wenn die Verbindung zum Client so steht, daß aus ihr gelesen werden kann. Dies führt dann zum Aufruf der Routine `server_handle` und gleichzeitig setzen wir die Verbindung auf ungepuffert (`line`-Mode).

Die Routine `server_handle` behandelt Eingabezeilen für den Server, indem sie die Anfrage liest und zusammensetzt. Das so entstehende Kommando wird in einem sicheren Interpreter ausgeführt und das Ergebnis wird an den Client zurückgeschickt.

Die letzte Anweisung im Serverprogramm oben setzt die Ereignisschleife in Gang und führt dazu, daß der Server nach Anfragen von Kunden sich umhört (UNIX Sockets: *listen*).

Den Server kann man im übrigen gut von Hand testen, wiederum ein Vorteil des Tcl-Interpreteransatzes.

```
wegner@elsie(chapter7)$ tclsh maths7.tcl &
[1] 22398
wegner@elsie(chapter7)$ telnet localhost 9001
Trying...
Connected to localhost.
Escape character is '^]'.
add 2 3 %v
5
add 2 3 "set x %v"
set x 5
exec date
error_result {invalid command name "exec"}
```

Die Client-Seite (`mathc7.tcl`) sieht wie folgt aus, wobei man wissen muß, daß vereinbarungsgemäß der Server Antworten der Art `show_result` oder `error_result` schickt, die im Client in einem sicheren Interpreter ausgeführt werden sollen und zur Anzeige des Ergebnisses führen.

```
package require Efftcl
option add *Entry.width 7 startupFile
option add *Entry.background white startupFile
option add *Entry.justify center startupFile
entry .x
pack .x -side left
```

```
frame .op
pack .op -side left
button .op.add -text "+" -command do_add
pack .op.add -fill both
button .op.sub -text "-" -command do_subtract
pack .op.sub -fill both
entry .y
pack .y -side left
label .result -text ""
pack .result -side left

proc do_add {} {
  set x [.x get]
  set y [.y get]
  client_send add $x $y {show_result %v}
}

proc do_subtract {} {
  set x [.x get]
  set y [.y get]
  client_send subtract $x $y {show_result %v}
}

proc cmd_show_result {num} {
  .result configure -text "= $num"
}

proc cmd_error_result {msg} {
  notice_show $msg error
}

set parser [interp create -safe]
$parser alias show_result cmd_show_result
$parser alias error_result cmd_error_result

proc client_handle {sid} {
  global backend parser buffer

  if {[gets $sid request] < 0} {
    catch {close $sid}
    set backend ""
    notice_show "Lost connection to server" error
  } else {
    append buffer $request "\n"
    if {[info complete $buffer]} {
```

```
        set request $buffer
        set buffer ""
        if {[catch {$parser eval $request} result] != 0} {
          notice_show $result error
        }
      }
    }
  }

  proc client_send {args} {
    global backend
    if {$backend != ""} {
      puts $backend $args
    }
  }

  set sid [socket localhost 9001]
  fileevent $sid readable "client_handle $sid"
  fconfigure $sid -buffering line
  set backend $sid
```

Auf Details verzichten wir, wie auch auf das schöne Beispiel mit dem elektronischen Gruppenterminkalender.

13 Tcl/Tk-Anwendungen ausliefern

Die Autoren des Buches stellen zwei Auslieferungsmodi vor:

- die Arbeitsplatzinstallation, bei der eine vollständige Auslieferung der Anwendung mit Pixelbildern, Programm- und Bibliothekscode und der wish erfolgen soll, und die
- Webinstallation, bei der die Anwendung Teil einer Webseite ist.

Letzterer Modus ist besonders für Demos und kleine, häufig aktualisierte Beiträge gedacht.

Wir wollen auf beide Modi kurz eingehen.

13.1 Die Anwendung auf Hochglanz bringen

Eine alte Softwareregel lautet: 90% der Anwendung beanspruchen 90% der Zeit, die restlichen 10% der Anwendung benötigen die anderen 90% der Zeit. Auch wenn es zum Ablieferungstermin immer knapp wird, sollte man Zeit für letztes Handanlegen beim äußeren Erscheinungsbild vorsehen.

Dazu gehört die Handhabung der Widget-Ressourcen, z. B. Farben und Fonts, die man bis zum Schluß offenhalten sollte. Die Tk Optionen-Datenbank ist hierzu das wesentliche Mittel.

Für die Taschenrechneranwendung könnte man das Aussehen durch die folgenden Kommandos festlegen.

```
option add *Entry.background white startupFile
option add *Scrollbar.width 8 startupFile
option add *readout.justify right startupFile
option add *Button.background DimGray startupFile
option add *Button.foreground white startupFile
option add *quit.background red startupFile
option add *quit.foreground white startupFile
```

Jedes Kommando setzt eine Standardvorbelegung in der Datenbasis. Sie wirken nur in dem Programm, in dem sie auftauchen und beeinflußen Widgets, die in folgenden Programmschritten angelegt werden.

Eine Zeichenkette `*Entry.background` beschreibt eine Widgetoption, die man auch traditionell eine Ressource nennt. Jede Ressource besteht aus einer Folge von Namen getrennt durch „`*`“ oder „`.`“, wobei „`*`“ wie „jedes“ und „`.`“ wie „die ... hat“ zu interpretieren ist. Demnach ist `*Entry.background` zu lesen als „jedes Entry-Widget, das die Background-Option gesetzt hat“. In unserem Fall wird diese Ressource auf `white` gesetzt.

Wie schon früher betont, ist die Groß- und Kleinschreibung der Namen relevant; Namen mit Großbuchstaben, wie in `Entry`, beziehen sich auf die Klasse, also alle `entry`-Instanzen des Programms. Dies geht übrigens auch für Styleoptionen, also z. B. `Background` als Vertreter der tatsächlichen Optionen `-activebackground`, `-selectforeground` und `-troughcolor`.

Das letzte Wort jeder Zeile enthält die Angabe `startupFile`, das der Einstellung eine Priorität verleiht, auf die unten eingegangen wird.

Farbsetzungen funktionieren oft nicht so wie gedacht, z. B. wenn nur eine Monochrom(schwarz/weiß)-Ausgabe verfügbar ist. Über

```
if {[string match *color [winfo screenvisual .]]} {
  ... color ...
} else {
  ... s/w ...
}
```

läßt sich dies einstellen mit der Chance, die ostfriesische Nationalfahne zu vermeiden.

Vorsicht ist mit der Schreibweise geboten. Durch Eingabe von

```
option add *Entry.borderwidth 4 startupFile
```

scheint man die Randbreite vorzubelegen. Dem ist nicht so, es passiert nichts, weil man die `-borderwidth`-Option mit der `borderWidth`-Ressource (großes `W`) kontrolliert. Dies bestätigt die folgende Anfrage:

```
$ wish
% entry .test
.test
% .test configure -borderwidth
-borderwidth borderWidth BorderWidth 2 2
```

Die `configure`-Operation liefert fünf Werte ab. Der erste ist der Name der Option, `-borderwidth`, der zweite ist der Ressourcenname, der dritte die Ressourcenklasse für die Optionendatenbasis. Somit gehört die Ressource `borderWidth` zur Klasse der `BorderWidth`-Optionen.

H+McL beschreiben als nächstes sehr anschaulich, wie man sich eigene Ressourcen mit neuen Namen schaffen kann, deren An-/Ausschalten man im Programm mit `option get` abfragen kann.

13.1.1 Prioritäten

Wir hatten oben im `option-add`-Kommando die `startupFile` Priorität gesetzt.

```
option add *Button.background DimGray startupFile
```

Diese Priorität ist so niedrig, daß ein Anwender die Vorgaben durch eigene Einstellungen überstimmen kann. Ohne die Angabe `startupFile` wäre die Besetzung aber fest auf `DimGray` unabhängig davon, was der Anwender zu setzen versucht.

Alternativ ließen sich unter UNIX Ressourcenvorgaben über `.Xresources` und `.Xdefaults` setzen. Für unser Taschenrechnerprogramm `calc` würden sie lauten:

```
calc*Button.background: yellow
calc*Button.foreground: black
calc*clear.background: red
```

```
calc*Scrollbar.borderWidth: 1
calc*readout.font: -*-courier-bold-r-normal--*-240-*
calc*readout.width: 10
calc.printTape: yes
```

Grundsätzlich ist die Angabe `startupFile` die richtige für eine einzelne, große Anwendung. Dagegen nimmt man die noch niedrigere Priorität `widgetDefault` für Bibliothekscode, der Eingang in viele Anwendungen findet.

13.1.2 Fehlerbehandlung

Alle größeren Anwendungen müssen mit unerwarteten Fehlern rechnen. H+McL machen Vorschläge für elegantes Sammeln von Fehlern in einer Fehlerlog-Datei und automatischer Benachrichtigung der Entwickler. Ein Beispiel für das Fehlerlogbuch wird in `logerr.tcl` gegeben.

```
package require Efftcl

proc bgerror {error} {
  set fid [open "/tmp/log.txt" a]
  puts $fid $error
  close $fid
}

button .err -text "Error" -command {
  expr 200e3000*1
}
pack .err
```

Die Lösung für das Fehlerfenster und den Versand der E-Mail gibt es in der `catalog.tcl`-Demo. Sie ist wegen doppelter Quotes usw. recht tricky. Die verschickte E-Mail an eine Phantasieadresse sieht wie folgt aus (enthält den Fehlerstack).


```
From: "Lutz Wegner" <wegner@DB.Informatik.Uni-Kassel.DE>
Message-Id: <199807091918.VAA21732@elsie>
To: efftcl-bugs@aw.com
Subject: BUG REPORT (lib/demos/bgerror.tcl)
content-length: 538

While the following program was executing...
------------------------------------------------------
 lib/demos/bgerror.tcl
------------------------------------------------------
...the following error was detected:

this is an error!
 while executing
"error "this is an error!""
 invoked from within
".error invoke"
 ("uplevel" body line 1)
 invoked from within
"uplevel #0 [list $w invoke]"
```

```
    (procedure "tkButtonUp" line 8)
    invoked from within
  "tkButtonUp .error"
    (command bound to event)
```

Wir übergehen die Details.

13.1.3 Startlogos (Placards)

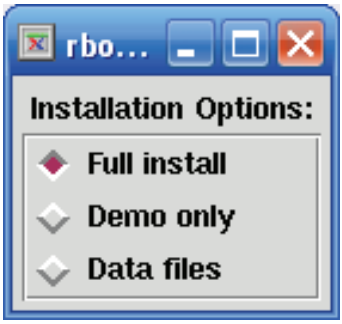
Auch recht aufwendig ist die Anzeige eines animierten Startlogos für die Zeit des Ladens einer größeren Anwendung. Der besondere Trick der von H+McL vorgestellten Lösung besteht im Starten einer zweiten `wish`, die ihre Ausgabe per `update` sofort anzeigt und sich vollständig abräumt, wenn die ursprüngliche Anwendung fertig zur Arbeitsaufnahme ist. Interessant, aber ...

13.2 Tcl/Tk-Bibliotheken anlegen

In diesem Abschnitt entwickeln die Autoren nochmals didaktisch sehr sorgfältig ihre Prinzipien der portablen Programmentwicklung und behandeln die Punkte

- Bibliothekskomponenten (~`_create`, ~`_add`, ~`_destroy`, ...)
- globale assoziative Arrays mit generierten Namen
- Callbacks für die Komponenten
- Autoloading von Prozeduren (via `auto_path` und `auto_mkindex` *Verzeichnis*)
- Packages

Den ersten Teil, der eine konventionelle Entwicklung einer auf Bibliothekscode ausgerichteten Programmierung entgegenstellt, wollen wir kurz zeigen, den Rest müssen wir hier wegen Zeitmangel übergehen.



Die konventionelle Lösung für die Radiobox oben lautet (`rbox1.tcl`):

```
frame .rbox
pack .rbox -padx 4 -pady 4

label .rbox.title -text "Installation Options:"
pack .rbox.title -side top -anchor w

frame .rbox.border -borderwidth 2 -relief groove
pack .rbox.border -expand yes -fill both

radiobutton .rbox.border.rb1 -text "Full install" \
   -variable current -value "Full install"
pack .rbox.border.rb1 -side top -anchor w

radiobutton .rbox.border.rb2 -text "Demo only" \
   -variable current -value "Demo only"
pack .rbox.border.rb2 -side top -anchor w

radiobutton .rbox.border.rb3 -text "Data files" \
   -variable current -value "Data files"
pack .rbox.border.rb3 -side top -anchor w

.rbox.border.rb1 invoke
```

Der generische Ansatz für das Anlegen einer Radiobox lautet dagegen:

```
proc radiobox_create {win {title ""}} {
  global rbInfo

  set rbInfo($win-current) ""
  set rbInfo($win-count) 0
```

```
    frame $win -class Radiobox

    if {$title != ""} {
       label $win.title -text $title
       pack $win.title -side top -anchor w
    }
    frame $win.border -borderwidth 2 -relief groove
    pack $win.border -expand yes -fill both

    bind $win <Destroy> "radiobox_destroy $win"
    return $win
}
```

Die beachtenswerten Punkte sind:

- Der Prozedurname genügt den Konventionen, wonach das zu erzeugende Widget, hier `radiobox`, zuerst genannt wird, dann zum Anlegen das `create`-Suffix.
- Die Prozedur legt eine Hülle an (frame), die die Widgetkreation umfaßt. Ihr Name wird nicht festverdrahtet, sondern als Argument übergeben, somit kann einmal eine Radiobox mit
  `radiobox_create .installieren`
  und ein andermal eine andere Box mit
  `radiobox_create .deinstallieren`
  erzeugt werden.
- Die Prozedur setzt die Klasse auf `Radiobox` (großes R). Somit können Vorbesetzungen in der Optionendatenbasis abgelegt werden, etwa
  ```
  option add *Radiobox.selectColor \
    ForestGreen startupFile
  ```
  wobei diese Einstellung nur die Radioboxes betrifft, nicht alle Radiobuttons der Anwendung!
- Die Prozedur führt kein `pack`-Kommando aus, packt aber die Widgetkomponenten im Inneren. Damit kann der Aufrufer entscheiden, wie und mit welchem Geometriemanager, etwa durch
  ```
  radiobox_create .flavors
  pack .flavors -padx 4 -pady 4
  ```
  er die erzeugte Hülle plazieren will.

- Die Prozedur liefert wie ein Standardwidget den Namen der Hülle als Resultat zurück.
- Ein Titel kann als optionaler Parameter übergeben werden.
- Die Prozedur fügt die Wahlmöglichkeiten nicht selbst ein, da deren Anzahl von Aufruf zu Aufruf wechseln könnte. Dafür wird z. B. für drei Auswahlmöglichkeiten dreimal `radiobox_add` mit dem Widgetbezeichner und dem Auswahltext aufgerufen (`rbox2.tcl`):

```
radiobox_create .options \
  "Installation Options:"
pack .options -padx 4 -pady 4

radiobox_add .options "Full install"
radiobox_add .options "Demo only"
radiobox_add .options "Data files"
```

was natürlich die folgende Prozedur voraussetzt:

```
proc radiobox_add {win choice {command ""}} {
  global rbInfo

  set name \
  "$win.border.rb[incr rbInfo($win-count)]"
  radiobutton $name -text $choice \
    -command $command \
    -variable rbInfo($win-current) \
    -value $choice
  pack $name -side top -anchor w

  if {$rbInfo($win-count) == 1} {
    $name invoke
  }
}
```

- Desweiteren wird man eine Auswahlprozedur (setzen Default beim ersten Anzeigen) – `radiobox_select` –, eine Abfrageprozedur `radiobox_get` sowie `radiobox_destroy` brauchen. Jede der Prozeduren bekommt den Hüllennamen als Argument.

H+McL betonen zu recht, daß dadurch im Vergleich zu normalen Widgets eine etwas andere Syntax entsteht, d. h. statt wie bei einem normalen Widget zu schreiben

```
radiobox .flavors
.flavors add "Vanilla"
.flavors add "Chocolate"
```

schreiben wir jetzt

```
radiobox_create .flavors
radiobox_add .flavors "Vanilla"
radiobox_add .flavors "Chocolate"
```

An dieser Stelle folgt bei H+McL der Unterabschnitt über den globalen Array, um z. B. für jede der möglichen Radioboxes die Besetzung der Variable `current` (Teil der `radiobutton`-Deklaration, siehe konventionelle Lösung oben) abzuspeichern.

Diesen Teil sowie Erläuterungen zum Autoloading und zu Paketen (`package require Efftcl`) übergehen wir.

13.3 Desktop-Anwendungen

Wie man ein selbstinstallierendes Programm produziert ...

13.4 Web-Anwendungen

Webseiten enthalten heutzutage aktive Elemente, sog. Applets. Mit Tcl/Tk lassen sich gut diese *embedded applications* schreiben, allerdings unterstützt Netscape unter AIX nicht das Tcl-Plugin.

Für die Abbildung unten wurde die Radiobox etwas aufgebohrt. Die Autoren betonen, daß einfache Tcl/Tk-Skripten meist ohne Modifikation in einer Browser-Umgebung funktionieren.

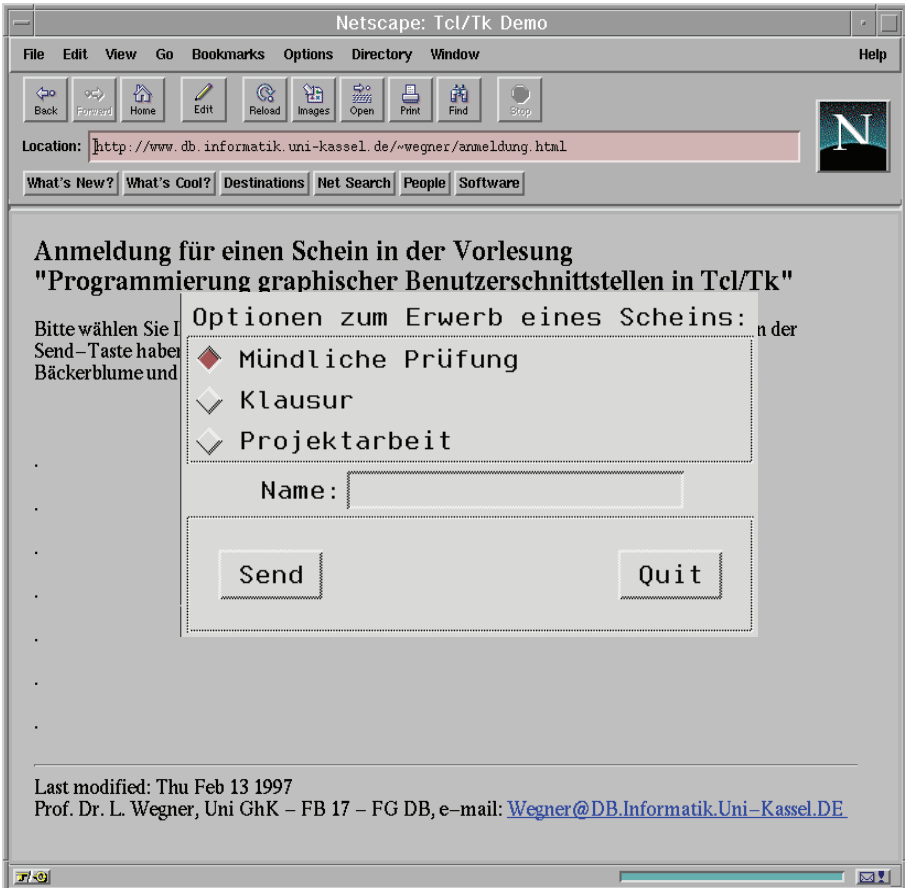


Abb. 13–1 *Simuliertes Bild; tatsächliches Aussehen variiert je nach Fähigkeit des Browsers. Zu Risiken und Nebenwirkungen fragen Sie Ihren ...*

Die Einbettung ist einfach, verlangt aber nach absoluten Größenangaben für seine Anzeige.

```
<HTML>
<HEAD>
<TITLE>Tcl/Tk Demo</TITLE>
<LINK REV="made"
HREV="mailto:Wegner@DB.Informatik.Uni-Kassel.DE">
</HEAD>
<BODY>
```

```
<H2>Anmeldung für einen Schein in der Vorlesung <br>
"Programmierung graphischer Benutzerschnittstellen
in Tcl/Tk"</H2>
<P>
Bitte wählen Sie Ihre Leistungsnachweisart und tragen
Sie Ihren Namen ein. Mit Drücken der Send-Taste haben Sie
sich angemeldet und gleichzeitig verbindlich ein
Jahresabo für die Bäckerblume und eine Lamawolldecke für
289,- DM bestellt.<P>
<P>
<embed src="schein.tcl" width=500 height=300>
<P>
<HR><!-- hhmts start -->
Last modified: Thu Feb 13 1997
<!-- hhmts end -->
<BR>
Prof. Dr. L. Wegner, Uni GhK - FB 17 - FG DB, e-mail:
<A HREF="mailto:Wegner@DB.Informatik.Uni-Kassel.DE">
Wegner@DB.Informatik.Uni-Kassel.DE
</A>
</BODY>
</HTML>
```

Weiterhin wichtig ist, daß Webseite und Appletskript im gleichen Verzeichnis liegen sollten.

Applets haben weitere Einschränkungen. So ist es wichtig, daß der gesamte Code, also auch Bibliotheksroutinen, alle Bitmaps und Images, die im Code für Anzeigen (z. B. Icons) gebraucht werden, in **einer** Datei vorliegen, die sich der Browser laden kann. Bilddaten müssen dazu inline in das Skript gestellt werden, ggf. codiert mittels `mimencode`.

Auch Menüs sind in Applets nicht erlaubt. Der Grund liegt darin, daß ein Menü den Focus hält, wodurch ein böswilliges Applet den Bildschirm einfrieren könnte. Aufklappmenüs müssen deshalb in Knöpfe gewandelt werden.

Auch `toplevel`-Widgets sind nicht erlaubt, da sie sich über den ganzen Bildschirm erstrecken oder Formen annehmen könnten, die den Anwender dazu bringen, geheime Information, z. B. ein Paßwort oder Kreditkartendaten, preiszugeben. Stattdessen sind *frames* zu verwenden.

Die für `toplevel`-Widgets mögliche Unterstützung durch einen Windowmanager (z. B. Titel, Ikonifizierung) entfällt.

Applets können kein `grab` machen und dürfen nicht die `vwait`-Schleife benutzen. Genausowenig dürfen sie uneingeschränkt auf Dateien zugreifen oder Sockets verwenden.

Es gibt allerdings die Möglichkeit, dem Tcl/Tk-Plug-in Direktiven zu geben, die erweiterte Rechte, z. B. in einem Intranet, zubilligen. Dazu gehören lt. Autoren

- `package require Browser`
  Applet kann Web-Seiten laden und Javascript-Kommandos ausführen.
- `package require Http`
  Zugriff auf kundenseitiges `HTTP/1.0` Protokoll und Proxyserver
- `package require Safesock`
  Socketverbindung zum Server, der das Applet geliefert hat.
- `package require Tempfile`
  Dateien anlegen und entfernen im `/tmp`-Bereich.
- `package require Trusted`
  Zugriff auf alle Tcl/Tk-Fähigkeiten, setzt Eintrag in Liste der sicheren Applets im Tcl/Tk-Plug-in voraus.

Nach unseren Tests funktionierten diese Angaben allerdings nicht. Das oben genannte, in der Webseite eingebettete Skript `schein.tcl` sah vielmehr wie folgt aus:

```
#package require Safesock
policy home

#set server $env(SERVER)
set server [getattr originHost]
set port 9002

if {[catch {socket $server $port} sid] != 0} {
  label .error -text "Keine Verbindung zum Server"
  pack .error -side top
}
```

```
fconfigure $sid -buffering line

frame .rbox
pack .rbox -padx 4 -pady 4

label .rbox.title -text \
  "Optionen zum Erwerb eines Scheins:"
pack .rbox.title -side top -anchor w

frame .rbox.border -borderwidth 2 -relief groove
pack .rbox.border -expand yes -fill both

radiobutton .rbox.border.rb1 -text "Mündliche Prüfung" \
  -variable current -value "Mündliche Prüfung"
pack .rbox.border.rb1 -side top -anchor w

radiobutton .rbox.border.rb2 -text "Klausur" \
  -variable current -value "Klausur"
pack .rbox.border.rb2 -side top -anchor w

radiobutton .rbox.border.rb3 -text "Projektarbeit" \
  -variable current -value "Projektarbeit"
pack .rbox.border.rb3 -side top -anchor w

frame .rbox.anmelder
pack .rbox.anmelder -padx 4 -pady 4

label .rbox.anmelder.label -text "Name:"
pack .rbox.anmelder.label -side left

entry .rbox.anmelder.student -relief sunken -bd 2
pack .rbox.anmelder.student -side left -anchor w \
  -expand yes -fill x

frame .rbox.buttons -borderwidth 2 -relief groove
pack .rbox.buttons -expand yes -fill both

button .rbox.buttons.send -text "Send" -command {
  puts $sid "[.rbox.anmelder.student get] $current"
}
pack .rbox.buttons.send -side left -padx 20 -pady 20

button .rbox.buttons.quit -text "Quit" -command {
  catch {close $sid}
  exit
```

```
}

pack .rbox.buttons.quit -side right -padx 20 -pady 20

.rbox.border.rb1 invoke
```

Den Server, der die Anmeldung annimmt und die Daten in eine Datei `anmeldeliste` schreibt, müssen wir auf dem selben Rechner laufen lassen, von dem die Webseite geladen wird, bei uns `gretel`.

```
proc server_accept {cid addr port} {
  fileevent $cid readable "server_handle $cid"
  fconfigure $cid -buffering line
}

proc server_handle {cid} {

  if {[gets $cid request] < 0} {
     close $cid
  } else {
     set fid [open "./anmeldeliste" a]
     puts $fid $request
     flush $fid
     close $fid
  }
}

socket -server server_accept 9002
vwait enter-mainloop
```

Damit beenden wir diese Behandlung der eingebetteten Tcl-Applikationen.

Zugleich ist damit auch unsere Besprechung des Buches von Harrison und McLennan zu Ende, wobei wir das neunte Kapitel zum Thema plattformübergreifende Entwicklungen überschlagen haben.

Anhang A
Regeln des Tcl-Interpreters

Die folgenden zwölf Regeln wurden dem Tcl/Tk-wiki auf http://wiki.tcl.tk/10259 entnommen und sind die kurzgefasste Sprachbeschreibung von Tcl aus der eigentlichen Tcl 8.5 Dokumentation auf

http://www.tcl.tk/man/tcl8.5/TclCmd/Tcl.htm

In dem Wiki heissen sie *Dodekalogue*, das engl. Wort für *Dodekalogie*, was wiederum der aus dem Griechischen stammende Begriff für zwölf zusammenhängende Kunstwerke ist. Eine deutsche Übersetzung findet man auf http://wiki.tcl.tk/4592 („Einfach man Tcl“). Wir geben hier den englischen Text geringfügig typographisch aufbereitet wieder. Die Angaben in [5] zur Argumentauflösung mittels vorangestelltem {*} sind neu für Tcl 8.5, genauso wie die Tcl-Dictionaries mit den `dict`-Kommandos.

The following rules define the syntax and semantics of the Tcl language:

[1] Commands.

A Tcl script is a string containing one or more commands. Semicolons and newlines are command separators unless quoted as described below. Close brackets are command terminators during command substitution (see below) unless quoted.

[2] Evaluation.

A command is evaluated in two steps. First, the Tcl interpreter breaks the command into words and performs substitutions as described below. These substitutions are performed in the same way for all

commands. The first word is used to locate a command procedure to carry out the command, then all of the words of the command are passed to the command procedure. The command procedure is free to interpret each of its words in any way it likes, such as an integer, variable name, list, or Tcl script. Different commands interpret their words differently.

[3] Words.

Words of a command are separated by white space (except for newlines, which are command separators).

[4] Double quotes.

If the first character of a word is double quote (“"”) then the word is terminated by the next double quote character. If semicolons, close brackets, or white space characters (including newlines) appear between the quotes then they are treated as ordinary characters and included in the word. Command substitution, variable substitution, and backslash substitution are performed on the characters between the quotes as described below. The double quotes are not retained as part of the word.

[5] Argument expansion.

If a word starts with the string “{*}” followed by a non-whitespace character, then the leading “{*}” is removed and the rest of the word is parsed and substituted as any other word. After substitution, the word is parsed again without substitutions, and its words are added to the command being substituted. For instance, “cmd a {*}{b c} d {*}{e f}” is equivalent to “cmd a b c d e f”.

[6] Braces.

If the first character of a word is an open brace (“{”) and rule [5] does not apply, then the word is terminated by the matching close brace (“}”). Braces nest within the word: for each additional open brace there must be an additional close brace (however, if an open brace or close brace within the word is quoted with a backslash then it is not

counted in locating the matching close brace). No substitutions are performed on the characters between the braces except for backslash-newline substitutions described below, nor do semi-colons, newlines, close brackets, or white space receive any special interpretation. The word will consist of exactly the characters between the outer braces, not including the braces themselves.

[7] Command substitution.

If a word contains an open bracket ("[") then Tcl performs command substitution. To do this it invokes the Tcl interpreter recursively to process the characters following the open bracket as a Tcl script. The script may contain any number of commands and must be terminated by a close bracket ("]"). The result of the script (i.e., the result of its last command) is substituted into the word in place of the brackets and all of the characters between them. There may be any number of command substitutions in a single word. Command substitution is not performed on words enclosed in braces.

[8] Variable substitution.

If a word contains a dollar sign ("$") then Tcl performs variable substitution: the dollar sign and the following characters are replaced in the word by the value of a variable. Variable substitution may take any of the following forms:

$*name* *Name* is the name of a scalar variable; the name is a sequence of one or more characters that are a letter, digit, underscore, or namespace separators (two or more colons).

$*name*(*index*) *Name* gives the name of an array variable and index gives the name of an element within that array. *Name* must contain only letters, digits, underscores, and namespace separators, and may be an empty string. Command substitutions, variable substitutions, and backslash substitutions are performed on the characters of *index*.

${*name*} *Name* is the name of a scalar variable. It may contain any characters whatsoever except for close braces.

There may be any number of variable substitutions in a single word. Variable substitution is not performed on words enclosed in braces.

[9] Backslash substitution.

If a backslash (“\”) appears within a word then backslash substitution occurs. In all cases but those described below the backslash is dropped and the following character is treated as an ordinary character and included in the word. This allows characters such as double quotes, close brackets, and dollar signs to be included in words without triggering special processing. The following table lists the backslash sequences that are handled specially, along with the value that replaces each sequence.

\a Audible alert (bell) (0x7).

\b Backspace (0x8).

\f Form feed (0xc).

\n Newline (0xa).

\r Carriage-return (0xd).

\t Tab (0x9).

\v Vertical tab (0xb).

\<newline>whiteSpace

A single space character replaces the backslash, newline, and all spaces and tabs after the newline. This backslash sequence is unique in that it is replaced in a separate pre-pass before the command is actually parsed. This means that it will be replaced even when it occurs between braces, and the resulting space will be treated as a word separator if it isn't in braces or quotes.

\\ Backslash (“\”).

ooo The digits *ooo* (one, two, or three of them) give an eight-bit octal value for the Unicode character that will be inserted. The upper bits of the Unicode character will be 0.

\x*hh* The hexadecimal digits *hh* give an eight-bit hexadecimal value for the Unicode character that will be inserted. Any number of hexadecimal digits may be present; however, all but the last two are ignored (the result is always a one-byte quantity). The upper bits of the Unicode character will be 0.

\u*hhhh* The hexadecimal digits *hhhh* (one, two, three, or four of them) give a sixteen-bit hexadecimal value for the Unicode character that will be inserted.

Backslash substitution is not performed on words enclosed in braces, except for backslash-newline as described above.

[10] Comments.

If a hash character (“#”) appears at a point where Tcl is expecting the first character of the first word of a command, then the hash character and the characters that follow it, up through the next newline, are treated as a comment and ignored. The comment character only has significance when it appears at the beginning of a command.

[11] Order of substitution.

Each character is processed exactly once by the Tcl interpreter as part of creating the words of a command. For example, if variable substitution occurs then no further substitutions are performed on the value of the variable; the value is inserted into the word verbatim. If command substitution occurs then the nested command is processed entirely by the recursive call to the Tcl interpreter; no substitutions are performed before making the recursive call and no additional substitutions are performed on the result of the nested script.

Substitutions take place from left to right, and each substitution is evaluated completely before attempting to evaluate the next. Thus, a

sequence like “set y [set x 0][incr x][incr x]” will always set the variable y to the value, 012.

[12] Substitution and word boundaries.

Substitutions do not affect the word boundaries of a command, except for argument expansion as specified in rule [5]. For example, during variable substitution the entire value of the variable becomes part of a single word, even if the variable's value contains spaces.

Literatur

[1] Marshall Brain, Motif programming: the essentials ... and more, Digital Press, Burlington, MA, 1992

[2] Eric F. Johnson and Kevin Reichard, Professional Graphics Programming in the X Window System, MIS:Press, New York, NY, 1993

[3] Dan Heller, Motif Programming Manual (2nd printing), O'Reilly & Associates, Sebastopol, CA, 1992

[4] A. Kay and A. Goldberg, Personal dynamic media, IEEE Computer, Vol. 10, No. 3 (1977) pp. 31-44

[5] Arnold Klingert, Einführung in Graphische Fenstersysteme: Konzepte und reale Systeme, Springer, Berlin-Heidelberg-New York, 1996

[6] Kevin Mullet and Darrell Sano, Designing Visual Interfaces, Prentice-Hall, Englewood Cliffs, N.J., 1995, 273pp.

[7] John K. Ousterhout, Tcl und Tk, Entwicklung graphischer Benutzerschnittstellen für das X Window System, Addison-Wesley, Bonn, 1995 (Titel der amerikanischen Orginalausgabe: Tcl and the Tk Toolkit, Addison-Wesley, 1994)

[8] Jenny Preece et al., Human-Computer-Interaction, Addison-Wesley, Wokingham, England, 1994

[9] Marc J. Rochkind, Advanced C Programming for Displays, Prentice Hall, Englewood Cliffs, NJ, 1988

[10] Sun Microsystems Computer Corporation, The Java Language Specification, Version 1.0 Beta, Oct. 1995

[11] Brent B. Welch and Ken Jones, Practical Programming in Tcl and Tk, Fourth Edition, Prentice Hall, 2003

[12] Harrison, M. and McLennan, M., Effective Tcl/Tk Programming. Writing Better Programs with Tcl and Tk. Addison-Wesley, 1998. 400 pp. ca. $ 45.95, ISBN 0201634740

[13] Stone, M. and Laird, C., Special Edition Using Tcl/Tk 8.0, Que, 1998. 600 pp. incl. CD-ROM. ca. DM 89,95, ISBN 0789714671

[14] Harrison, M., Tcl/Tk Tools. O'Reilly, 1997. 675 pp. incl. CD-ROM with Tcl/tk, the extensions, and other tools in source and as binaries for Solaris and Linux. DM 99,-, ISBN 1565922182

[15] Steve McConnel: What Have You Learned Today, Interview with Brian Kernighan, IEEE Software, March/April 1999, pp. 66-68

[16] Myers, Brad A. "User Interface Software Tools." *ACM Transactions on Computer-Human Interaction 2*, 1 (March 1995): 64-108.

[17] Gibson, J. J. (1979): *The ecological approach to visual perception*. Boston, Houghton Mifflin.

[18] Paul Klimsa und Kai Bruns, Kap. 12 - Multimedia. in: Informatik für Ingenieure kompakt, Bruns und Klimsa (Hrsg.) Vieweg 2001

[19] Kurt Wall. Tcl and Tk Programming for the absolute beginner, Thomson Course Technology, 2008