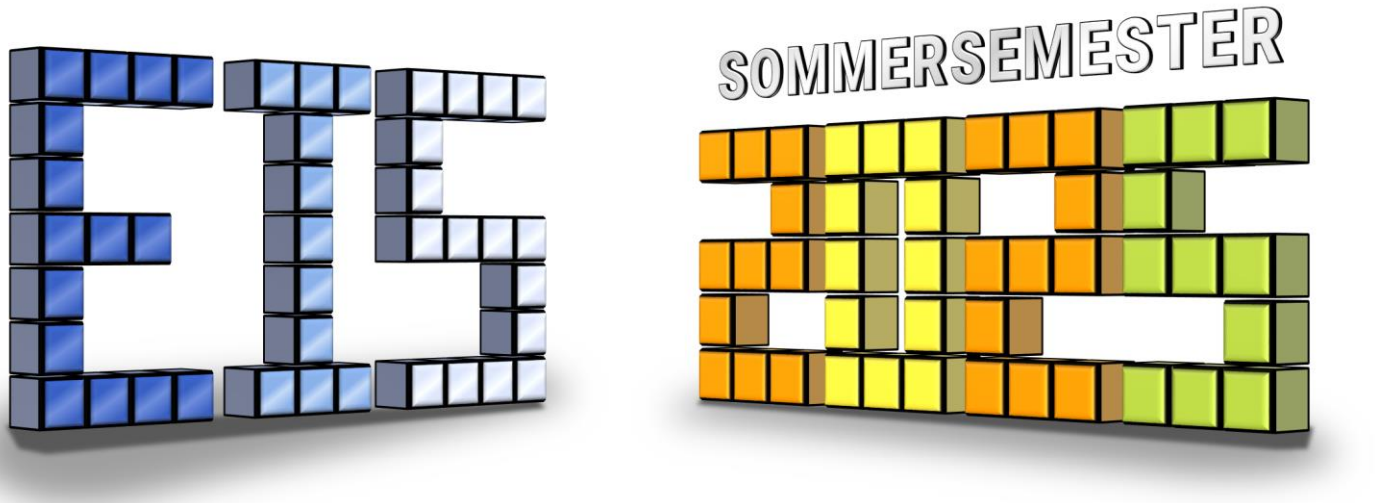
EINFÜHRUNG IN DIE SOFTWAREENTWICKLUNG

Sommersemester 2025



Foliensatz #8

GUIs: Graphische Benutzerschnittstellen

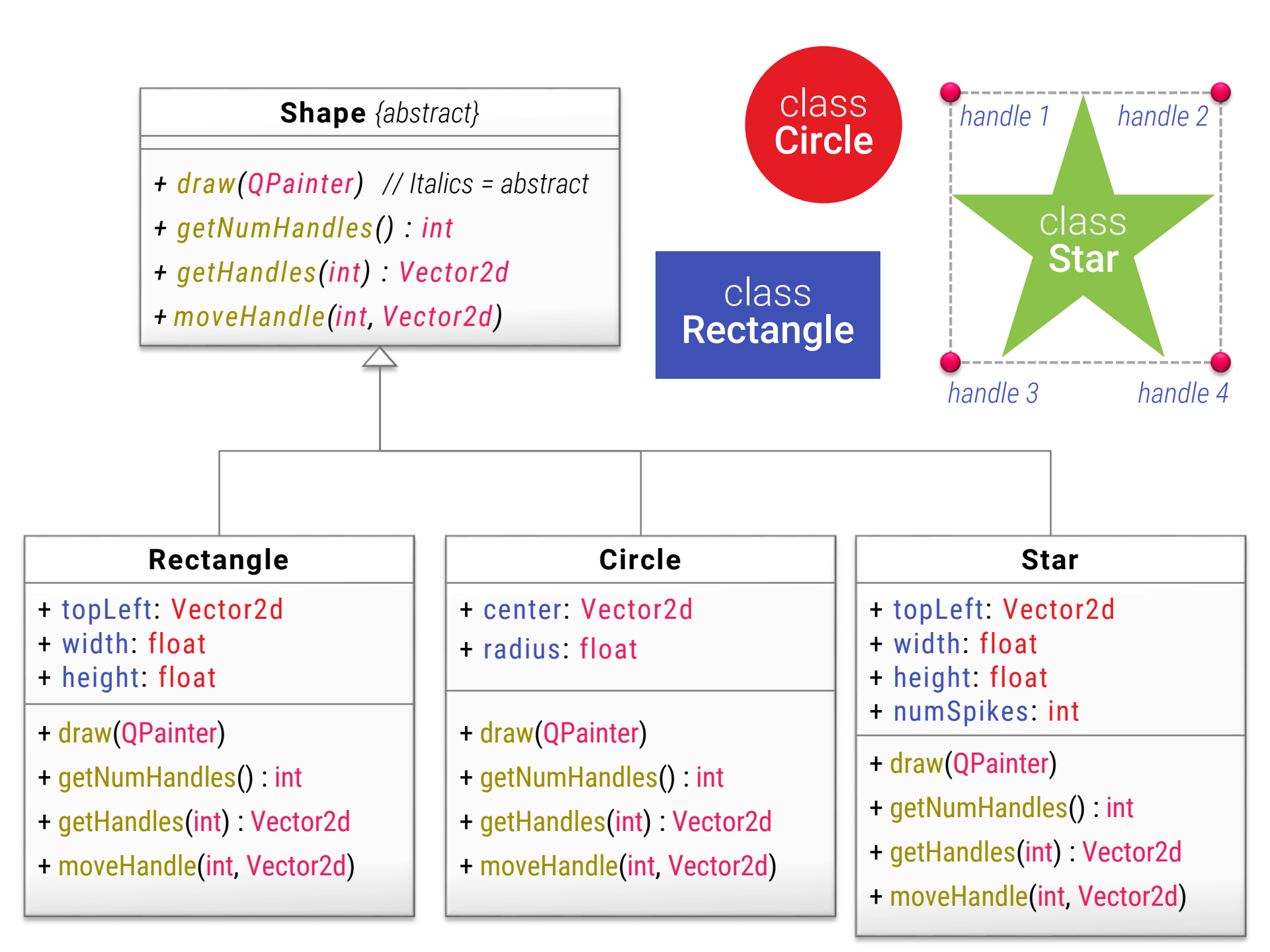
Michael Wand
Institut für Informatik
Michael.Wand@uni-mainz.de

Techniken

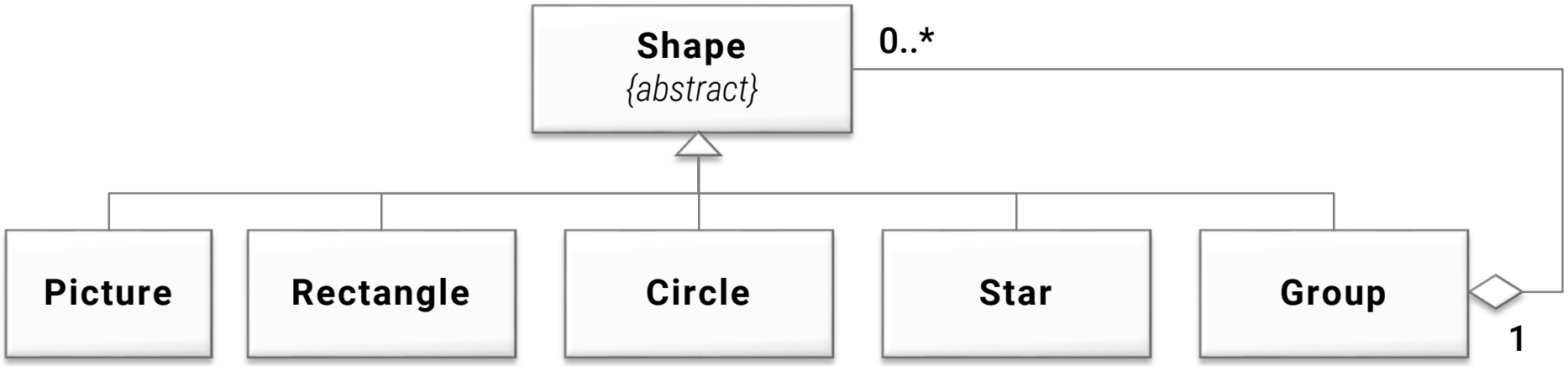
Strukturierung von Programmen

- Prozedural
- Objekt-orientiert
  - Konzepte und Methoden
- GUIs und ereignisorientierte Programmierung
  - OO-Komponenten + Events
  - Richtlinien für gute (G)UIs

Beispielentwurf EIS-Zeichenprogram


Shape {abstract}
+ draw(QPainter) // Italics = abstract
+ getNumHandles() : int
+ getHandles(int) : Vector2d
+ moveHandle(int, Vector2d)
class Circle
class Rectangle
handle 1
handle 2
class Star
handle 3
handle 4
Rectangle
+ topLeft: Vector2d
+ width: float
+ height: float
+ draw(QPainter)
+ getNumHandles() : int
+ getHandles(int) : Vector2d
+ moveHandle(int, Vector2d)
Circle
+ center: Vector2d
+ radius: float
+ draw(QPainter)
+ getNumHandles() : int
+ getHandles(int) : Vector2d
+ moveHandle(int, Vector2d)
Star
+ topLeft: Vector2d
+ width: float
+ height: float
+ numSpikes: int
+ draw(QPainter)
+ getNumHandles() : int
+ getHandles(int) : Vector2d
+ moveHandle(int, Vector2d)

Composition Hierarchy



Graphische Benutzerschnittstellen

(OOP + „Events“)

Oberklasse: Widgets

Oberklasse:
Ein **Widget** ist ein rechteckiges GUI-Element.

Widget

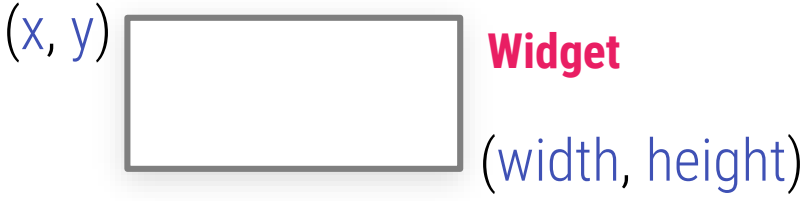
+ x, y, width, height: int

 # draw(p: Painter)

 # mouse_down(x: int, y: int, button_no: int)
 # mouse_move(x: int, y: int)
 # mouse_up()

 # key_pressed(key: Character)

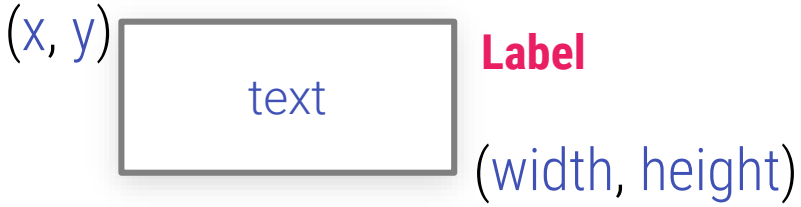
Oberklasse: Widgets



Oberklasse:
Ein **Widget** ist ein rechteckiges GUI-Element.

```
@dataclass
class Widget(ABC):
    # public: Öffentliche Eigenschaften
    x: int = 0          # Geometrie
    y: int = 0          # .
    width: int = 64     # .
    height: int = 64 # .

    # protected: Interface für geerbte Klassen

    # Idee: „Event-oriented programming“
    # „Hollywood architecture“ – we will call you back!

    def _draw(self, g: Painter): # Painter: not real, only loosly inspired by QPainter
        g.fill_rect(self.x, self.y, self.width, self.height, WHITE_COLOR)

    def _mouse_down(self, x: int, y: int, buttonNo: int) -> None: pass
    def _mouse_move(self, x: int, y: int) -> None: pass
    def _mouse_up(self) -> None: pass

    def _key_pressed(self, key: int) -> None: pass
```

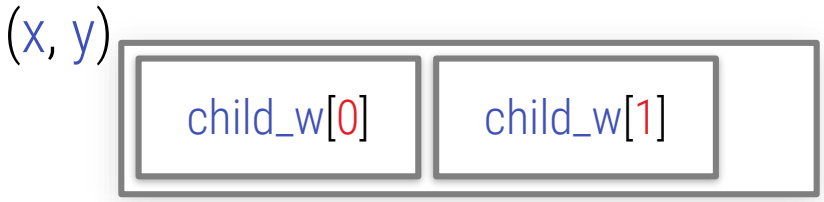
Gui-Framework (schematisch)



Konkrete Klasse:
Ein **Label** ist ein beschriftetes GUI-Element.

```
@dataclass
class Label(Widget):
    # public: zusätzliche öffentliche Eigenschaften
    text: str = ""

    # protected: implementation geerbter Schnittstellen
    def _draw(self, p: Painter) -> None::
        super().draw(p)
        p.draw_text(self.x, self.y, self.text)

    # Für Labels: Der Rest macht nichts (nichts zu tun hier)

    def _mouse_down(self, x: int, y: int, buttonNo: int) -> None: pass
    def _mouse_move(self, x: int, y: int) -> None: pass
    def _mouse_up(self) -> None: pass

    def _key_pressed(self, key: int) -> None: pass
```

Baum von Widgets



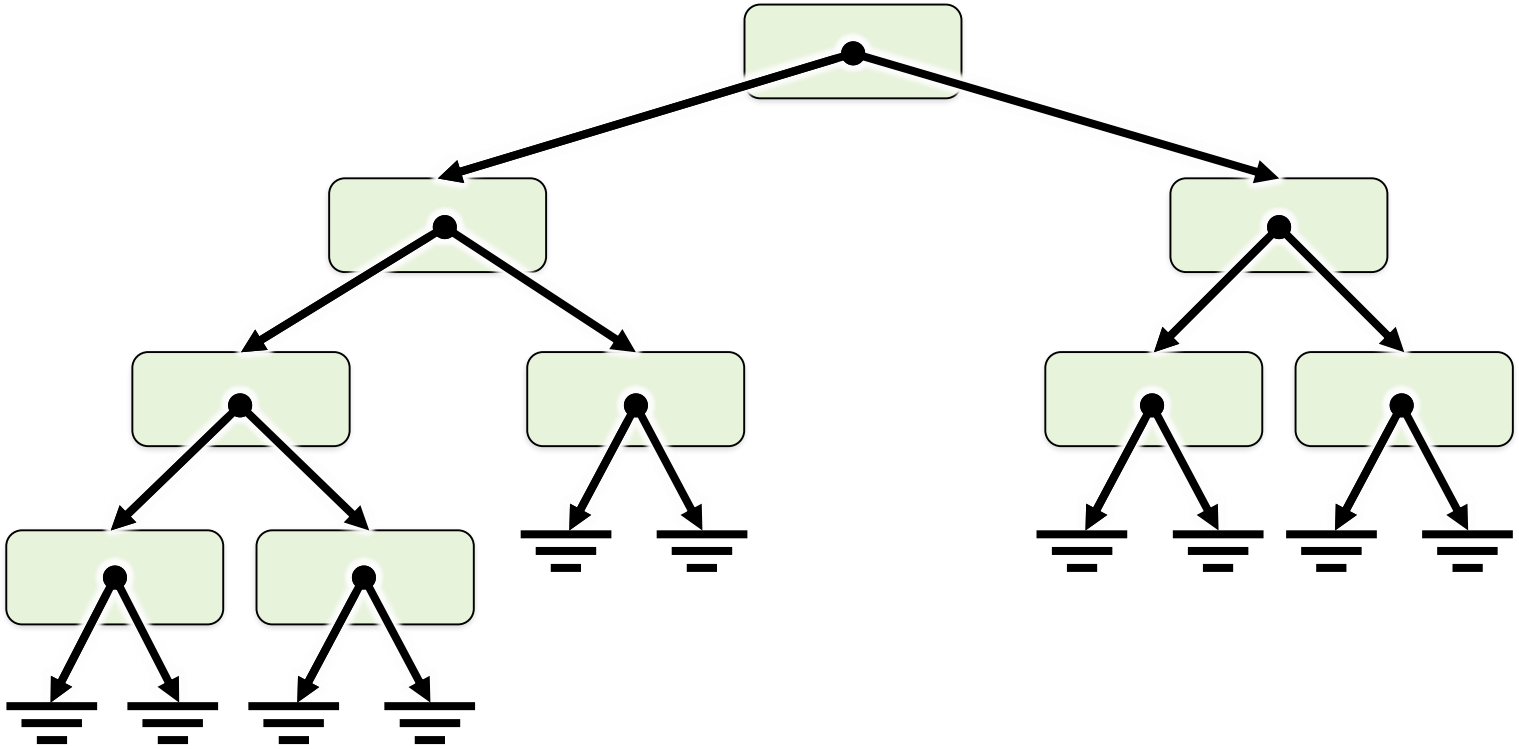
Samlung von Widgets

HList: horizontale Anordnung

z.B. eine Toolbar

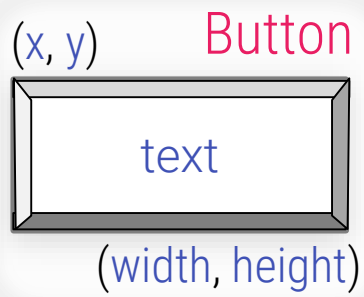
```
@dataclass
class HList(Widget):
    # public: zusätzliche öffentliche Eigenschaften
    child_widgets: List[Widget] = field(default_factory=list) # leere Liste

    # protected: implementation der Ereignisse
    def _draw(self, p: Painter) -> None:
        … # call draw on all contained widgets

    def _mouse_down(self, x: int, y: int, buttonNo: int) -> None:
        … # check which widget was hit, then call its “mouseDown” (update x,y coordinates as needed)
    def _mouse_move(self, x: int, y: int) -> None:
        … # check which widget was hit, then call its “mouseMove” (here: update coordinates as well)
    def _mouse_up(self) -> None:
        … # check which widget was hit, then call its “mouseUp”

    def _key_pressed(self, key: int) -> None:
        … # check which widget has keyboard focus, then call its “keyPressed”
```

Widget-Hierarchie: Objektbaum



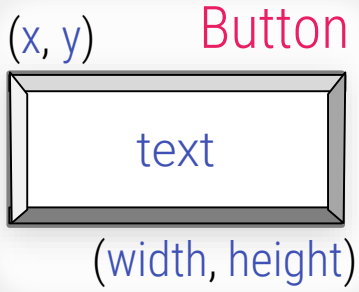
Gui-Frameworks (schematisch)

```
class Button(Label):   #  Button ist ein Label? Kein gutes Design, nur ein Beispiel…

    #  protected:

    #  Neues Ereignis: In Unterklasse überschreiben um Verhalten zu definieren
    @abstractmethod
    def _button_pressed(self) -> None: pass

    #  ----- Geerbtes ----------------------------------------------------------------------------
    #  Button um Text herum zeichnen
    def _draw(self, p: Painter) -> None:

        #  inherited – draw text (Button extends Label)
        super().draw(p);

        #  draw some 3D-shaded button (fictious graphics library)
        GfxUtils.draw_button(p, self.x, self.y, self.width, self.height);


    def _mouse_down(self, x: int, y: int, buttonNo: int) -> None:

      if buttonNo == 0: #  left mouse button
         self._buttonPressed()
```



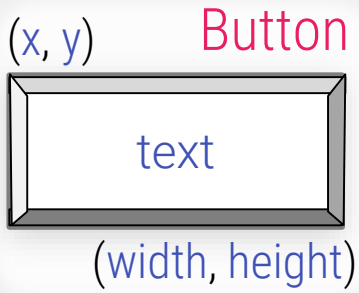
Nutzung der „Button“ Klasse

```
# Definiert vom Benutzer der Bibliothek
class MyButton(Button):

    # protected
    # In Unterklasse überschreiben…
    def _buttonPressed(self) {

        print("Button was pressed!")
```



Sinnvolle Nutzung

```
#  Definiert vom Benutzer der Bibliothek
@dataclass
class MyButton(Button):

    app_object: ApplicationObject

    #  Überschreiben in Unterklasse
    def _button_pressed(self) ->None:

        #  forward the event to my own application...
        self.app_object.perform_some_action()

#  später, irgendwo in der Anwendung
b: Button = MyButton()
b.x = ...
b.y = ...
b.width = ...
b.height = ...
b.text = "Start Server“

b.app_object = my_application.some_part.server_startup

toolbar: HList = ...; #  Eine Toolbar raussuchen (wurde schon erstellt)
toolbar.child_widgets.append(b); #  Button dazu
```



Eventbehandlung

Reale Implementierung?

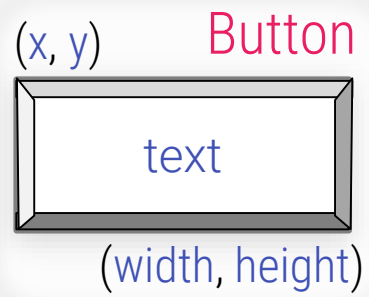
- JAVA AWT (abstract window toolkit)
  - Design sehr ähnlich zu unserem Beispielcode
  - Zu einfach für komplexe System
  - Immer noch verfügbar (Teil von JAVA SWING)

Die Sache ist etwas umständlich!

- Eigene Unterklasse für jedes eigene Objekt
- Bessere Event-behandlung nötig

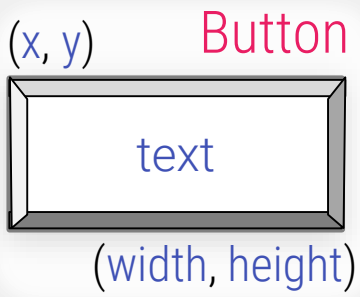
Bessere Button-Ereignisse

```
class Button(Label):   # Button ist ein Label? Kein gutes Design, nur ein Beispiel…

    # protected:

    # Neues Ereignis: In Unterklasse überschreiben um Verhalten zu definieren
    _button_pressed: Callable

    # Geerbtes
    # Button um Text herum zeichnen
    def _draw(self, p: Painter) -> None:

        # ...


    def _mouse_down(self, x: int, y: int, buttonNo: int) -> None:

      if buttonNo == 0: # left mouse button
         self._buttonPressed() # Aufruf des Methodenzeigers
```



Benutzung der neuen Klasse

```
#  Für Anwendung keine neue Button Klasse nötig

#  Beispiel Applikationsklasse
class MyApplication:
    …
    #  Methode (objektbezogen) von MyApplication
    def server_startup(self):
        …

#  Benutzung
b: Button = Button()
b.x = ...
b.y = ...
b.width = ...
b.height = ...
b.text = "Start Server"

my_app: MyApplication = MyApplication()
# ...

b._button_pressed = my_app.server_startup #  Referenz auf my_app wird implizit auch gespeichert!
#  Python Spezialität: Callables enthalten Kontext (“Closure”) – hier die Referenz auf “self” zusätzlich zur Methode
```



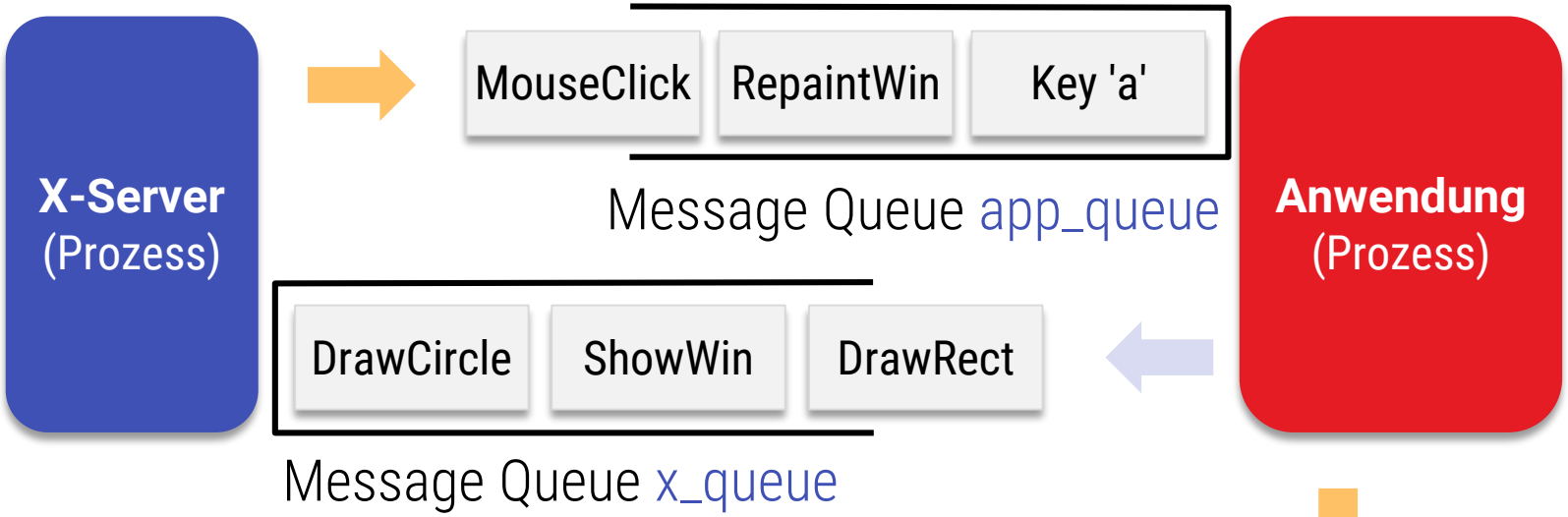
Event-Queues/Loops

(Ereignisorientierte Architektur)

Ereignisorientierte Programmierung

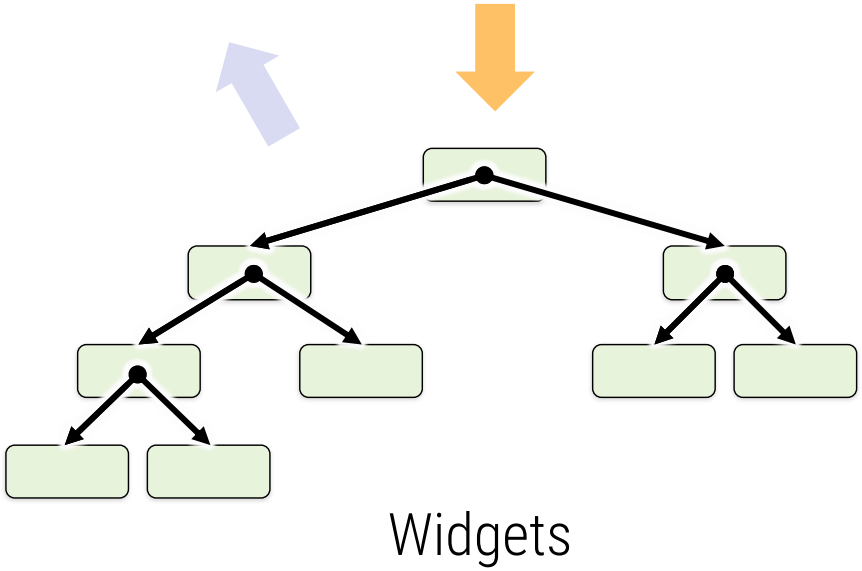
- Methodenaufruf, wenn Ereignis eingetreten ist
- Hollywood-Prinzip
  - „Don‘t call us – we call you“

Ereignisorientierte Programmierung

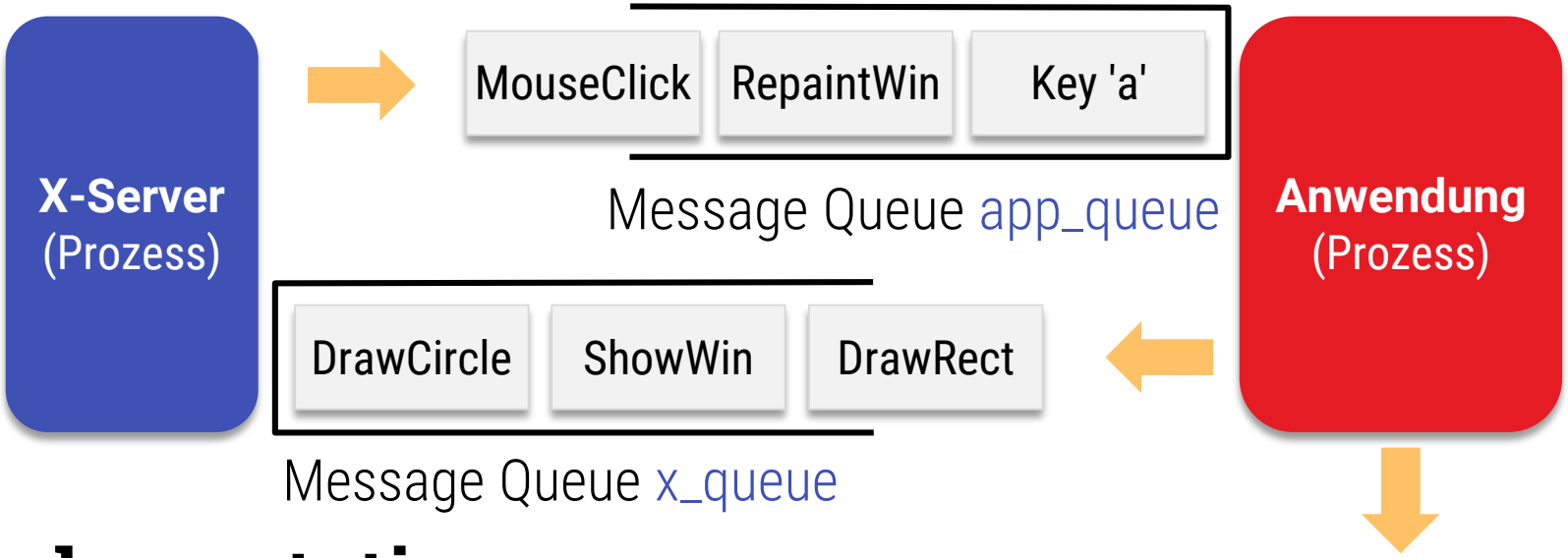


Implementation

- Messages & Callbacks
- Fenstermanager als nebenläufiger Prozess



Ereignisorientierte Programmierung



Implementation

- Messages & Callbacks
- Fenstermanager als nebenläufiger Prozess
- „Eventloop“ erzeugt Ereignisse

```
def event_loop(app_queue, top_window):
  while True:
    msg = wait_message(app_queue)
    if (msg.type = MOUSE_DOWN):
      top_window.mouse_down(msg.x, msg.y)
    elif (msg.type == MOUSE_UP):
      top_window.mouse_up(msg.x, msg.y)
    elif:
      …
```

Pseudo-Code!

Design Patterns

Observer Pattern

- Widgets als Observer
- Passives Verhalten
- „Reactive“

Vorteil

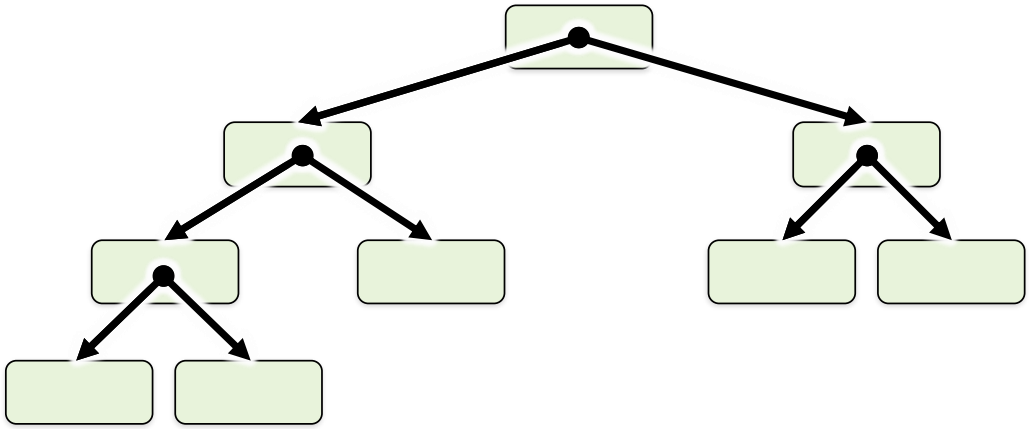
- Erweiterbarkeit, viele Widgets operieren parallel
- Widgets hinzufügen ohne Ereignisvert. zu ändern

Nachteil

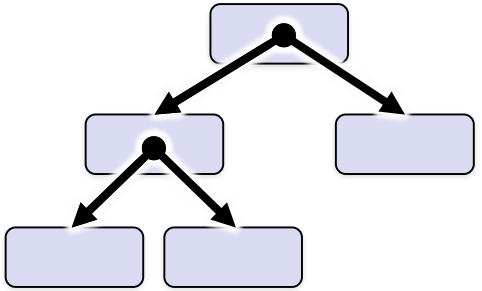
- Unintuitiv, Abläufe werden aufgebrochen

Ereignisse im Widget-Baum

Event-Behandlung

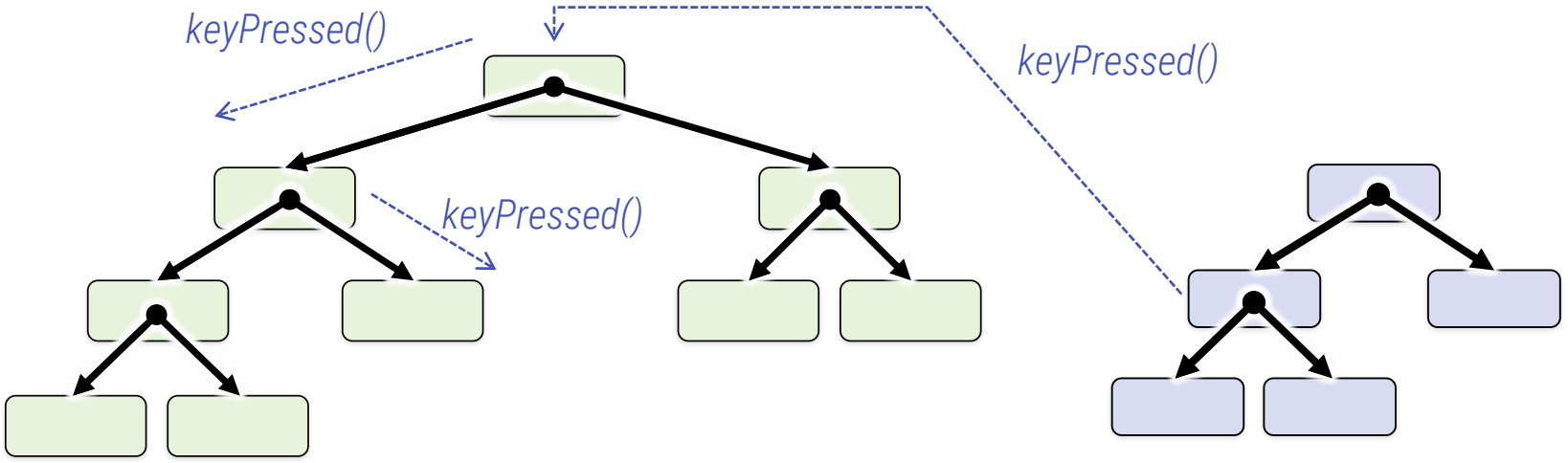


Widget Hierarchie



Anwendungs-
objekte

Event-Behandlung



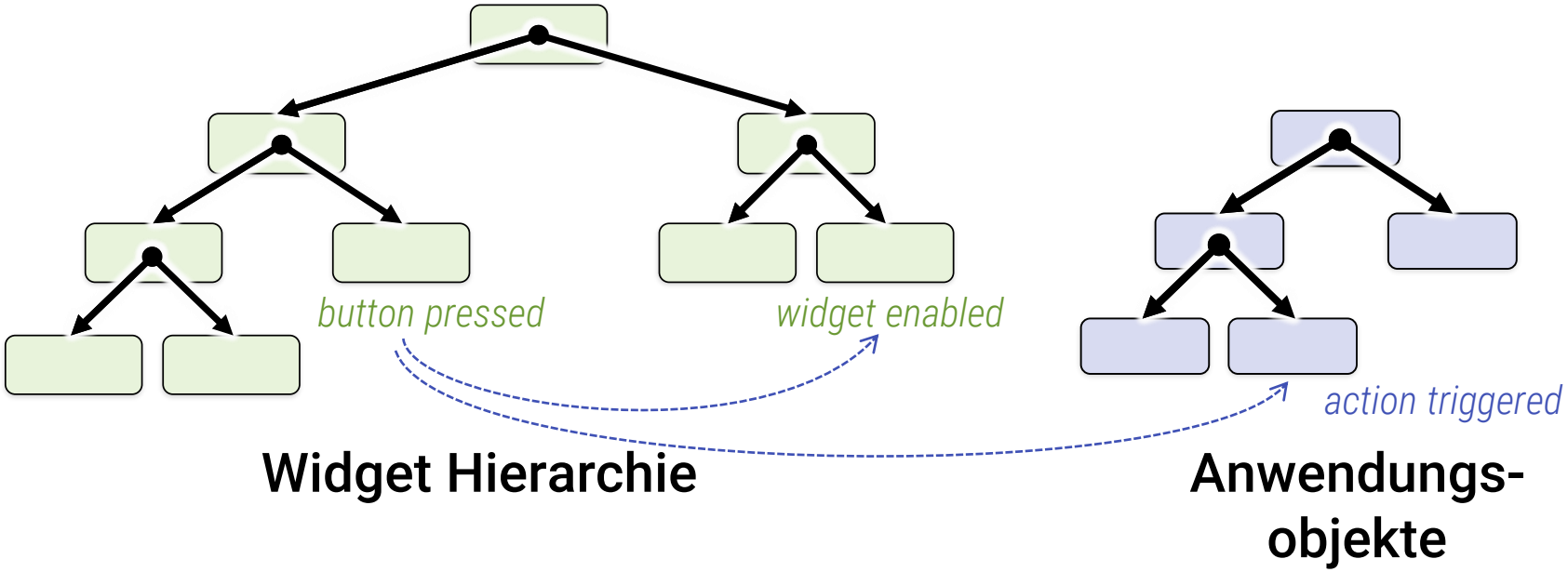
Widget Hierarchie

Anwendungs-objekte

Hierarchische Event-Verteilung

- Über Membermethoden gut möglich
- Anwendung oder System schickt Event an Hauptfenster, hiearchische Verteilung

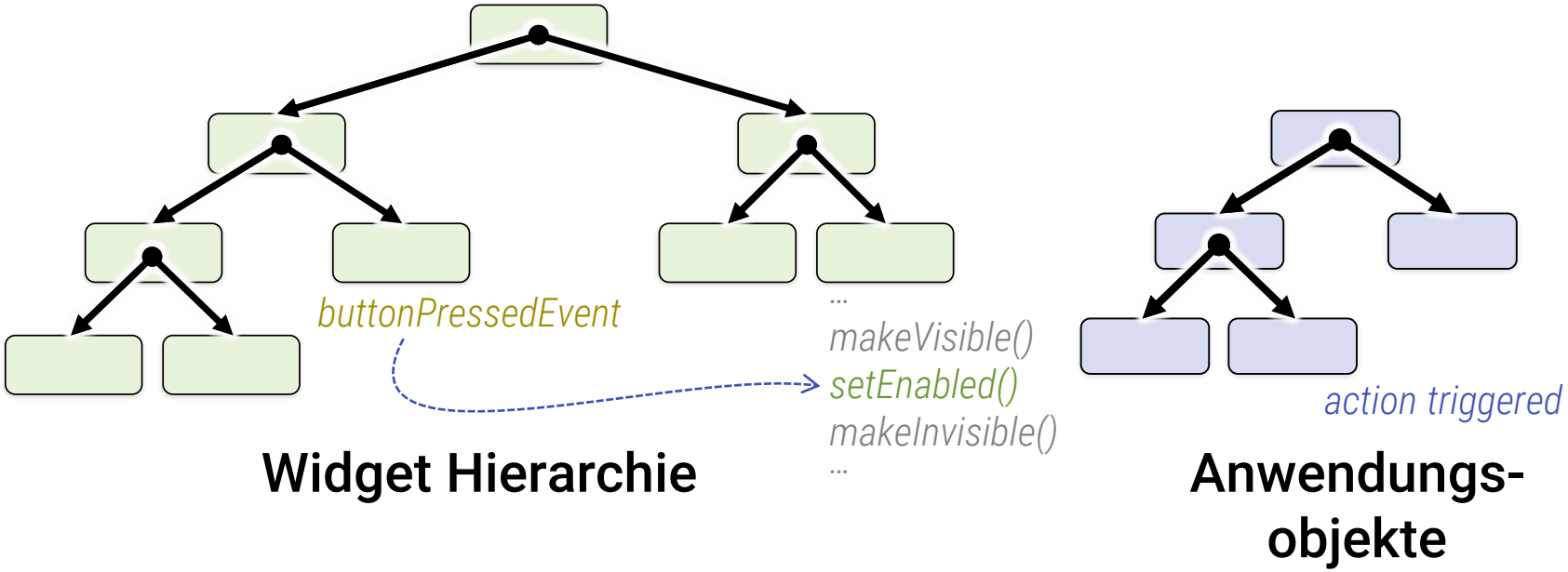
Event-Behandlung



Querverweise

- Neuer Mechanismus nötig
- Nachrichten quer zur Hierarchie versenden

Event-Behandlung



Querverweise

- Event-Variablen an Widgets
  - In Qt: Listen mit mehreren Empfängern
- Empfänger zugeordnet: Empfänger = (Objekt, Methode)

Lösungen

Mögliche Lösungen

- Abstrakte Methoden für Subklassen (z.B. Java-AWT)
  - Typischerweise: Nachfahre enthält Zeiger auf Empfänger
- Delegation: Zeiger auf Methoden von Objekten (z.B. Delphi)
  - Zeiger auf ein Objekt und eine seiner Methoden
- „Delegates“ oder Signales+Slots (z.B. Qt, C#)
  - Mehrere Empfängern möglich
    - C# Delegates: List von Zeigern (Objekt, Methode)
    - Qt Signals+Slots: Sicher (keine „dangeling references“ in C++)
- Moderne Sicht: Funktionen + Closure (Funktional)
  - z.B. Callable Objects in Python (auch Qt + PySide)

Beispiel: QT

Qt für Python: Connect

- Verbinden von ein oder mehreren Slots mit Signal
  - QPushButton hat ein Signal „`clicked`“
  - „Connect“ direkt zu Objekt mit Methode (obj.method)

```
class TestClass:
    def func():
        print("click click click!")

def func2() -> None:
    print("clicked the button")

test_obj: TestClass = TestClass()
button: QPushButton = QPushButton("Clickme")
button.clicked.connect(test_obj.func)
button.clicked.connect(func2) # beide werden bei Click aufgerufen
```

Qt für Python: Connect

- Verbinden von ein oder mehreren Slots mit Signal
  - QPushButton hat ein Signal „clicked“
  - „Connect“ direkt zu Objekt mit Methode (obj.method)

```
class TestClass:
    def func():
        print("click click click!")

def func2() -> None:
    print("clicked the button")

test_obj: TestClass = TestClass()
button: QPushButton = QPushButton("Clickme")
button.clicked.connect(test_obj.func)
button.clicked.connect(func2)
```

„stand-alone“ Funktion callable enthält Referenz auf func2()

Qt mit Python

Qt für Python: Connect

- Verbinden von ein oder mehreren Slots mit Signal
  - QPushButton hat ein Signal „`clicked`“
  - „Connect“ direkt zu Objekt mit Methode (obj.method)

```
class TestClass:
    def func():
        print("click click click!")

def func2() -> None:
    print("clicked the button")

test_obj: TestClass = TestClass()
button: QPushButton = QPushButton("Clickme")
button.clicked.connect(test_obj.func)
button.clicked.connect(func2)
```

„member“ Funktion
Callable enthält Referenz
auf func() sowie „test_obj“

„stand-alone“ Funktion
Callable enthält Referenz
auf func2()

Qt mit Python

Qt für Python: emit

- Auslösen / senden eines Signals
  - Wieder Beispiel QPushButton mit Signal „`clicked`“

```
button: QPushButton = QPushButton("Clickme")
button.clicked.connect(func)

button.clicked.emit() # emit löst Signal aus – alle verbundenen
                      #  Funktionen warden aufgerufen
```

  - Signale+Slots können auch Parameter haben
- Eigenen Signalen und Slots
  - Signale sind Instanzen von Klasse `QtCore.Signal`
  - Slots als Funktionen/Methoden mit „decorator“ `@slot`
  - Siehe Doku für Details → z.B. https://wiki.qt.io/Qt_for_Python_Signals_and_Slots

Die Eventhölle

Probleme mit Events

„Widgets“ modellieren zwei Ideen

- Kapselung: Komponenten als „Black Box“
- Reaktion auf Ereignisse

Probleme „Event-Spaghetti-Code“

- Code oft schwer zu verstehen / warten
- Mögliche Probleme
  - Verteilter Code + Zustand
    - Schwer wartbar und durschaubar
  - Zyklische Eventschleifen
    - Beobachtet z.B. bei Properties mit gettern/settern, die selbst Events auslösen

Was tun?

Wir brauchen mehr Struktur!

- MVC – Strukturelle Ordnung
  - Alte Idee, aber bewährt
- FRP – Ordnung des Kontrollflusses
  - Relativ junge Idee (2010er Jahre)
- Wiederholbarkeit, Scripting und Undo
  - Command Object Architectures

GUI-Architektur

Es gibt noch mehr Probleme (und Lösungsideen)

- Historie / Undo+Redo
  - Command-Objekt Architekturen
- Scriptfähigkeit von GUIs?
  - Abstraktion über Command-Objekts

Alles Beispielhaft

- Wenn Zeit, werden wir uns einige Beispiele anschauen

Entwurf von Benutzerschnittstellen

Gute (G)UIs

Bisher

- Wie programmiert man GUIs

Jetzt (kurz angerissen)

- Wie entwirft man GUIs?
- Schlechte UIs sind nicht selten…

Regel Nr. 1

Unersetzlich

- Gesunder Menschenverstand und…
- …Empathie: Denke an die Menschen
  - Nutzer/innen
    - Das wichtigste
    - Verschiedene Nutzergruppen (von Anfänger bis Guru)
    - Verschiedene Neigungen
  - Auch an Entwickler/innen
    - Wenn die Softwarearchitektur gute UIs behindert, wird das Ergebnis oft schlecht
- Erfahrung
  - Nutze gelungene Systeme als Beispiele/Vorbilder

Nach dem offensichtlichen…

Tipps zur Gestaltung

- Arten / Ansätze für UIs
- Heuristiken / Leitlinien
- Testen / Benutzerstudien

Ansätze

GUIs

- „Direktmanipulative Schnittstellen“
  - Visualisierung eines Datenmodell als 2D/3D Bilder
  - Direkte Interaktion (i.d.R. Maus/Touch/Tastatur)
    - Bildsprache entwerfen (Icons, Handles, „Gizmos“ etc.)
    - Direkte Interkation:
      - nah an physikalischen Vorgängen,
      - aber semantisch durchaus abstrakt
      - (z.B. „Datei in Mülleimer ziehen“)
  - Wichtig: Konsistent und konzeptionell einfach genug
    - Wenige Konzepte, die vieles ausdrücken können
- Andere: Dialogboxen, Command-Lines, etc.

Leitlinien (Beispiel)

„Important Design Considerations“ *)

- Be consistent – sei konsistent
  - Gleiche Farbe = ähnliche Bedeutung
  - Statusmeldungen immer am gleichen (logischen) Ort
  - Icons/Menüs immer an der gleichen Stelle
    - „muscle memory“
  - Konzeptionell gleiche Bedeutung für alles, z.B.
    - Globale Befehle (z.B. Anwendung beenden)
    - Lokale Befehle (z.B. Copy+Paste)
    - Short-Cuts (z.B. Ctrl+C, Ctrl+V oder „halte <Strg>“ für Kopieren)

*) Loosely based on: Foley, van Dam, Feiner, Hughes:
Computer Graphics, Principles & Practice, 2nd Edition (Addison-Wesley 1990)

Leitlinien (Beispiel)

„Important Design Considerations“ *)

- Provide Feedback
  - User sollte verstehen, was gerade vor sich geht
  - Reaktionen des Systems nachvollziehbar
  - Länger laufende Prozesse erkennbar (z.B. Progress-Bar)
    - Alles was länger als ein paar Sekunden dauert „nervt“
- Minimize Error Possibilities
  - Was man nicht tun soll, wird nicht angeboten
    - z.B. Unmögliche Aktionen „ausgrauen“
  - Fehleranfällige Befehle vermeiden
    - Negativ-Beispiel: `rm -r .*` unter Unix (nicht machen!!!)

*) Loosely based on: Foley, van Dam, Feiner, Hughes:
Computer Graphics, Principles & Practice, 2nd Edition (Addison-Wesley 1990)

Leitlinien (Beispiel)

„Important Design Considerations“ *)

- Provide Error Recovery
  - Benutzer sollte keine Angst haben, etwas kaputt machen zu können
  - Beispiel: Undo/Redo, Command Histories aufzeichnen
  - Beispiel: Cancel-Button
- Accommodate Multiple Skill Levels
  - Anfänger/innen sollten „schnell reinkommen“
  - Fortgeschrittene Benutzer können effizient arbeiten
    - Möglichst „Learning on the Fly“
    - Beispiel: Tastaturshortcuts in GUI anzeigen
    - Befehle aufzeichnen (oder Macros) für „Customization“ / Erweiterbarkeit

*) Loosely based on: Foley, van Dam, Feiner, Hughes:
Computer Graphics, Principles & Practice, 2nd Edition (Addison-Wesley 1990)

Leitlinien (Beispiel)

„Important Design Considerations“ *)

- Minimize Memorization
  - Man sollte möglichst wenig auswendig lernen müssen
    - Negativbeispiel: `<ESC>:wq!<return>` um Editor `vi` zu verlassen
    - Explorability: Funktionen einfach selbst entdecken
- Modi und Syntax
  - Möglichst wenige (insbesondere verborgene Modi)
  - Klarer, einfacher Syntax (Abfolge von Befehlen)
    - Oft empfohlen: Select and Apply Command (aus Palette)
    - Andere Quellen: „Erzwungende Sequentialität“ reduzieren
- Visual Clarity: Hübsch und gut strukturiert

*) Loosely based on: Foley, van Dam, Feiner, Hughes:
Computer Graphics, Principles & Practice, 2nd Edition (Addison-Wesley 1990)

Andere berühmte Heuristiken

„Ben Shneiderman's Eight Golden Rules"

1. Consistency
2. Provide Shortcuts
3. Provide Informative Feedback
4. Dialogue *(Erklären, was gerade passiert)*
5. Error handling *(Fehler erklären und vermeiden)*
6. Permit reversal of actions
7. Support internal locus of control
   - *Benutzer Herr der Lage ("feels in control")*
8. Reduce short-term memory load

Was kann ich noch tun?

Prototypen

- „Storyboards“ – Skizzen von UIs auf Papier malen
  - Ggf. Diskussion / Anwendungsszenarien durchspielen

Testen

- UIs ausprobieren
  - Selbst kontinuierlich ausprobieren („Dogfooding“ oft hilfreich)
- Benutzerstudien
  - Benutzertests, Feedback sammeln
  - Bei komplexen Produkten:
    Mit Prototypen/Mock-Ups während Analysephase(n)
  - Szenario beschränkt: Gefahr von „Optimierung für Anfänger“
    - Gesunder Menschenverstand + Augenmaß nötig

Persönliche Erfahrungen

Eigene Erfahrungen

- Architektur der Anwendung reflektiert sich oft im UI
  - Ob man es will (kann Sinn machen!) oder nicht (Komplexität)
  - Gute, klare, einfache interne Struktur hilft UI-Design
- Erprobte Prinzipien helfen
  - UI-Konventionen, Style-Guides hilfreich (z.B. Ctrl+C, Ctrl+V)
  - Architekturmuster
    - z.B. OOP-Struktur mit objekt-orientiertem GUI mit Icons, Objektbezogenen Aktionen etc.
    - z.B. Funktionale Strukturen visualisert als Datenflußgraph
- Die hohe Kunst (u.U. schwer zu erreichen)
  - Wie „Lego“: Orthogonal, modular, erweiterbar, konfigurierbar

Meine Zusammenfassung

Gute UIs

- An (alle) Benutzer/innen denken
  - Mensch im Mittelpunkt
  - Bereitet ihnen Freude!
- Informieren: Ratschläge (kritisch) lesen
  - Nichts wildes
  - Größtenteils „gesunder Menschenverstand“, aber man muss es sich bewusst machen
- Aus Vorbildsystemen übernehmen, was funktioniert

Mehr dazu…

Human-Computer-Interaction…

- …ist ein eigenes Forschungsgebiet

Hier lediglich

- Anregungen / Denkanstöße zur Gestaltung