FRIEDRICH-SCHILLER-
UNIVERSITÄT
JENA

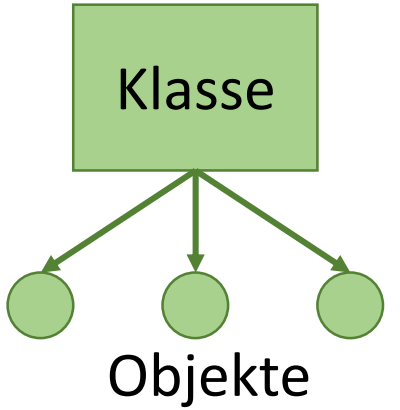
Einführung in Python

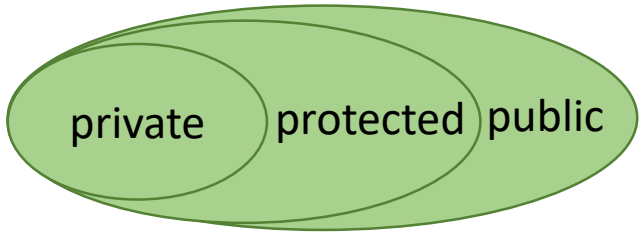
6. Vorlesung

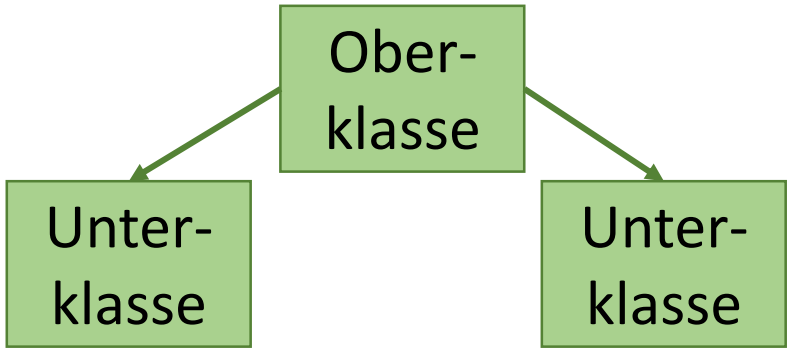
Objektorientierte Modellierung GUI-Programmierung mit Tkinter

Wiederholung letztes Mal

- Objektorientierte Programmierung mit Python
  - Objekte: Daten (**Attribute**) und deren Funktionen (**Methoden**) sind als Objekte zusammengefasst; Diese werden von **Klassen** abgeleitet.
  - Datenabstraktion: Daten eines Objekts sollten vor **direktem Zugriff** geschützt und von außerhalb **nicht sichtbar** sein.
  - Vererbung: Weitergabe von Eigenschaften einer **allgemeinen Oberklasse** an eine **spezialisierte Unterklasse**.





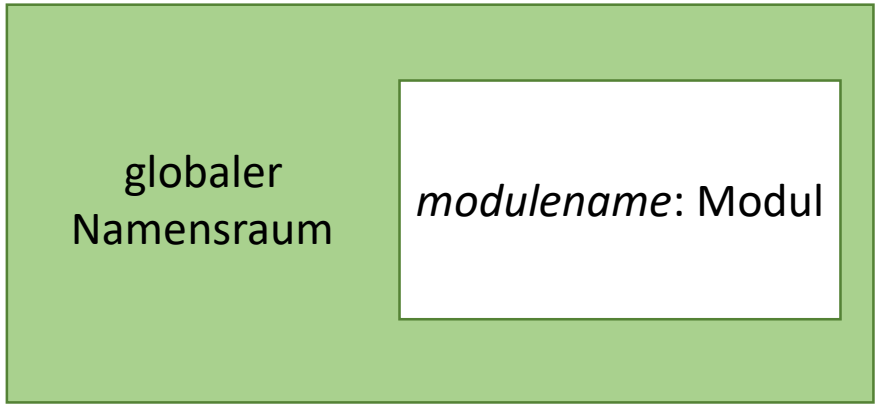


Wiederholung letztes Mal

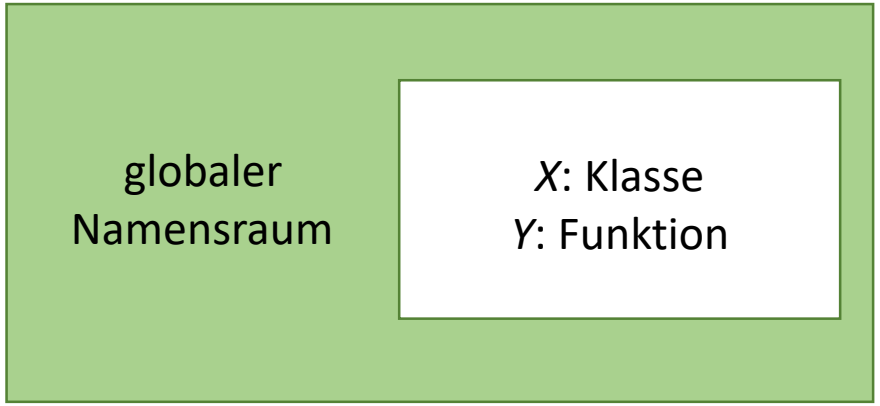
- Der Befehl **import** *modulname* lädt das gesamte Modul unter seinem Namen in den globalen Namensraum

- Mit **from** *modulename* **import** *name* lassen sich gezielt Klassen und Funktionen eines Moduls in den Namensraum laden

import *modulname*



from *modulname* import *X, Y*



Objektorientierte Modellierung

- Phasen einer objektorientierten Software-Entwicklung
  - OOA
  - OOD
  - OOP
- Assoziationen zwischen Klassen
  - Reflexive Assoziation
  - Aggregation

FRIEDRICH-SCHILLER-
UNIVERSITÄT
JENA

Phasen einer OO Software-Entwicklung

- Objektorientierte Analyse (OOA):
  - Analyse des Wirklichkeitsausschnitts, welcher durch ein Programm abgebildet werden soll
  - Umgangssprachliche Beschreibung des Verhalten des Systems
  - Festlegen von Klassen und Beziehungen zwischen diesen

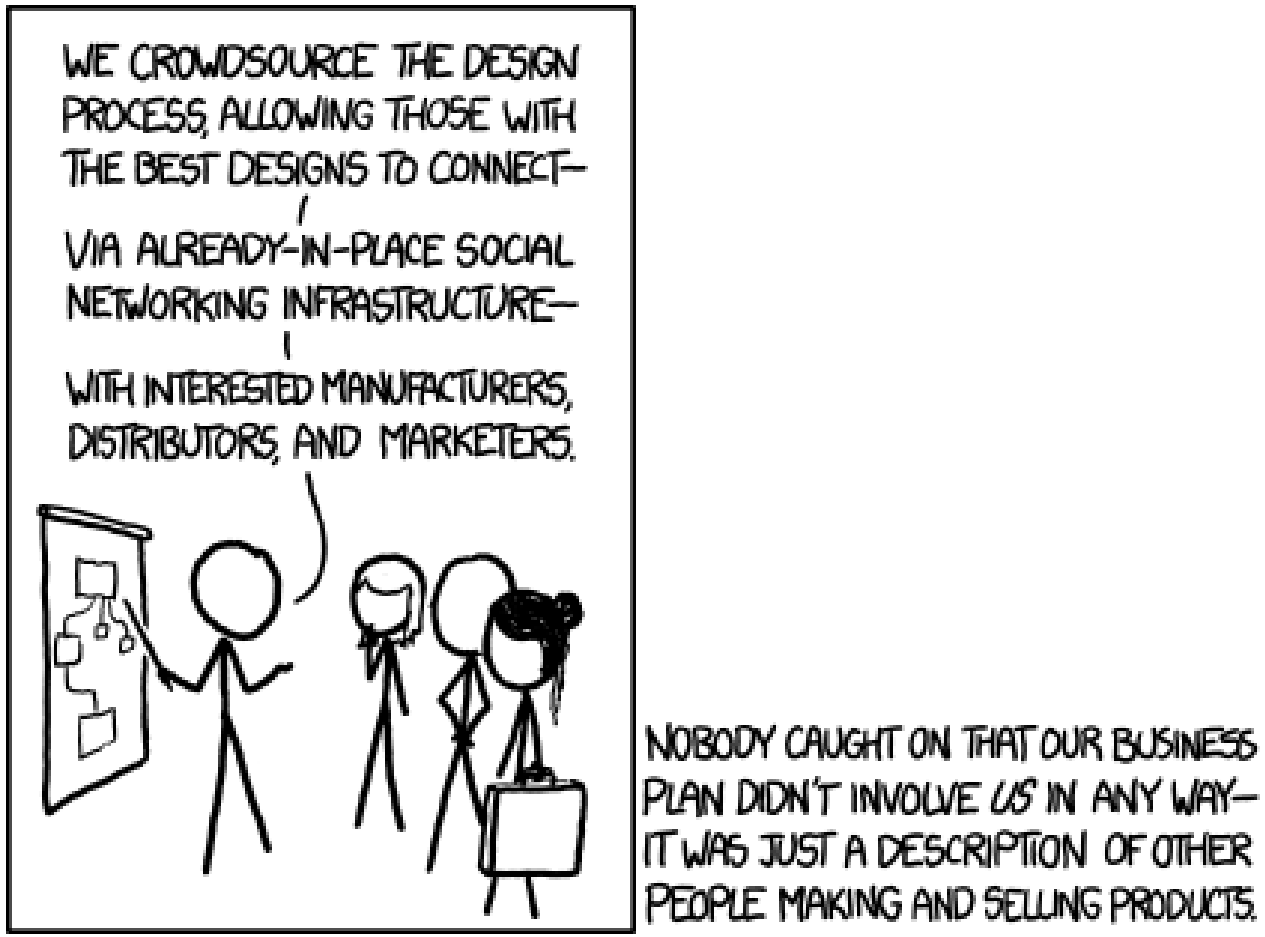
  →Abstraktes Modell ohne konkrete technische Realisierung

- Objektorientierter Entwurf (OOD):
  - Verfeinerung des OOA-Modells hinsichtlich praktischer Umsetzung und Effizienz in ein OOD-Modell
  - Wahl der Programmiersprache; Sichtbarkeit von Attributen und Methoden; Aufteilung des Gesamtsystems auf Module; Festlegung von Vor- und Nachbedingungen

- Objektorientierte Programmierung (OOP):
  - Implementierung, Dokumentierung und Testen des OOD-Modells

Fallbeispiel: Modell einer Sequenzdatenbank

- OOA: Abbildung der Realität in ein abstraktes Modell

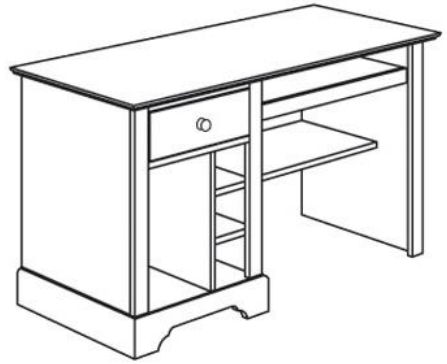


Fallbeispiel: Modell einer Sequenzdatenbank

- Sammlung und Verwaltung verschiedener biologischer Sequenzen in geeigneter Form

- Passende Benutzungsoberfläche (Schnittstelle) um auf den Inhalt dieser Sammlungen zuzugreifen

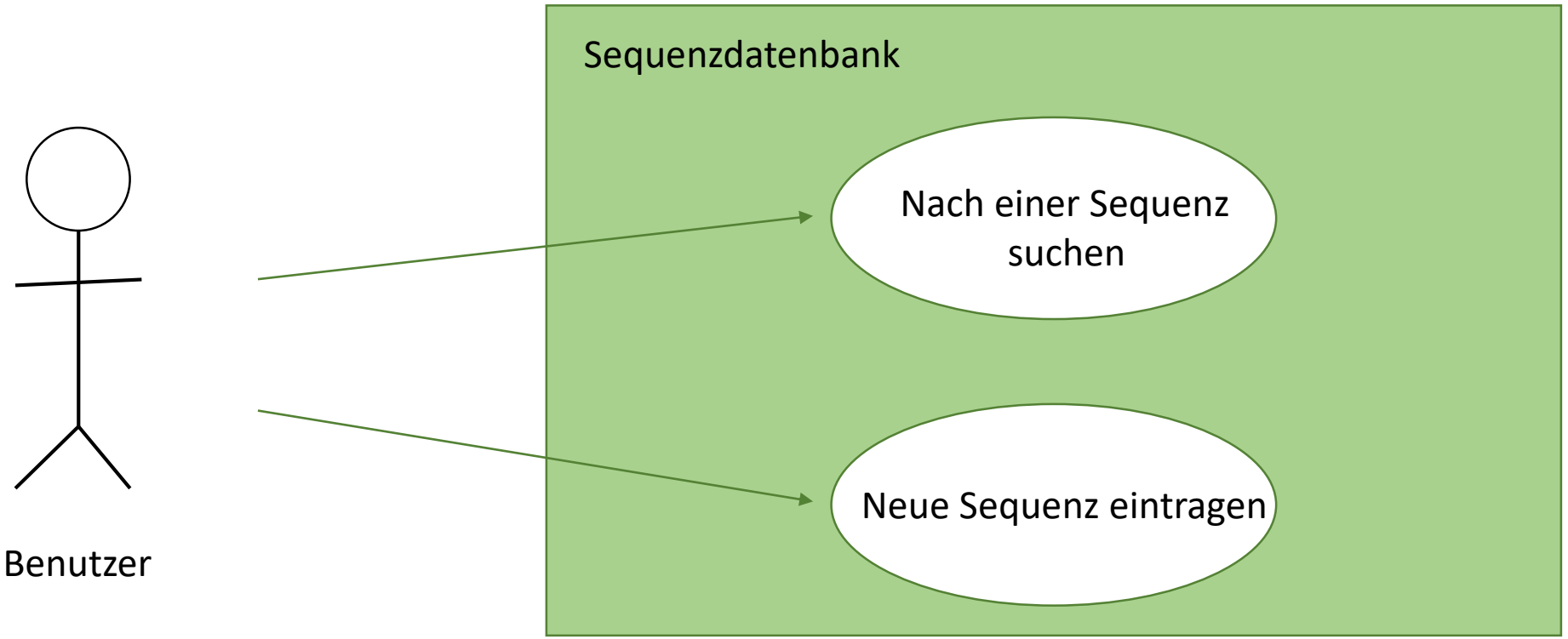
Benutzungsoberfläche



Sequenzdatenbank

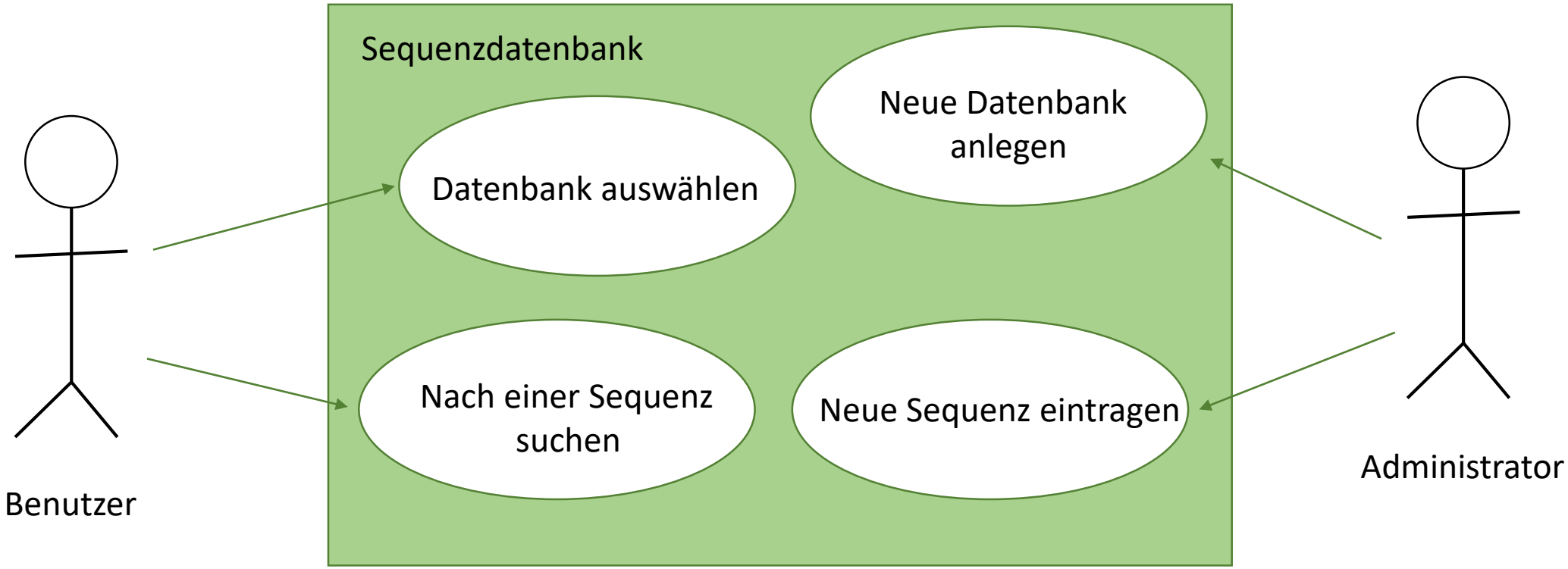
Fallbeispiel: Modell einer Sequenzdatenbank

- **Geschäftsprozesse** (*use cases*) sind typische Anwendungsfälle für die Software und werden mit eigenen Namen in einem Diagramm festgehalten.



Fallbeispiel: Modell einer Sequenzdatenbank

- **Geschäftsprozesse** (*use cases*) sind typische Anwendungsfälle für die Software und werden mit eigenen Namen in einem Diagramm festgehalten.



Fallbeispiel: Modell einer Sequenzdatenbank

- Jeder einzelne Geschäftsprozess wird umgangssprachlich beschrieben, wobei man sich meist an folgendes Schema orientiert:
  - Name
  - Ziel
  - Vorbedingung
  - Nachbedingung Erfolg
  - Nachbedingung Fehlschlag
  - Akteure
  - Auslösendes Ereignis
  - Beschreibung

Fallbeispiel: Modell einer Sequenzdatenbank

- Geschäftsprozess *Nach einer Sequenz suchen*:
  - Name — Sequenz suchen
  - Ziel — biologische Sequenz zu einem Header ausgeben
  - Vorbedingung — -
  - Nachbedingung Erfolg — Die Sequenz des dazugehörigen Headers in gut lesbarer Form auf den Bildschirm ausgeben, danach ist das System bereit für eine neue Eingabe
  - Nachbedingung Fehlschlag — Ausgabe einer Fehlermeldung (‘Nicht gefunden‘)
  - Akteure — Benutzer
  - Auslösendes Ereignis — Auswahl der Funktion *Suchen* im Menü
  - Beschreibung — Eingabe eines Headers; Ausgabe einer Sequenz

Fallbeispiel: Modell einer Sequenzdatenbank

- Geschäftsprozess *Neue Sequenz eintragen*:
  - Name — Sequenz eintragen
  - Ziel — Datenbank um eine Sequenz und ihren Header erweitern
  - Vorbedingung — Header und Sequenz müssen das richtige Format haben
  - Nachbedingung Erfolg — Der eingegebene Header und Sequenz sind in der Datenbank verzeichnet. Falls vor dem Prozess ein Eintrag mit gleichem Header existierte, wird dieser überschrieben, danach ist das System bereit für eine neue Eingabe
  - Nachbedingung Fehlschlag — -
  - Akteure — Administrator
  - Auslösendes Ereignis — Auswahl der Funktion *Sequenz eintragen* im Menü
  - Beschreibung — Eingabe eines Headers; Eingabe einer Sequenz; Beenden des Prozesses durch Bestätigung

Fallbeispiel: Modell einer Sequenzdatenbank

- Geschäftsprozesse spiegeln das gewünschte Verhalten des Systems aus Nutzersicht wieder, enthalten aber keine vollständigen Details der Systemfunktionalität (z. B. automatische Operationen wie Speichern oder Laden einer vorhandenen Sequenzdatenbank)

- Auf Grundlage der ausformulierten Geschäftsprozesse lassen sich die dazugehörigen Klassenstrukturen verfeinern

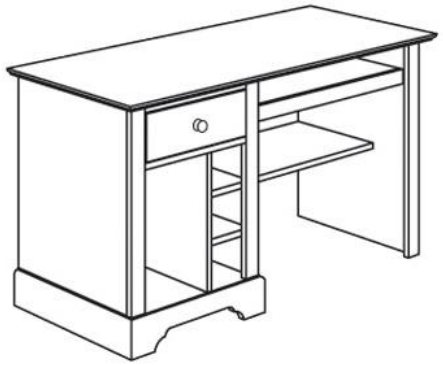
Fallbeispiel: Modell einer Sequenzdatenbank

- Klasse Sequenzdatenbank
  - <u>Attribute:</u>
  - sequenztyp — beschreibt die Art der Sequenzen
  - alphabet — definiert die erlaubten Zeichen
  - sequenzen — Abbildung von Headern auf Sequenzen
  - pfad — Ort an dem die Sequenzen gespeichert werden

  - <u>Methoden:</u>
  - suchen()
  - einfügen()
  - speichern()

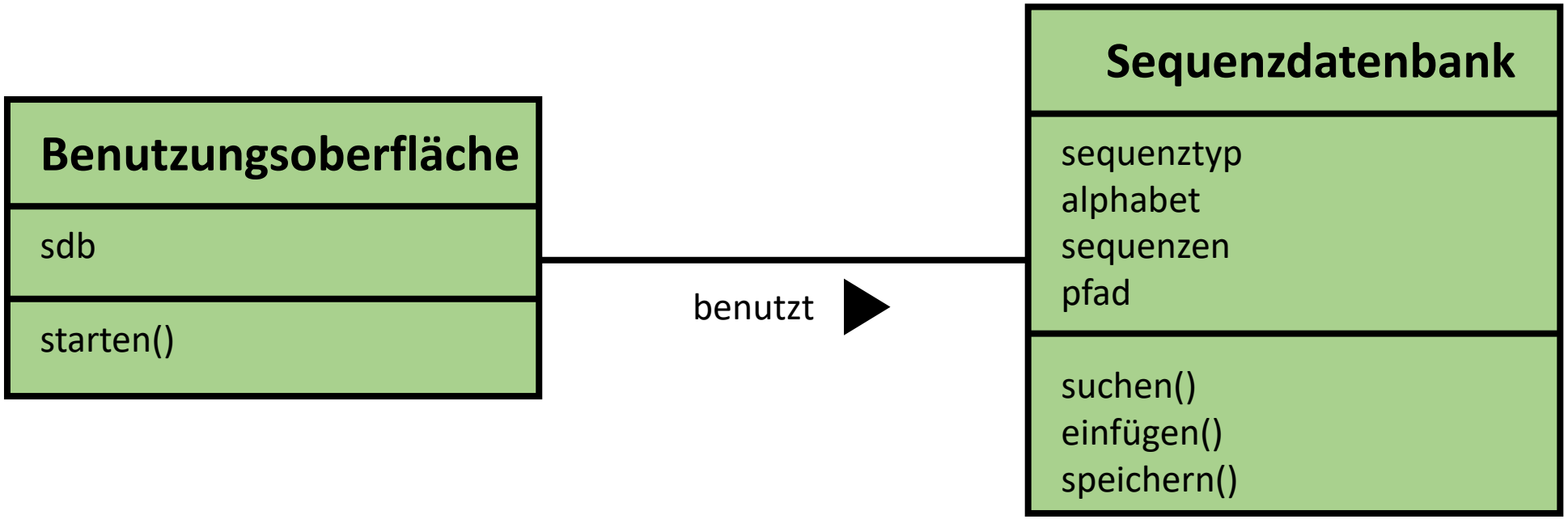
Fallbeispiel: Modell einer Sequenzdatenbank

- Klasse Benutzungsoberfläche
  - <u>Attribute:</u>
  - sdb — Referenz eines Sequenzdatenbank-Objekts
  - <u>Methoden:</u>
  - starten() — Starten der Benutzeroberfläche und überführt sie in einen Zustand in dem sie auf Benutzereingaben wartet



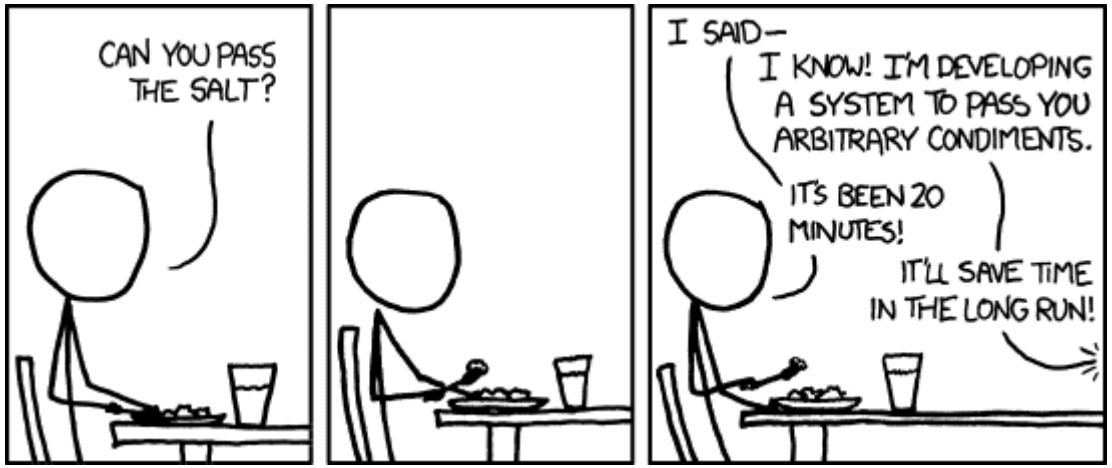
Fallbeispiel: Modell einer Sequenzdatenbank

- *Assoziationen* zwischen Klassen entstehen dann, wenn Instanzen der einen Klasse von der anderen als Attribute benutzt werden



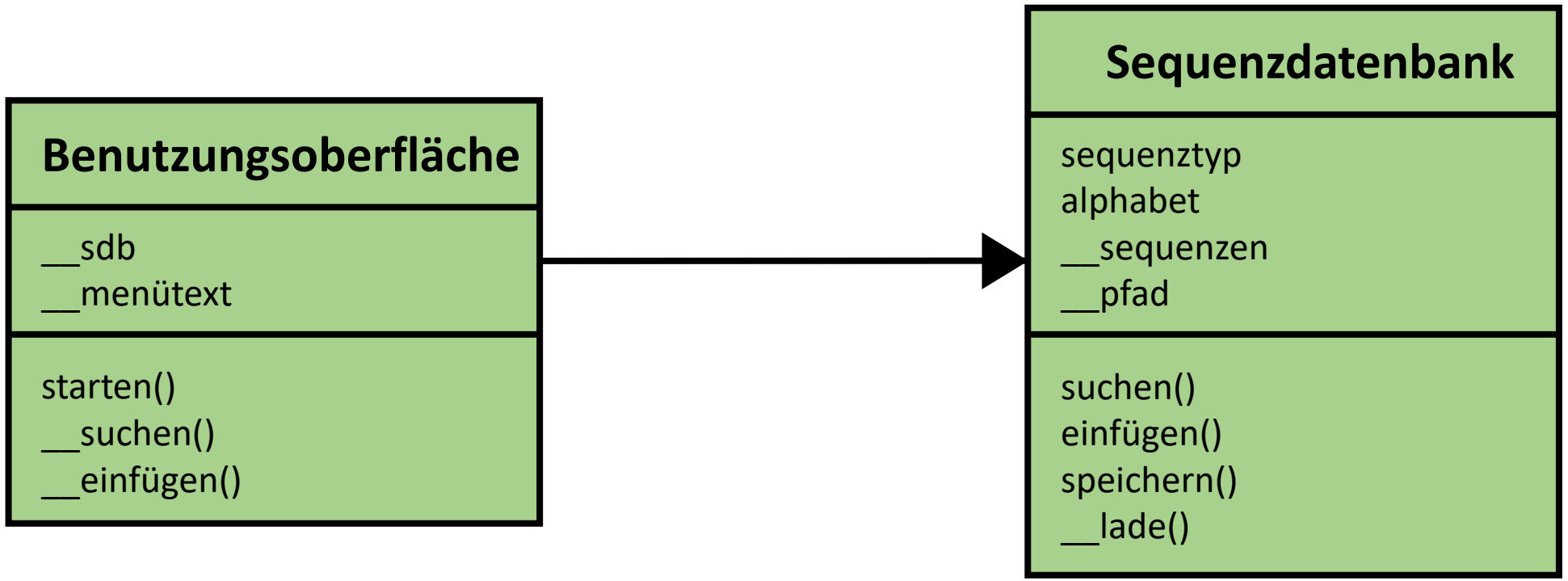
Fallbeispiel: Modell einer Sequenzdatenbank

- OOD: Spezifizierung des abstrakten OOA-Modells



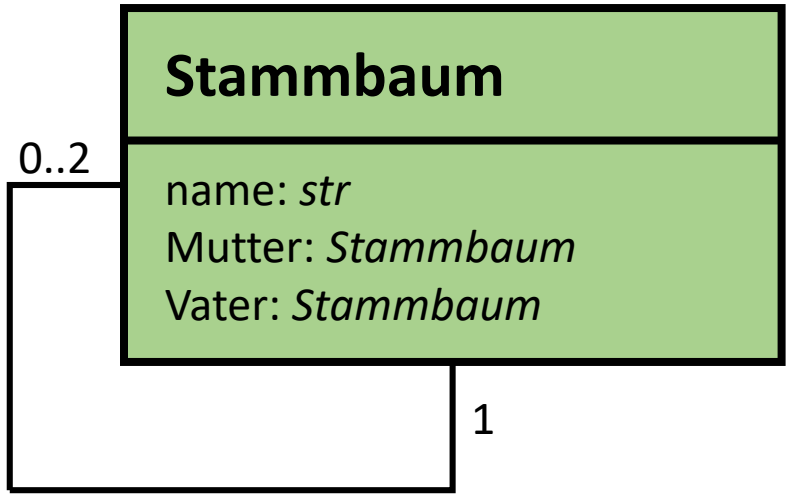
Fallbeispiel: Modell einer Sequenzdatenbank

- OOD: Spezifizierung des abstrakten OOA-Modells

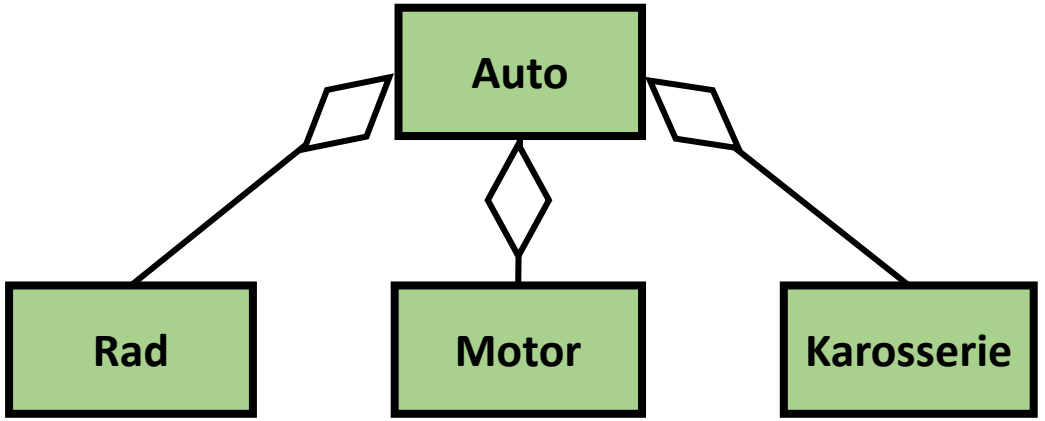


Weitere Assoziationen zwischen Klassen

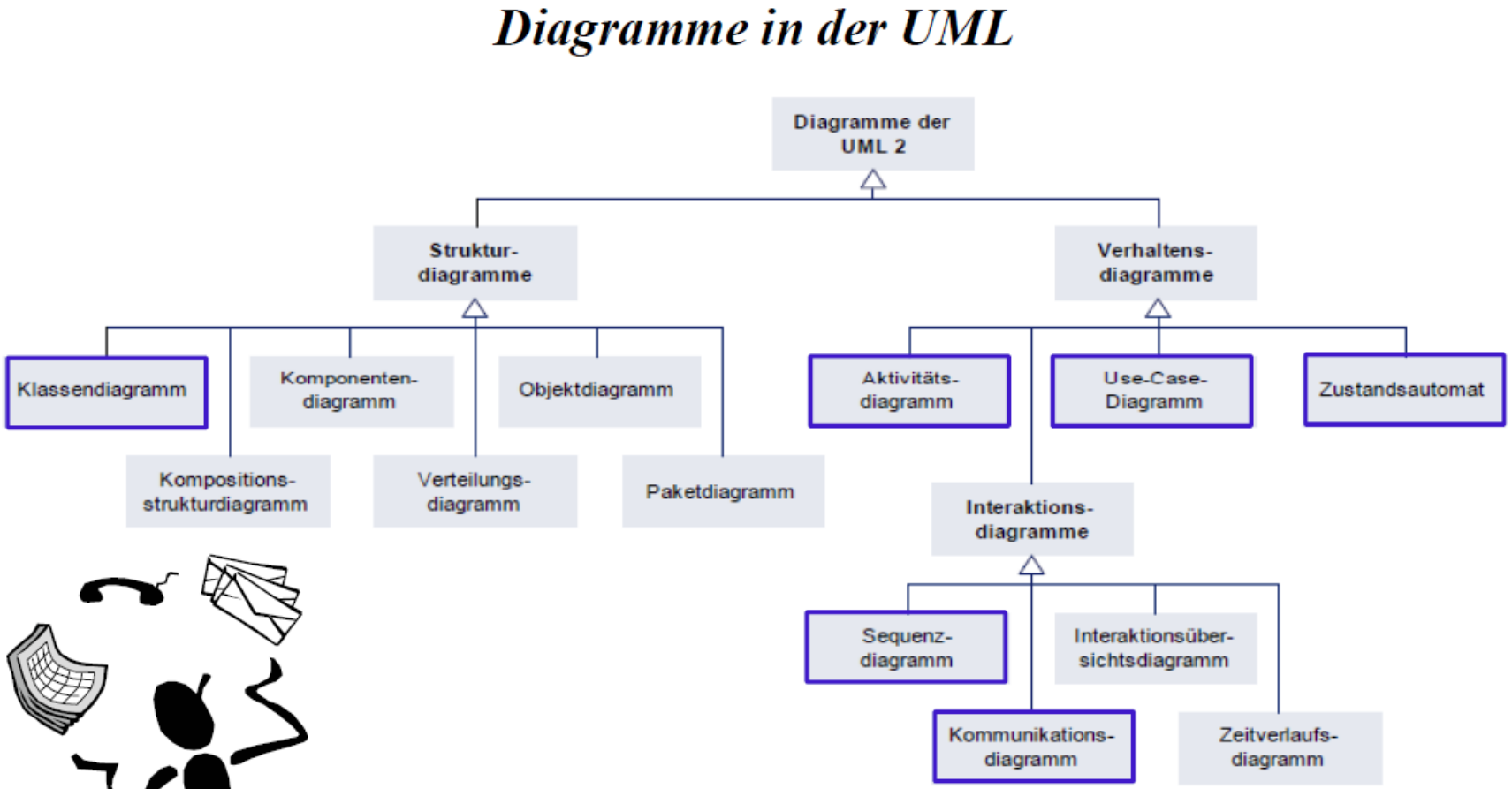
- **Reflexive Assoziationen** bestehen zwischen Objekten derselben Klasse z. B. das OO-Modells eines Stammbaums



- **Aggregationen** sind Klassen die aus mehreren Objekten anderer Klassen bestehen

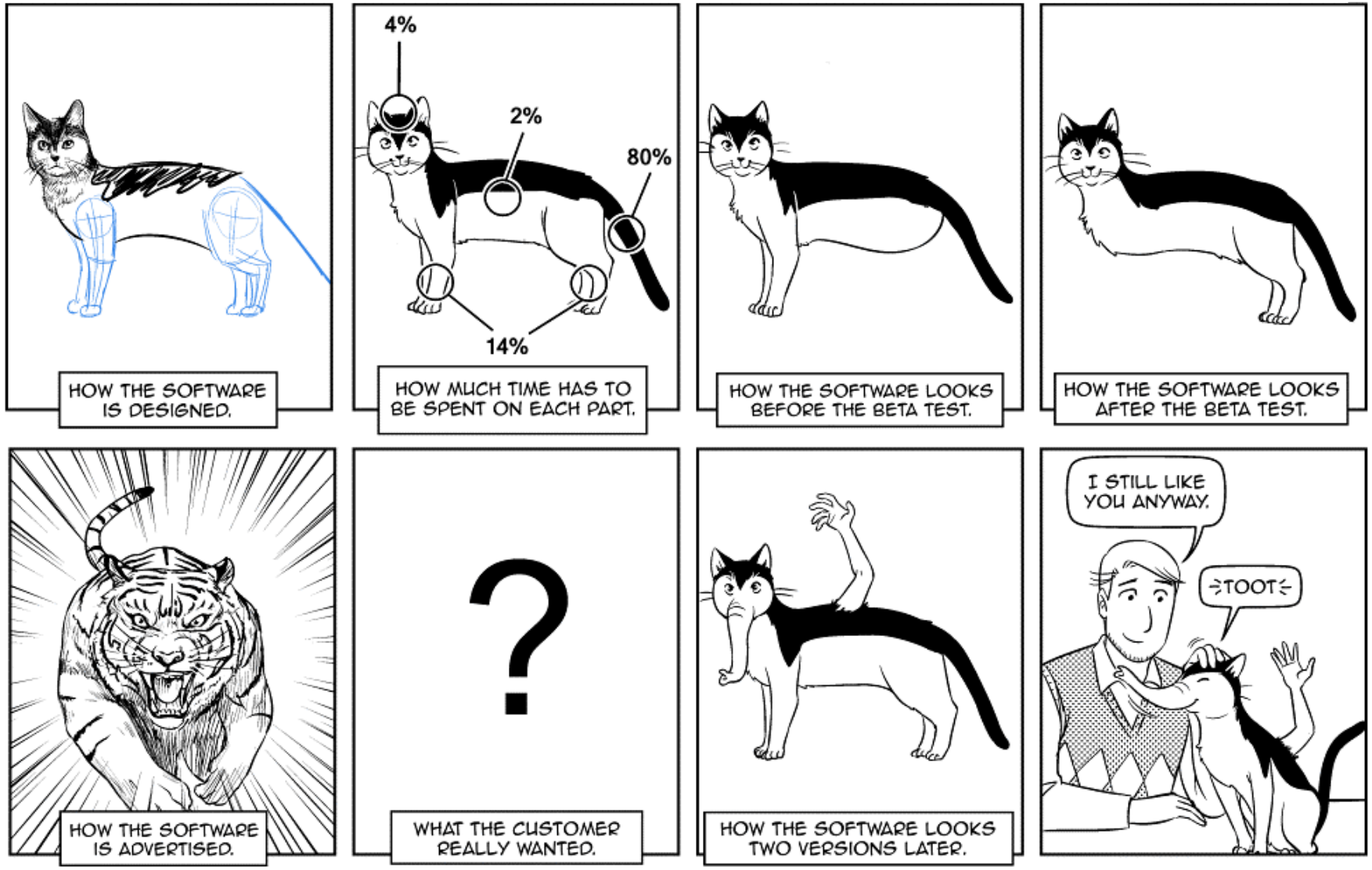


Weiterführendes: *Unified Modeling Language*

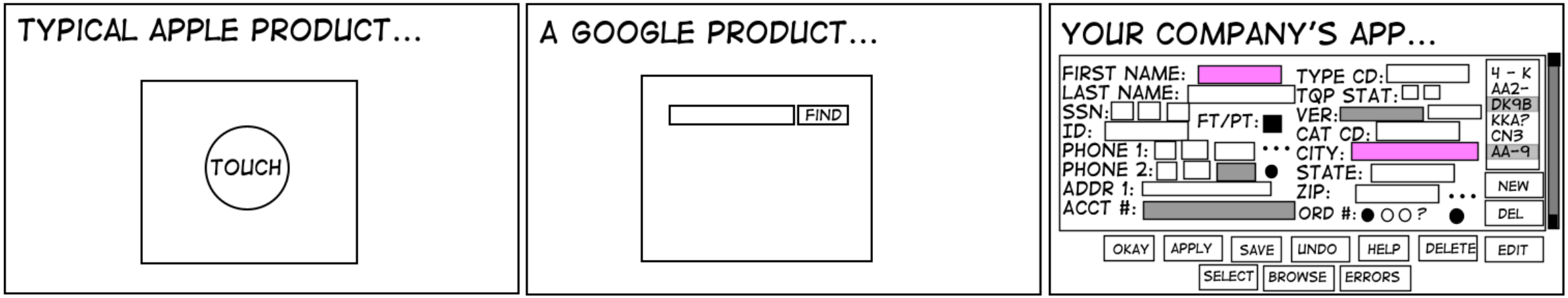


Quelle: Jeckle et al

Objektorientierte Modellierung



Grafische Benutzeroberflächen



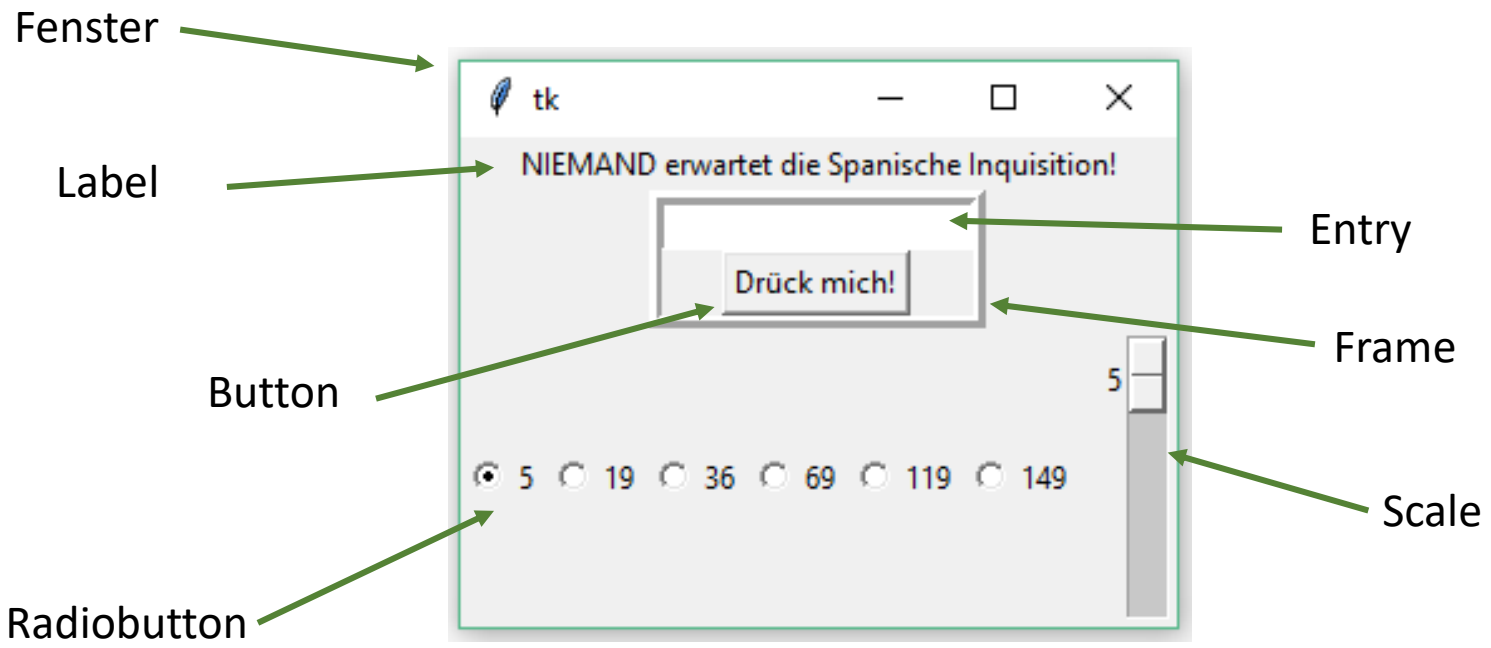
Grafische Benutzeroberflächen

- Ein interaktives Programm erlaubt die Kommunikation zwischen Mensch und Computer über eine Benutzungsoberfläche

- Bisher:
  - Kommunikation ist *textbasiert* (interaktives Konsolenfenster)
  - Kommunikation ist *synchronisiert*, d. h. zeitliche Abfolge von Ein- und Ausgabe ist genau festgelegt
- Graphical User Interface (GUI):
  - Kommunikation ist *multimedial*, d. h. die Benutzungsoberfläche ist ein grafisches Fenster mit verschiedenen Komponenten
  - Kommunikation ist *asynchron*, d. h. der Verlauf ist nicht immer vorhersehbar

Grafische Benutzeroberflächen

- Ein GUI besteht aus einem Fenster und *Steuerelementen* (*Widgets*)



Grafische Benutzeroberflächen

- Drei Schritte der GUI-Programmierung:

1. Definition der grafischen Element des Programms
2. Sinnvolle Anordnung der Elemente im Fenster in ein *Layout*
3. Definition der Interaktivität (Elemente mit Funktionalität verknüpfen)

GUI Toolkits für Python

- **Tkinter** (Tk interface) ist die Standardbibliothek für Oberflächenprogrammierung in Python

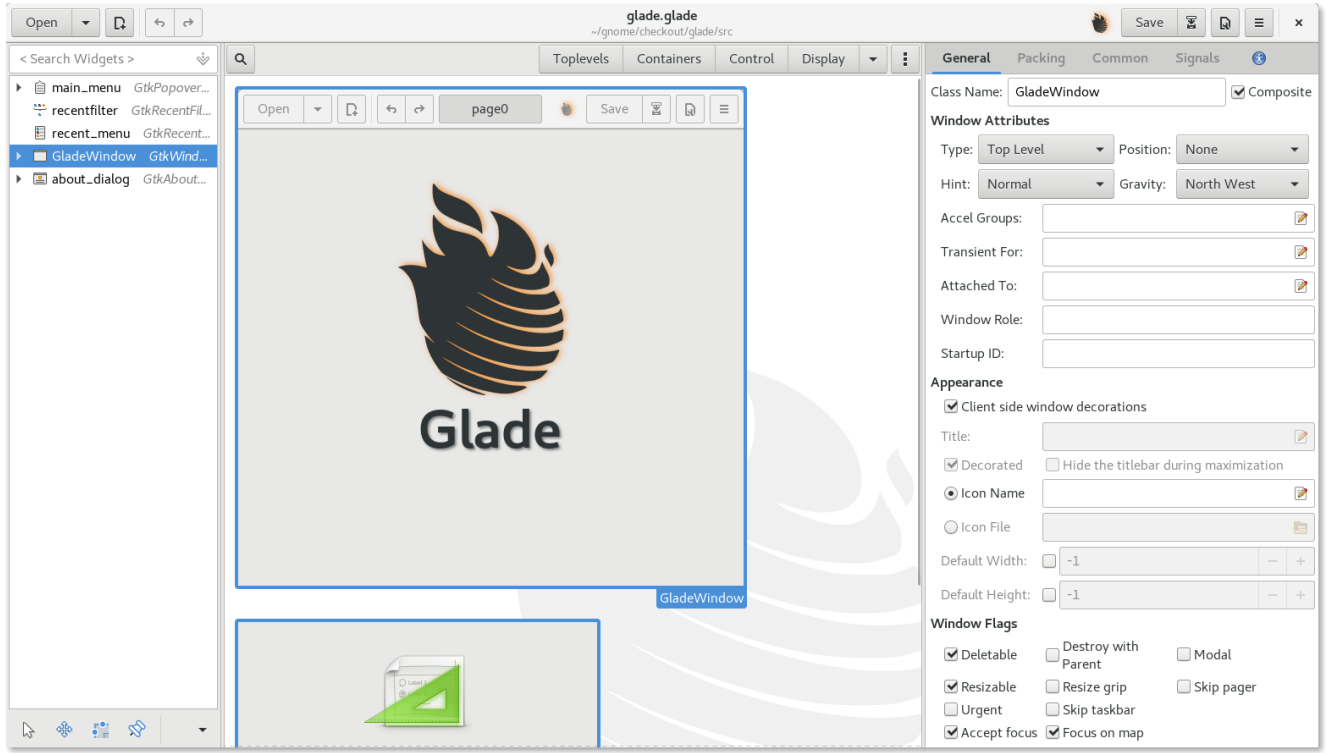
- **PyGtk** (GIMP Toolkit) eigentlich für das Grafikprogramm GIMP entwickelt; heute aber allgemein unter Linuxanwendern sehr beliebt

- **PyQt** ist ein umfassendes Framework der Firma *Trolltech*; sehr mächtig und weitverbreitet da es auch viele GUI-fremde Funktionalitäten enthält

FRIEDRICH-SCHILLER-
UNIVERSITÄT
JENA

GUI IDE für Python

- **Glade** ist ein *Rapid Application Development* Programm zur einfachen Erstellung von Benutzungsoberflächen basierend auf dem Gtk-Toolkit

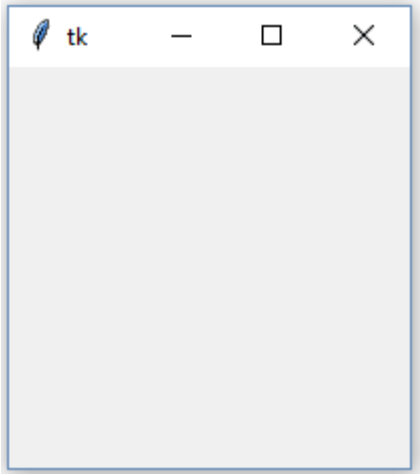


Tkinter

- Tkinter ist ein *Wrapper* für das GUI-Toolkit Tk für Python
- Es hat ein Modul für ein direktes Tk-Interface
- Für jedes vorhandene Tk-Widget gibt es eine eigene Klassendefinition
- Tk ist (so gut wie) Plattform unabhängig

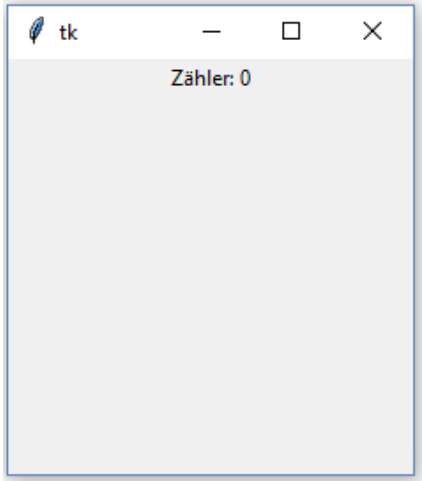
Tkinter – ein kleines Beispiel

```
from tkinter import *
fenster = Tk()              #erzeugen eines leeren Tk-
                            #Fenster Objekts
fenster.mainloop()          #aktivieren des Tk-Fensters
```



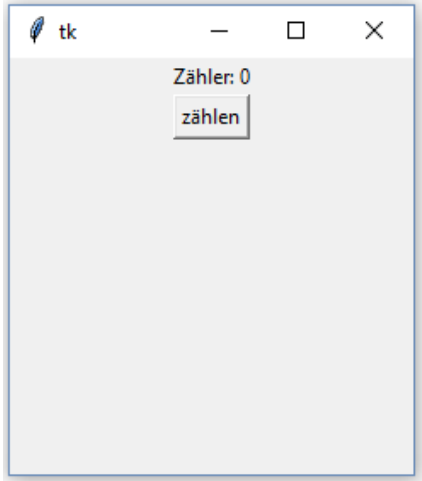
Tkinter – ein kleines Beispiel

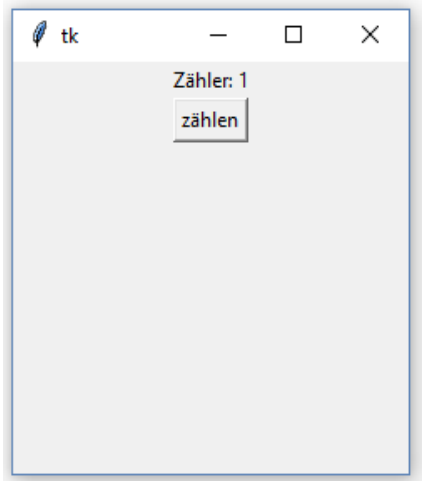
```
from tkinter import *
fenster = Tk()
#Definition eines Label-Widgets in fenster
fenster.label = Label(master=fenster,
                      text='Zähler: 0')
fenster.label.pack()      #aktivieren des Widgets
fenster.mainloop()
```



Tkinter – ein kleines Beispiel

```
from tkinter import *
def zählen():
    global i
    i += 1
    fenster.label.config(text=f‘Zähler: {i}‘)

i = 0
fenster = Tk()
fenster.label = Label(master=fenster,
                      text=‘Zähler: 0‘)
fenster.label.pack()
#Definition eines Button-Widgets in fenster
fenster.button = Button(master=fenster,
                        text=‘zählen‘,
                        command=zählen)
fenster.button.pack()
fenster.mainloop()
```



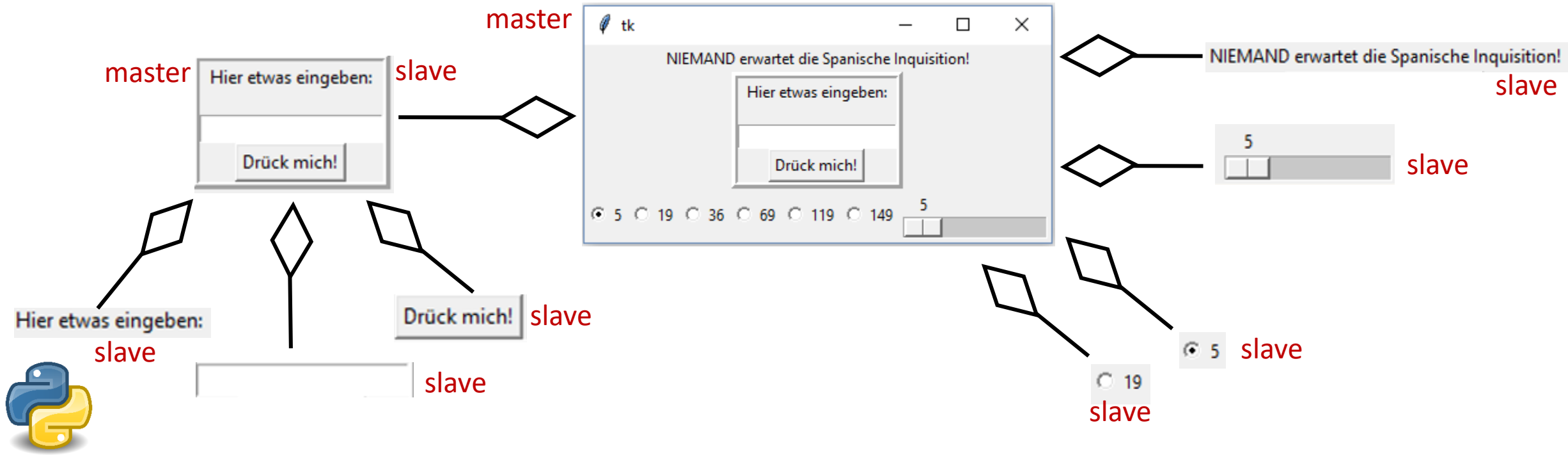


Einfache Widgets

- Widgets sind die Komponenten aus denen eine GUI aufgebaut ist
- Programmtechnisch gesehen sind es Instanzen von Klassen aus dem tkinter Modul
- tkinter stellt folgende Standard-Widgets zur Verfügung:
  - Button
  - Canvas
  - Checkbutton
  - Entry
  - Frame
  - Label
  - Listbox
  - Menu
  - Menubutton
  - Radiobutton
  - Scale
  - Scrollbar
  - Text
  - Toplevel
  - Spinbox
  - PanedWindow
  - LabelFrame
  - OptionMenu

Die Master-Slave-Hierarchie

- Jede GUI hat eine feste Struktur, ähnliche wie ein Aggregat von Objekten
- Jedes Widget-Objekt gehört zu genau einem Fenster- oder Frame-Objekt (wobei diese auch wieder Widget-Objekte sind)



Die Master-Slave-Hierarchie

- Immer, wenn ein Widget-Objekt instanziiert wird, muss ihm ein Master-Widget zugeordnet werden
- Das erste Argument eines Widget-Objekts ist immer die Referenz zum Master-Widget
- Die Master-Slave-Hierarchie ist losgelöst von der Objekt-Attribut-Beziehung

```
from tkinter import *
fenster = Tk()
fenster.label = Label(master=fenster, text='Label 1')

label2 = Label(fenster, text='Label 2')
```

Optionen der Widgets

- Widgets sind Objekte mit (vorgegebenen) Attributen, welche sich auf ihr Erscheinungsbild beziehen können → *Widget-Optionen*

- Bei der Instanziierung können Attribute gleich in der Form *option=wert* gesetzt werden, wobei die Reihenfolge beliebig ist

- Es gibt eine Vielzahl an Optionen um so ziemlich alles am Aussehen und Verhalten eines Widgets zuändern

```
from tkinter import *
fenster = Tk()
fenster.label = Label(master=fenster, text='Label 1',
                      font=('Comic Sans MS', 14), fg = 'green')
```

Standardoptionen

Option	Erklärung
bd, borderwidth	Integer, der die Breite des Rahmens in Pixel angibt
width	Breite des Widgets (horizontal) in Pixel
height	Höhe des Widgets (vertikal) in Pixel
bg, background	Hintergrundfarbe
fg, foreground	Vordergrundfarbe (Textfarbe)
image	Referenz eines Bild-Objekts, das auf dem Widget zu sehen sein soll
justify	Ausrichten von Textzeilen auf dem Widget
padx	Leerer Raum in Pixel rechts und links vom Widget oder Text
pady	Leerer Raum in Pixel über und unter vom Widget oder Text
relief	Form des Rahmens (*flat, ridge, groove, raised, sunken*)
text	Beschriftung des Widgets (z. B. Button oder Label)
textvariable	Ein Objekt der Klasse *StringVar*, das den (variablen) Text des Widgets enthält
underline	Zum Unterstreichen des Textes auf dem Widget

Optionen nachträglich ändern

- Widget-Optionen können auch mithilfe der *config()*-Methode nachträglich geändert werden

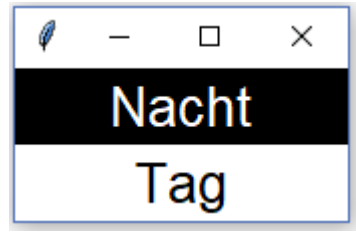
```
from tkinter import *
fenster = Tk()
fenster.label = Label(text='Label 1', master=fenster,
                      font=('Comic Sans MS', 14), fg = 'green')

fenster.label.config(text='Neuer Text', justify=CENTER)
```

Font Optionen

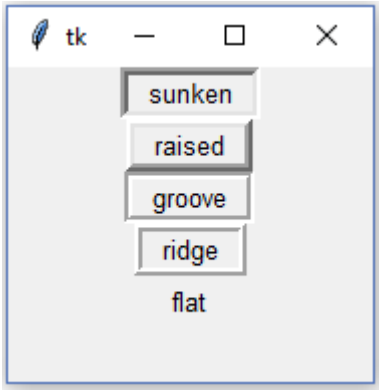
```
from tkinter import *
f = Tk()
l1 = Label(f, text='Verdana Größe 40', font=('Verdana', 40)).pack()
l2 = Label(f, text='Times Größe 30 Bold', font=('Times', 30, 'bold')).pack()
l3 = Label(f, text='Comic Sans Größe 20', font=('Comic Sans MS', 20, 'underline')).pack()
l4 = Label(f, text='Courier Größe 10 Italic', font=('Courier', 10, 'italic')).pack()
```

Farben und Rahmen Optionen

```
from tkinter import *
f = Tk()
labelNacht = Label(f, text='Nacht', font=('Arial', 20),
                   fg='white', bg='black', width=10).pack()
labelTag = Label(f, text='Tag', font=('Arial', 20),
                  fg='#000000', bg='#ffffff', width=10).pack()
```

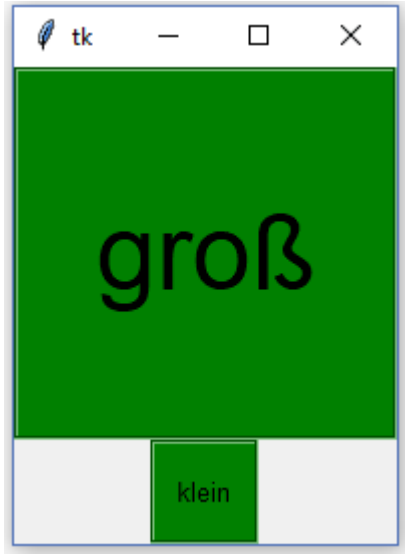



```
from tkinter import *
f = Tk()
rahmentypen=[SUNKEN, RAISED, GROOVE, RIDGE, FLAT]
for rahmen in rahmentypen:
    label = Label(f, text=str(rahmen), font=('Arial', 10),
              bd=4, relief=rahmen, padx=10).pack()
```

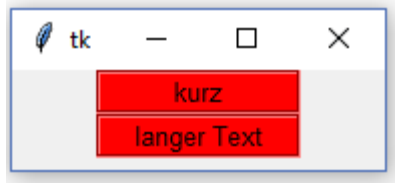


Widget Größe

```
from tkinter import *
f = Tk()
l1 = Label(f, text='groß', font=('Arial', 40),
           relief='groove', bg='green').pack()

l2 = Label(f, text='klein', font=('Arial', 10),
           relief='groove', bg='green).pack()
```




```
from tkinter import *
f = Tk()
l1 = Label(f, text='kurz', font=('Arial',10), width=12,
           relief='groove', bg='red').pack()
l2 = Label(f, text='langer Text', font=('Arial',10),
           width=12, relief='groove', bg='red').pack()
```



Leerraum um Text

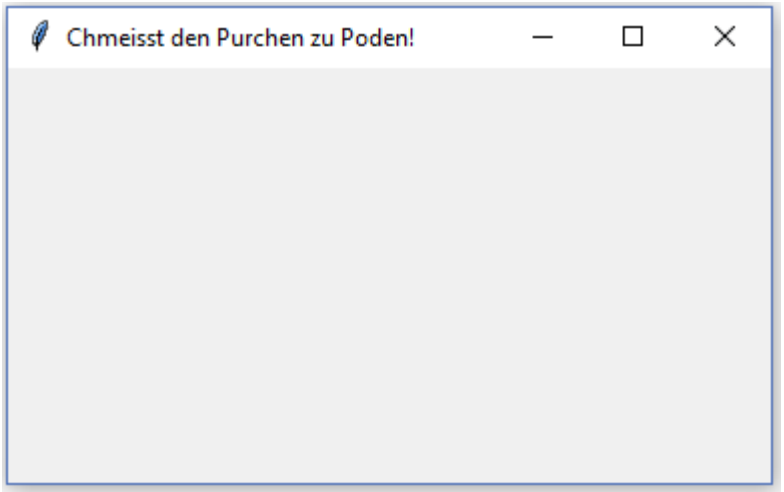
```
from tkinter import *
f = Tk()
l1 = Label(f, text='Eins', bg='green', font=('Arial', 12)).pack()
l2 = Label(f, text='Zwei', bg='red', font=('Arial', 12),
           pady=40).pack()
l3 = Label(f, text='Drei', bg='yellow', font=('Arial', 12),
           padx=40).pack()
```

Die Klasse Tk

- Programme mit GUI laufen immer in einem Anwendungsfenster

- In Tkinter ist dies ein Objekt der Klasse Tk, welches immer das oberste Objekt in der Master-Slave-Hierarchie ist

```
from tkinter import *
fenster = Tk()
fenster.title('Chmeisst den Purchen zu Poden!')
```



Die Klasse Button

- Eine einfache Schaltfläche, welche durch anklicken eine Aktion auslöst

- Ihre wichtigste Widget-Option ist *command* durch die man eine Funktion an den Button binden kann

```
from tkinter import *
def func():
    pass

fenster = Tk()
fenster.button = Button(master=fenster,
                        text='zählen',
                        command=func)
fenster.button.pack()
fenster.mainloop()
```

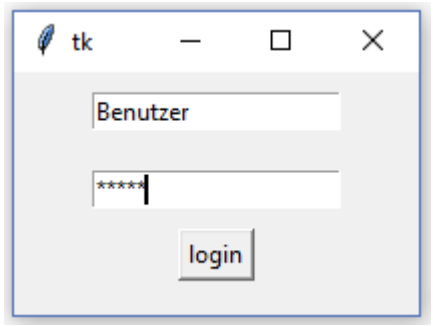
Die Klasse Label

- Zum anzeigen von ein- oder mehrzeiligen Text
- Änderbar über die *config()*-Methode oder über die *textvariable* Widget-Option
- An die *textvariable* können **Kontrollvariablen** an Widgets gebunden werden:
  - DoubleVar()
  - IntVar()
  - StringVar()
- Ändert man den Inhalt einer Kontrollvariable, ändert sich automatisch jedes Widget das diese eingebunden haben

Die Klasse Entry

- Zur Eingabe einzelner Textzeilen

```
from tkinter import *
def einloggen():
    global f
    user = f.e1.get()
    password = f.e2.get()

f = Tk()
f.e1 = Entry(f)
f.e2 = Entry(f, show='*')
f.b = Button(f, text='login', command=einloggen)
f.e1.pack(padx=10, pady=10)
f.e2.pack(padx=10, pady=10)
f.b.pack()
```

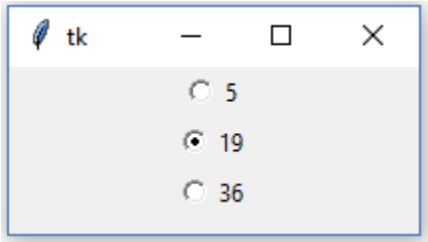


Die Klasse Radiobutton

- Schaltflächen für eine „Eins-aus-n-Wahl“

- Radiobuttons die zu einer Gruppe gehören, teilen sich eine gemeinsame Kontrollvariable, über den Parameter *variable* und haben einen *value* der bei Auswahl an diese übergeben wird

```
from tkinter import *

f = Tk()
ctr = IntVar()
f.r1 = Radiobutton(f, text='5', value=5, variable=ctr)
f.r2 = Radiobutton(f, text='19', value=19, variable=ctr)
f.r3 = Radiobutton(f, text='36', value=36, variable=ctr,
                   command=func)
for r in [f.r1, f.r2, f.r3]:
    r.pack()
```

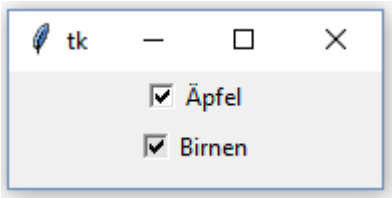


Die Klasse Checkbutton

- Schaltflächen für eine Mehrfachauswahl

- Jeder Checkbutton befindet sich unabhängig von allen anderen entweder im Zustand *Ein* oder *Aus*, wobei beiden Zuständen ein eigener Wert zugewiesen werden kann

```
f = Tk()
apfel = IntVar()
birne = StringVar()
f.c1 = Checkbutton(f, text='Äpfel', offvalue=0,
                   onvalue=1, variable=apfel)
f.c2 = Checkbutton(f, text='Birnen',
                   offvalue='off', onvalue='on',
                   variable=birne, command=func)
f.c1.pack()
f.c2.pack()
```



Die Klasse Scale

- Darstellung von Schiebereglern um eine Kontrollvariable in einem bestimmten Wertebereich zu verändern

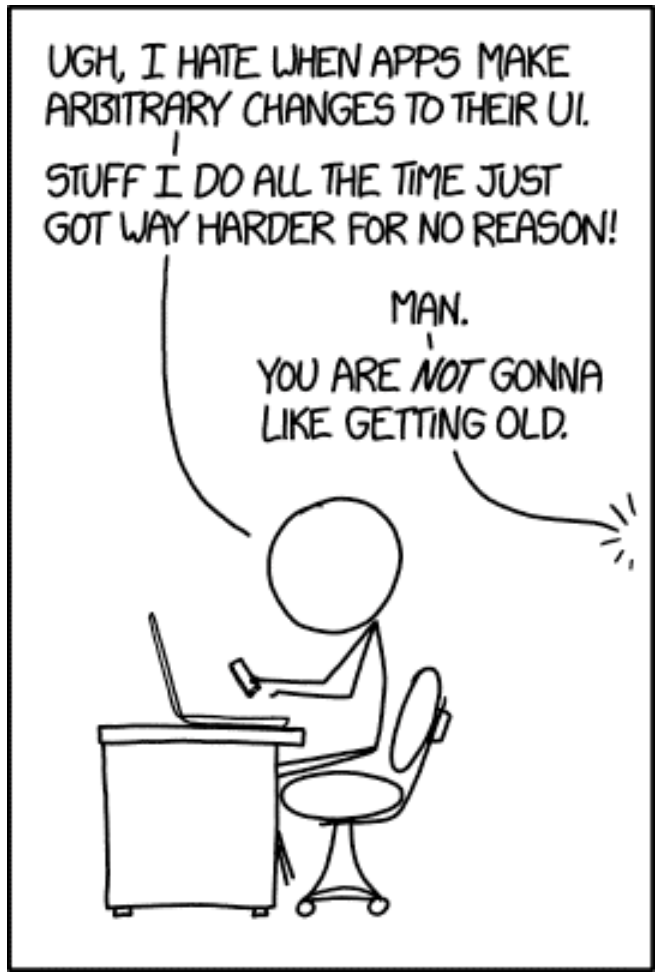
- Der Wertebereich wird durch die Optionen *from_* und *to* definiert

```
def func():
    global f
    wert2 = float(f.s2.get())

f = Tk()
wert1 = IntVar()
f.s1 = Scale(f, from_=5, to=149, variable=wert1)
f.s2 = Scale(f, from_=5, to=149, length=300,
             orient=HORIZONTAL, command=func)
f.s1.pack()
f.s2.pack()
```

Die Klasse Frame

- Frames sind Behälter für Widgets mit eigenem, unabhängigem Layout
- Sie werden dazu verwendet um komplexe GUIs modular aufzubauen
- Per default ist der Rahmen eines Frames auf *FLAT* eingestellt, so dass man diesen nicht sieht

Layout


UGH, I HATE WHEN APPS MAKE ARBITRARY CHANGES TO THEIR UI.
STUFF I DO ALL THE TIME JUST GOT WAY HARDER FOR NO REASON!
MAN.
YOU ARE NOT GONNA LIKE GETTING OLD.

Layout

- Durch die Master-Slave-Hierarchie ist bereits eine grobe Struktur der Widgets in einer Benutzeroberfläche definiert
- Allerdings ist meist noch keine genaue oder gewünschte Anordnung der einzelnen Widgets vorhanden
- Tkinter nutzt dafür verschiedene *Layoutmanager*, welche das System zur Laufzeit versucht so gut wie möglich umzusetzen
- Es gibt drei verschiedene Layoutmanager: *pack, grid* und *place*

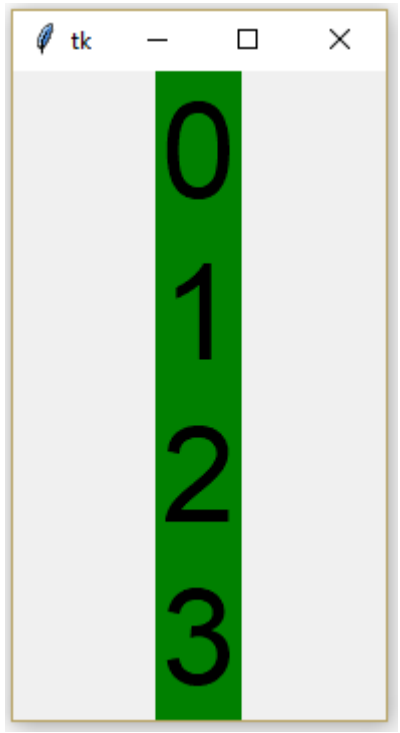
Der Layoutmanager *Packer*

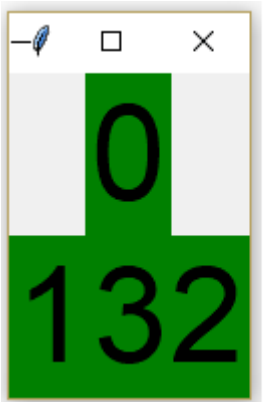
- Mit dem Packer kann man Widgets untereinander und nebeneinander im dazugehörigem master platzieren
- Jedes Widget besitzt die Methode *pack()* um den Packer aufzurufen und verschiedene Optionen um die Positionierung an bereits vorhandene Widgets festzulegen

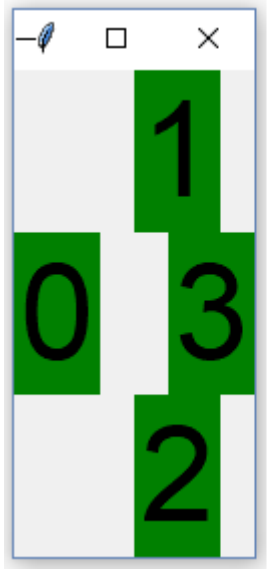
Option	Erklärung
anchor	Mögliche Werte: CENTER, E, N, NE, NW, S, SE, SW; Widget wird in eine Ecke oder mittig an eine der Seiten des Frames platziert (entspricht den Himmelsrichtungen)
side	Mögliche Werte: LEFT, RIGHT, TOP, BOTTOM
expand	expand=0: Die Größe des Widgets ändert sich nicht, wenn das Fenster vergrößert wird; expand=1: Die Größe passt sich dem Fenster an
fill	fill=X oder Y: Das Widget wird in waagerechter oder senkrechter Richtung den Ausmaßen des Masters angepasst; fill=both: Anpassen der Widgetgröße in beide Richtungen fill=none: Das Widget behält seine Größe unabhängig von seinem Master

Der Layoutmanager *Packer*

```
f = Tk()
labels=[]
for i in range(4):
    labels.append(Label(f, text=str(i),
       bg='green', font=('Arial', 50)))

labels[0].pack(side=TOP)
labels[1].pack(side=TOP)
labels[2].pack(side=TOP)
labels[3].pack(side=TOP)

labels[0].pack(side=TOP)
labels[1].pack(side=LEFT)
labels[2].pack(side=RIGHT)
labels[3].pack(side=LEFT)

labels[0].pack(side=LEFT)
labels[1].pack(side=TOP)
labels[2].pack(side=BOTTOM)
labels[3].pack(side=RIGHT)
```

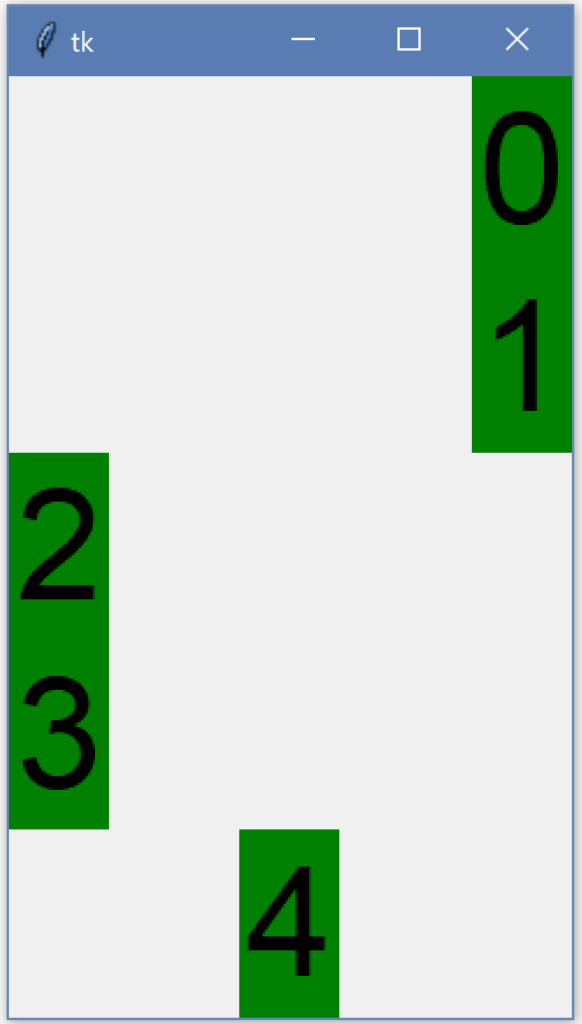


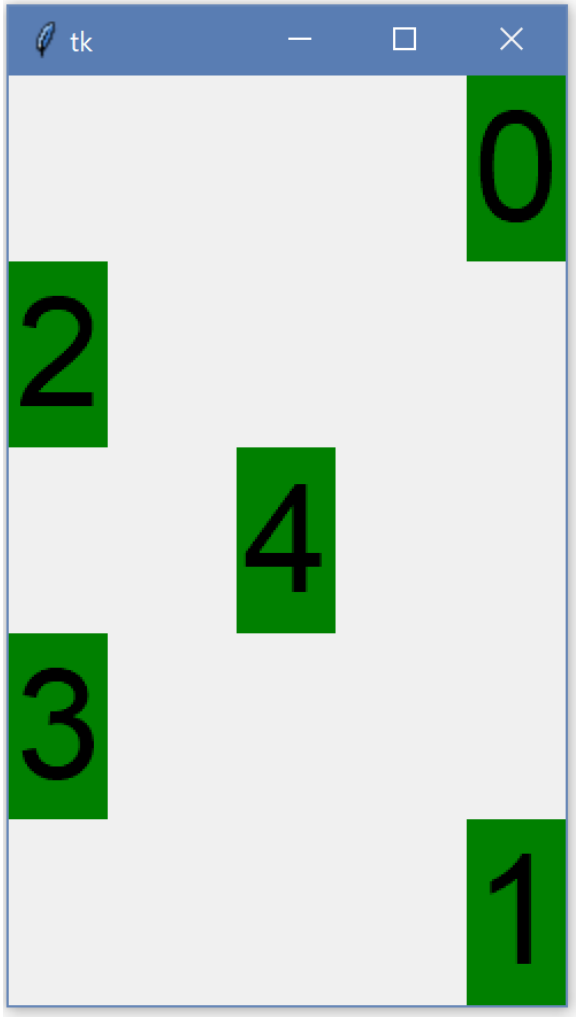




Der Layoutmanager *Packer*

```
from tkinter import *
f = Tk()
labels=[]
for i in range(5):
    labels.append(Label(f, text=str(i),
       bg='green', font=('Arial', 50)))

labels[0].pack(anchor=NE)
labels[1].pack(anchor=SE)
labels[2].pack(anchor=NW)
labels[3].pack(anchor=SW)
labels[4].pack(anchor=CENTER)

labels[0].pack(anchor=NE)
labels[2].pack(anchor=NW)
labels[4].pack(anchor=CENTER)
labels[3].pack(anchor=SW)
labels[1].pack(anchor=SE)
```





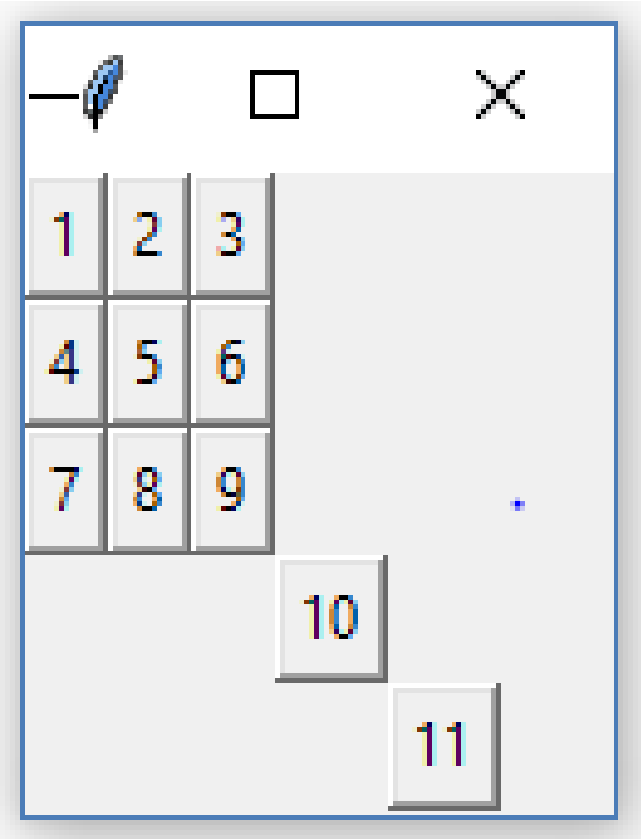
Der Layoutmanager *Grider*

- Mit dem Grider lässt sich ein Raster aus gleichförmigen Zellen definieren, in welche dann Widgets platziert werden können
- Jedes Widget besitzt die Methode *grid()* um den Grider aufzurufen

Option	Erklärung
column	Die Nummer der Spalte, in der das Widget angezeigt werden soll.
row	Die Nummer der Zeile, in der das Widget angezeigt werden soll.
columnspan	Normalerweise befindet sich das Widget in genau einer Zelle, wenn es sich aber über *n* benachbarte Zellen erstrecken soll, wird die Option columnspan=n gestetzt.
rowspan	Das Widget erstreckt sich über *n* Zellen der Spalte.

Der Layoutmanager *Grider*

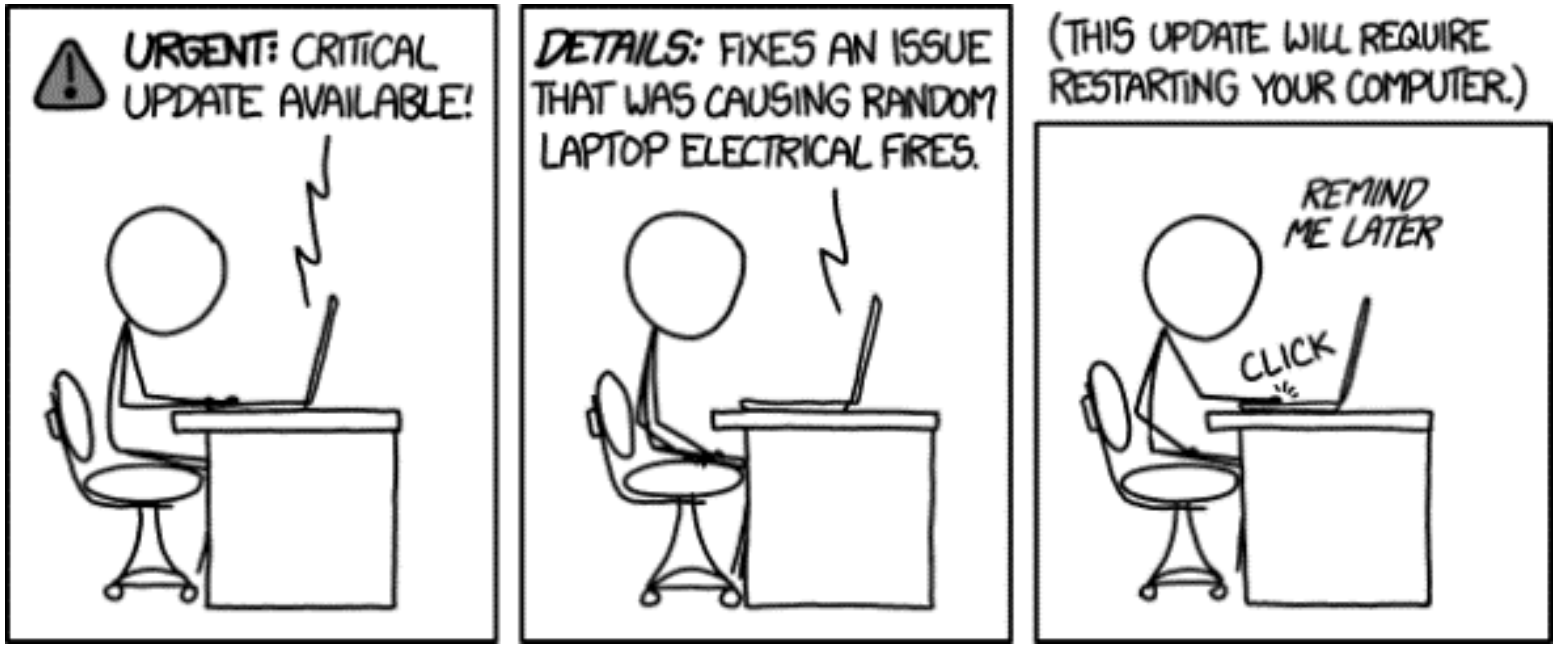
```
f = Tk()
t=[(0,0,'1'), (1,0,'2'), (2,0,'3'),
   (0,1,'4'), (1,1,'5'), (2,1,'6'),
   (0,2,'7'), (1,2,'8'), (2,2,'9')]

for (i,j,z) in t:
    Button(f, text=z).grid(column=i, row=j)

Button(f, text='10').grid(column=4, row=4)
Button(f, text='11').grid(column=19, row=19)
```



Der Layoutmanager *Placer*

- Mit dem Placer lassen sich Widgets absoluten Positionen zuweisen, dabei entsprechen die X/Y-Koordinaten den Pixeln von der linken oberen Ecke des Masters
- Jedes Widget besitzt die Methode *place(x, y)* um den Placer aufzurufen

Event-Verarbeitung



Event-Verarbeitung

- Ein Event ist ein Ereignis, das zu einem unbestimmten Zeitpunkt (asynchron) eintritt
- In einem Event-gesteuertem GUI-Programm löst ein Event ein bestimmtes Systemverhalten aus, das in einem *Eventhandler* definiert ist
- In Python werden Events durch *Event-Sequenzen* (Auslöser) beschrieben, und an ein oder mehrere Eventhandler gebunden
- Events sind immer an Widgets gebunden und jedes Widget besitzt Methoden um ein Event an einen Eventhandler zu binden

Event-Sequenzen

- Eine Event-Sequenz ist ein String der ein einzelnes Event spezifiziert

- Es besteht aus einem oder mehreren *Event-Pattern,* die jeweils ein elementares Event beschreiben, dabei sieht das Schema folgendermaßen aus:

<*Modifizierer*- Typ -*Qualifizierer*>

Event-Sequenzen

<*Modifizierer*- Typ -*Qualifizierer*>

Event-Typ	Erklärung
Activate	Ein Widget wechselt in den Zustand aktiv
Button	Eine Maustaste wird gedrückt
ButtonRelease	Eine Maustaste wird losgelassen
Configure	Die Größe eines Widgets wurde verändert
Enter	Der Benutzer bewegt den Mauszeiger in ein sichtbares Widget
Expose	Ein zuvor verdecktes Widget wird sichtbar
KeyPress	Eine Taste wird gedrückt
KeyRelease	Eine Taste wird losgelassen
Leave	Der Mauszeiger wird aus einem Widget rausbewegt
Motion	Der Mauszeiger wurde innerhalb eines Widgets bewegt

Event-Sequenzen

- Manche Event-Beschreibungen brauchen einen zusätzlichen Qualifizierer um z. B. die genau gedrückte Taste zu spezifizieren

<*Modifizierer*- Typ -*Qualifizierer*>

Keysym	Keysym_num	Taste
Return	65293	Enter
Up	65362	Pfeiltaste oben
F1	65470	F1-Taste
Space	32	Leertaste
Escape	65307	Escape-Taste

z. B. ‘<KeyPress-Return>‘

Event-Sequenzen

- Mit Qualifizierern können komplexere Events beschrieben werden, wie z. B. das gleichzeitige Drücken mehrerer Tasten

<Modifizierer- Typ *-Qualifizierer>*

Modifizierer	Erklärung
Alt	Alt-Taste wird gedrückt
Any	Der Eventtyp wird generalisiert, z. B. *<Any-KeyPress>* bedeutet das irgendeine Taste betätigt worden ist.
Control	Strg-Taste wird gedrückt
Double	Zwei Ereignisse passieren kurz hintereinander.
Triple	Drei Ereignisse passieren kurz hintereinander
Shift	Shift-Taste wird gedrückt

z. B. ‘<Control-KeyPress-Escape>‘

‘<Any-KeyPress>‘

Programmierung eines Eventhandlers

- Ein Eventhandler ist eine Methode oder Funktion (ohne return Anweisung) die beim auslösen eines Events aufgerufen wird

- Als Argument wird vom System ein Event-Objekt übergeben, welches zum Zeitpunkt des Ereignisses erzeugt worden ist

- Dieses Objekt enthält eine Reihe von Attributen, die ein Event beschreiben und können vom Eventhandler abgefragt werden

Attribut	Erklärung
widget	Referenz zu dem Widget durch welches das Event ausgelöst wurde.
char	Wenn das Event durch eine Tastaturtaste ausgelöst wurde, enthält das Attribut dieses Zeichen.
num	Zahl der Maustaste die gedrückt wurde.

Event-Verarbeitung – Ein Beispiel

```
from tkinter import *
def linksklick(event):
    event.widget.config(bg='green')

def rechtsklick(event):
    event.widget.config(bg='blue')

def doppelklick(event):
    event.widget.config(bg='white')

liste=[(x,y) for x in range(10) for y in range(10)]
fenster = Tk()
for (i,j) in liste:
    l=Label(fenster, width=2, height=1, bg='white')
    l.grid(column=i, row=j)
    l.bind(sequence='<Button-1>', func=linksklick)
    l.bind(sequence='<Button-3>', func=rechtsklick)
    l.bind(sequence='<Double-Button-1>', func=doppelklick)
```

Spezielle Bindemethoden

- Man kann ein Event an alle Widgets einer bestimmten Klasse binden

```
w.bind_class(classname, sequence=event func=f)
```

- Von einem Widget aus können alle Events der Applikation an einen Eventhandler gebunden werden

```
w.bind_all(sequence=event func=f)
```

- Bindungen können auch wieder aufgehoben werden

```
w.unbind(sequence, handler)
```

Grafiken in Tkinter

Grafiken in Tkinter

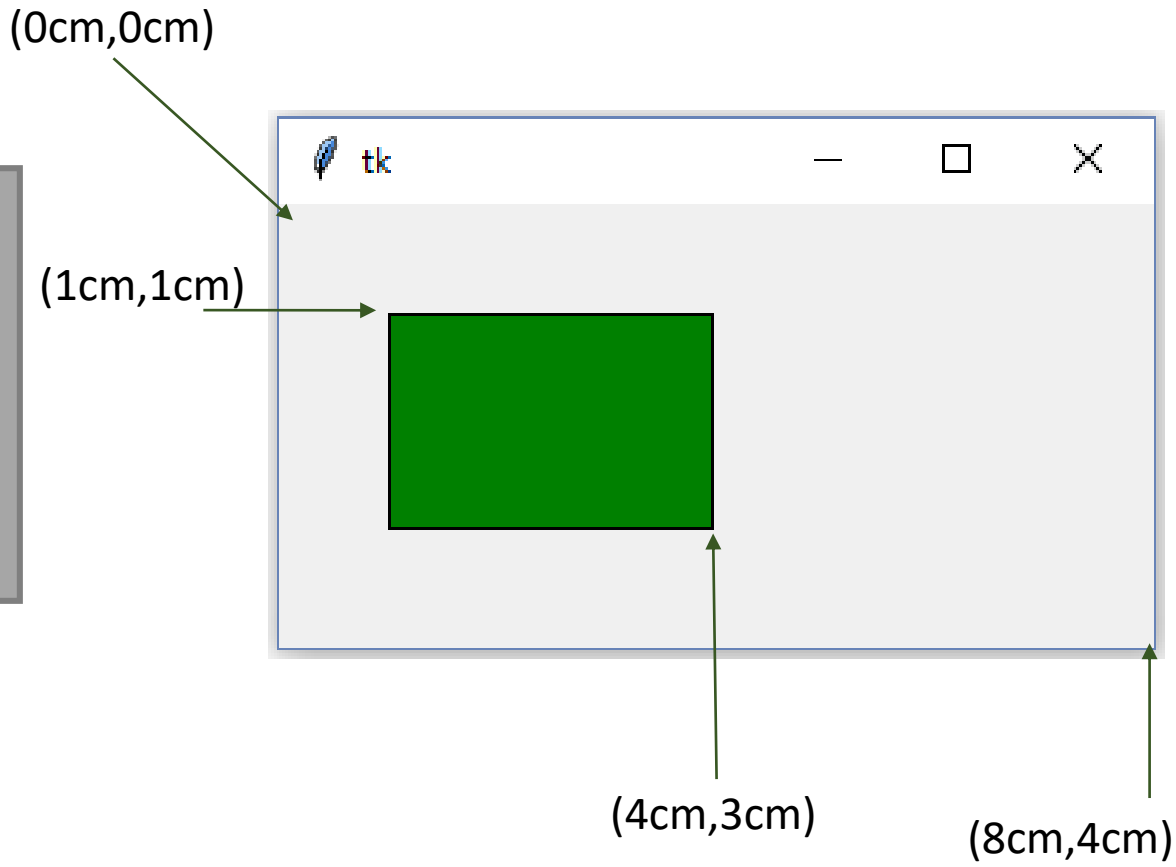
- Die Klasse *canvas* von Tkinter ist ein mächtiges Werkzeug zur Erstellung von Grafiken in GUIs oder Grafikdateien

- Es enthält eine Vielzahl von Methoden um alle möglichen geometrischen Formen, Texte oder Polygone zu erzeugen und zu manipulieren

- Die Klasse *PhotoImage* ermöglicht es Pixelgrafiken einzubinden und zu manipulieren

Die Klasse Canvas

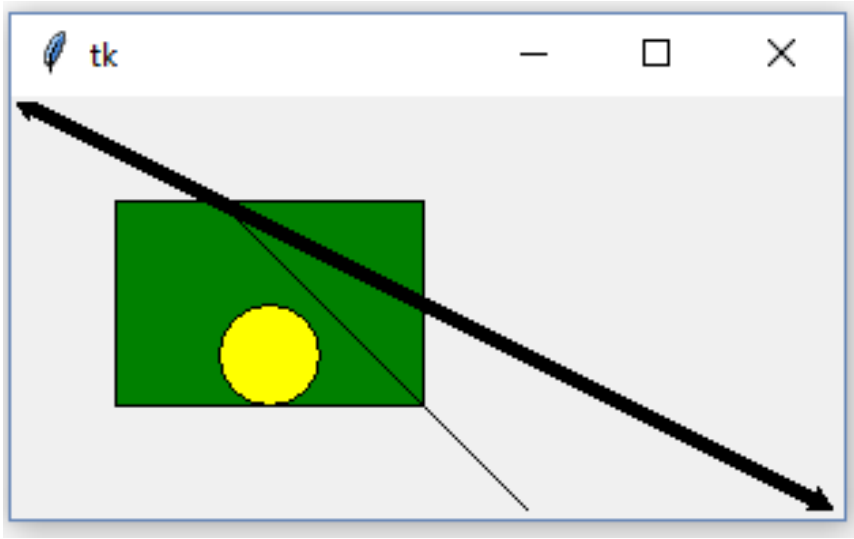
```
from tkinter import *

f = Tk()
c = Canvas(f, width='8c', height='4c')
c.pack()
r = c.create_rectangle('1c','1c','4c','3c',
                       fill='green')
```



Die Klasse Canvas

- Jedes grafische Canvas Elemente ist wieder ein eigenes Objekt mit einer Vielzahl an Attributen und Methoden um z. B. Füllfarbe, Linienstärke usw. zu ändern
- Nachträglich lassen sich diese Eigenschaften mit der *itemconfigure()*-Methode ändern

```
from tkinter import *

f = Tk()
c = Canvas(f, width='8c', height='4c')
c.pack()
r = c.create_rectangle('1c','1c','4c','3c',
                       fill='green')
o = c.create_oval('2c','2c','3c','3c',
                  fill='yellow')
l = c.create_line('5c','4c','2c','1c')
a = c.create_line('0c','0c','8c','4c',
                  width=5, arrow=BOTH)
```



Die Klasse in PhotoImage

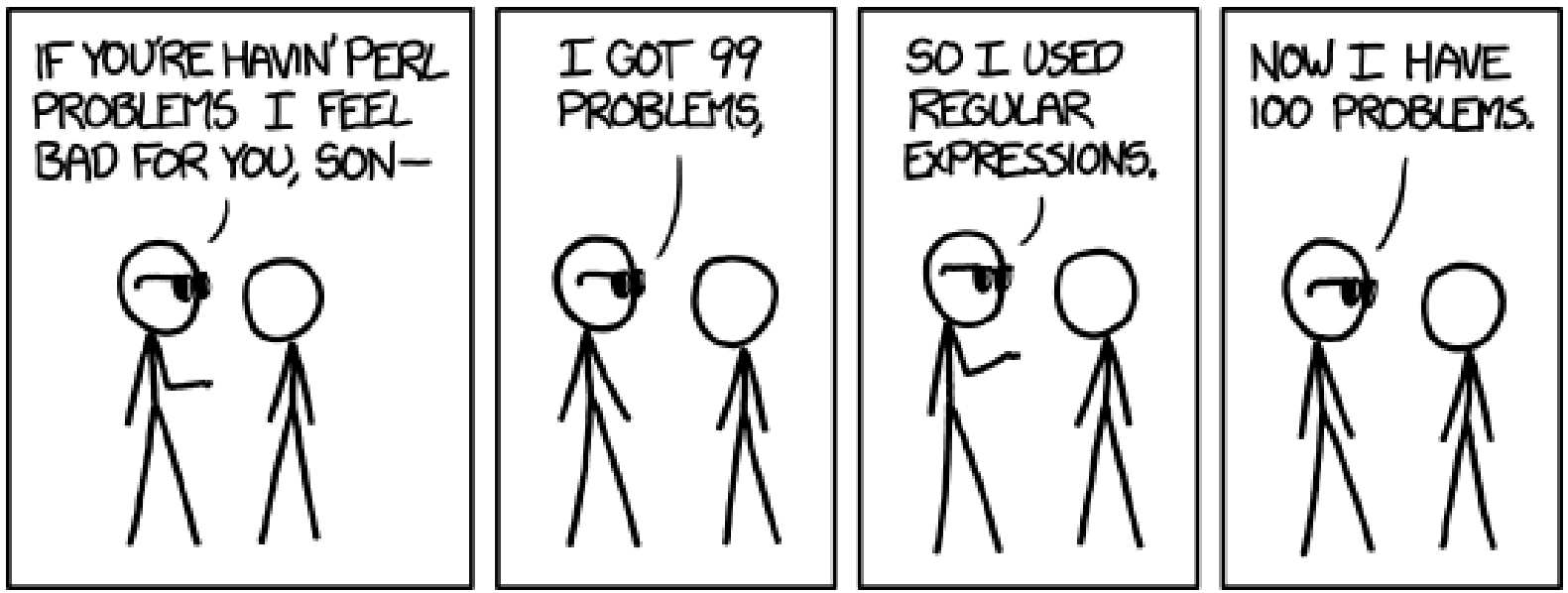
- Mit der Klasse PhotoImage lassen sich Bilder mit 24 Bit pro Pixel darstellen
- Es können schon vorhandene Bilder geladen werden

```
Bild = PhotoImage(file=Pfad)
```

- Oder leere Bilder erzeugt werden

```
Bild = PhotoImage(width=breite, height=höhe)
```

Methode	Erklärung
get(x, y)	Liefert den Farbwert des Pixels an Position X, Y als String.
put(farbe, position)	Setzt einen oder mehrere Pixel mit dem gegebenem Farbwert.
write(pfad)	Speichert das Bild unter dem gegebenem Pfad.


IF YOU'RE HAVIN' PERL PROBLEMS I FEEL BAD FOR YOU, SON—
I GOT 99 PROBLEMS,
SO I USED REGULAR EXPRESSIONS.
NOW I HAVE 100 PROBLEMS.