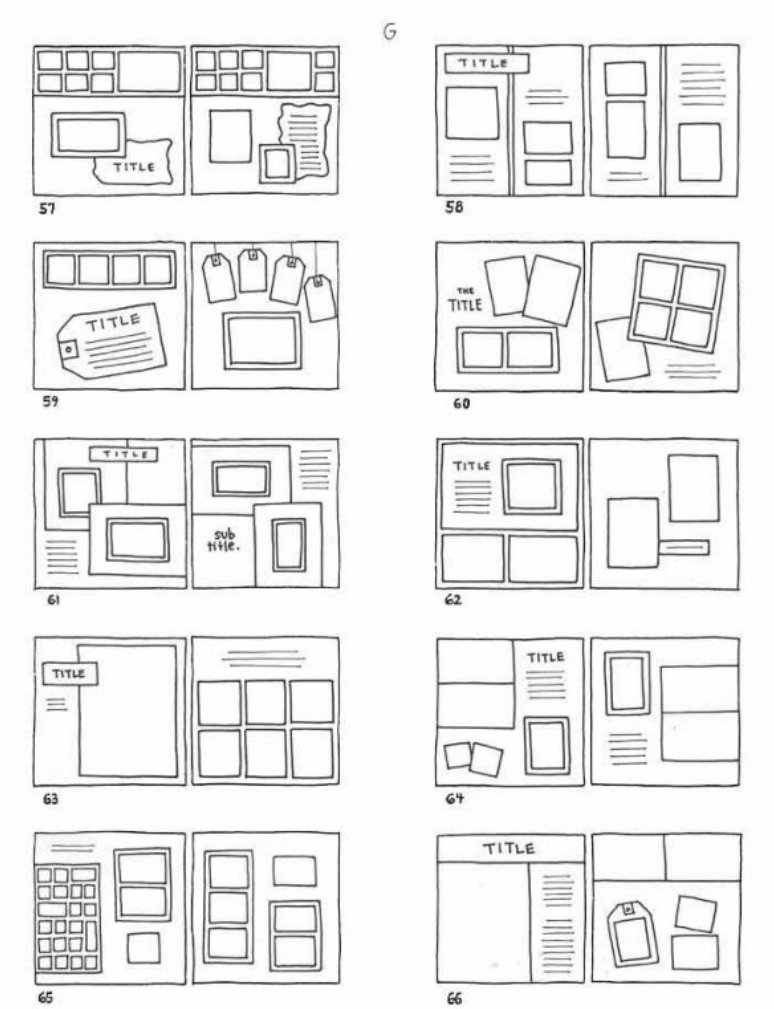
Block 5 – Layouts I



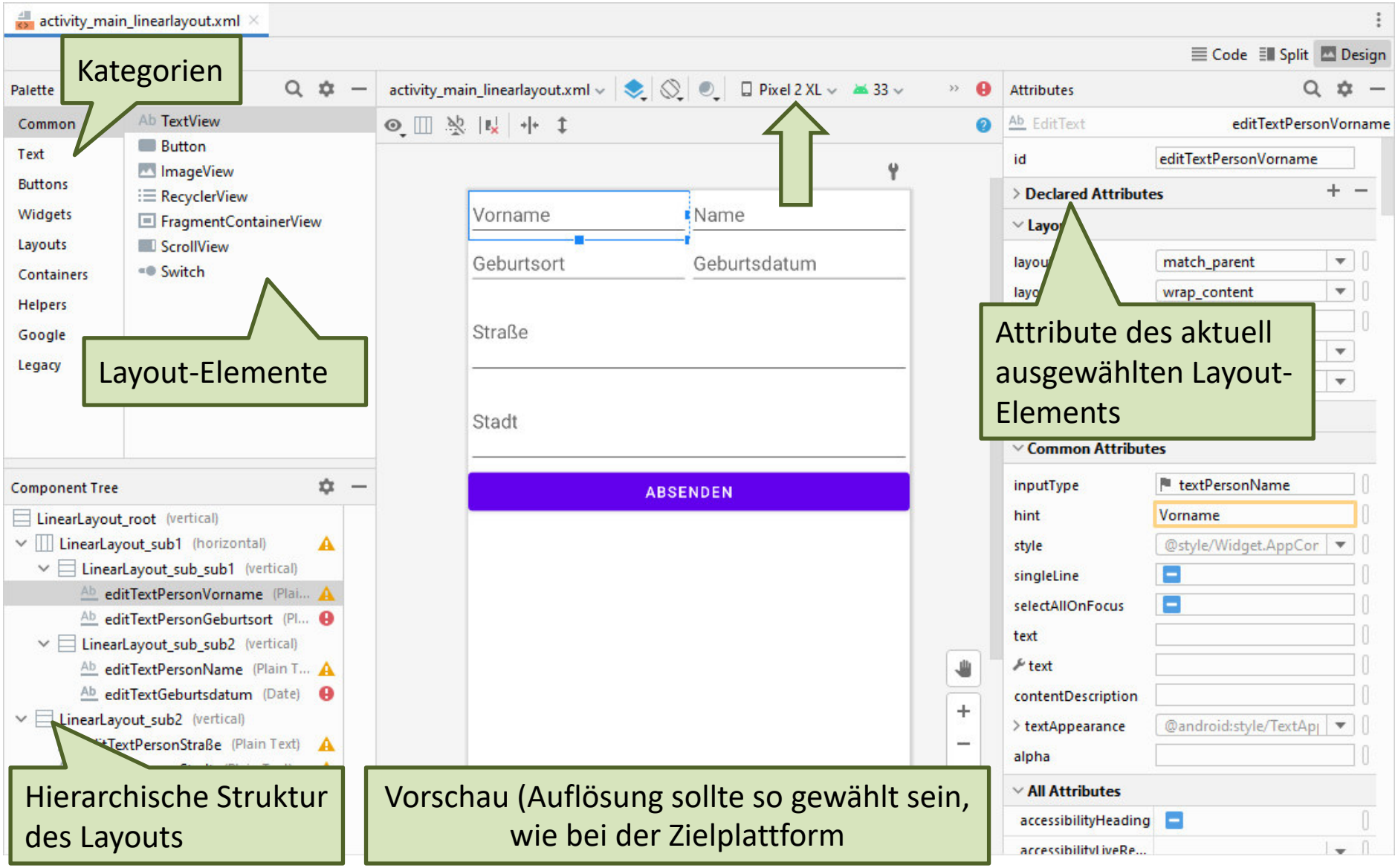
- Erstellung
- Hierarchie der GUI-Komponenten
- GUI-Komponenten
- Linear Layout
- Frame Layout
- Constraint Layout
- Kontextspezifische Layouts
- Ausblick

Block 5 – Lernziele

In diesem Block werden Sie lernen ...

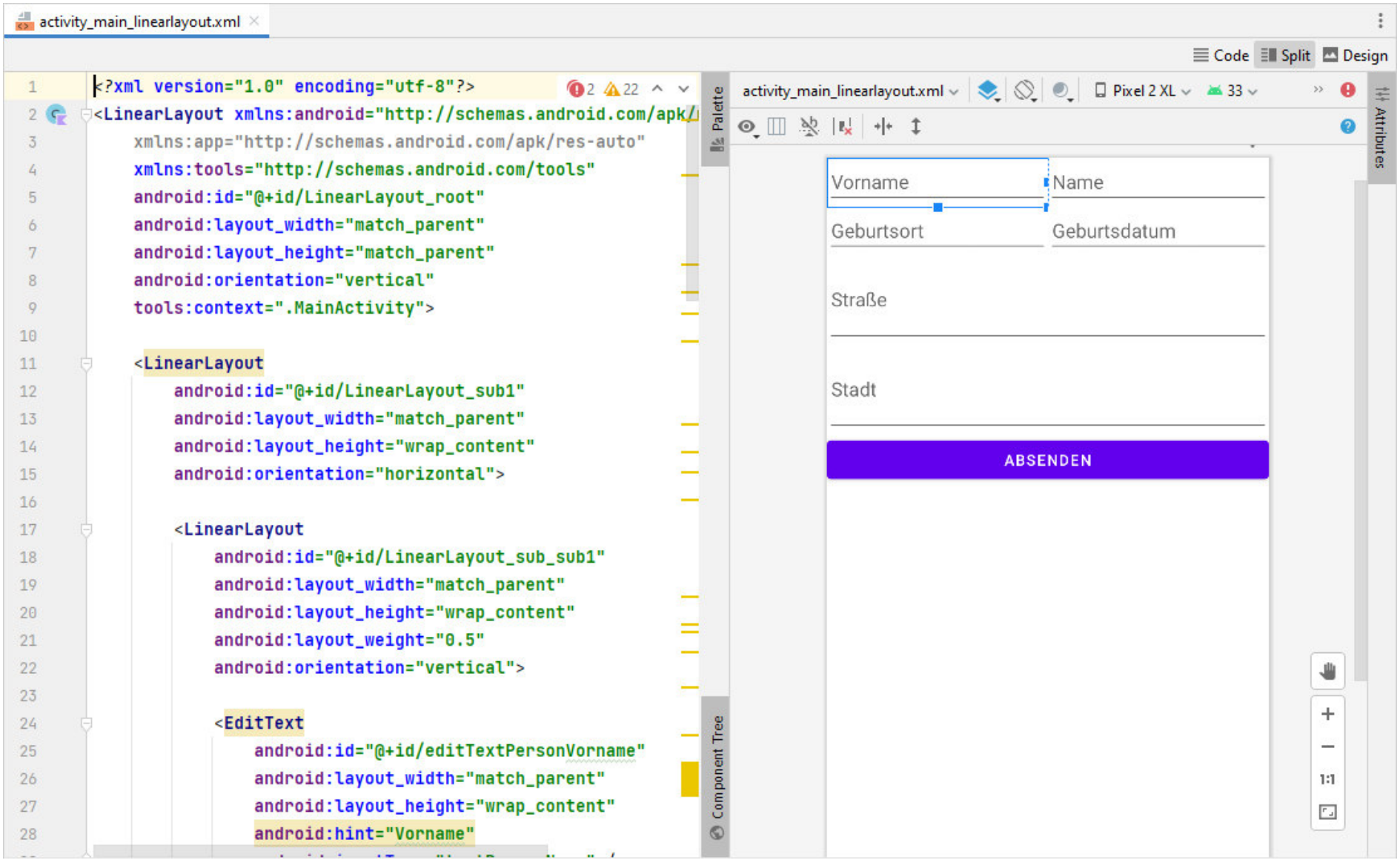
- eine Benutzeroberfläche auf verschiedene Arte zu definieren bzw. zu modifizieren (Layout-Editor, XML-Editor, Programmatisch, Programmatisch generisch).
- welche Hierarchie bei GUI-Komponenten der Android Plattform besteht.
- hierarchisch gegliederte grafische Benutzeroberflächen zu erstellen (z. B. Linear Layout).
- frei gegliederte Benutzeroberflächen zu erstellen (z. B. Frame Layout, Constraint Layout).
- welche Vor- und Nachteile die Layoutvarianten haben.
- wie die geräteabhängige Darstellung von GUI-Komponenten erfolgt.
- Layoutvarianten für bestimmte Kontexte zu nutzen.

5.1 Erstellung (Graphischer Layout-Editor)



Variante 1: Graphischer Layout-Editor

5.1 Erstellung (Textueller XML-Editor)



Variante 2: Textueller XML-Editor

5.1 Erstellung (Programmatisch)

Variante 3: Programmatisch

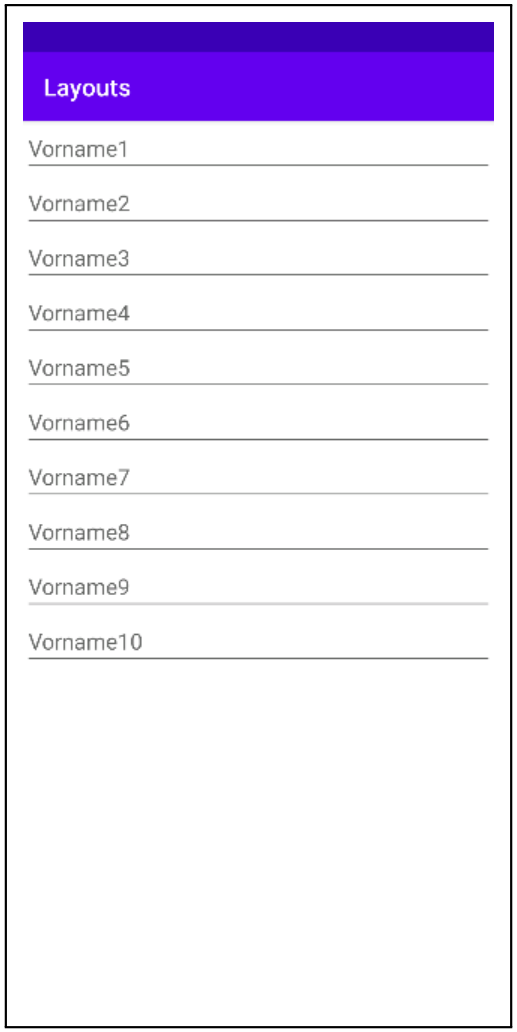
```
fun programmatic_layout(context: Context): View {

    val LinearLayout_root = LinearLayout(context)

    LinearLayout_root.orientation = LinearLayout.VERTICAL
    val layoutParams_match_match =
    ViewGroup.LayoutParams(ViewGroup.LayoutParams.MATCH_PARENT,
        ViewGroup.LayoutParams.MATCH_PARENT)
    LinearLayout_root.layoutParams = layoutParams_match_match
    ...
    val editTextPersonVorname = EditText(context)
    editTextPersonVorname.hint = "Vorname"
    editTextPersonVorname.layoutParams = layoutParams_match_wrap
    LinearLayout_sub_sub1.addView(editTextPersonVorname)
    ...
    return LinearLayout_root

}
```

5.1 Erstellung (Programmatisch - generisch)

Variante 3: Programmatisch (generisch)

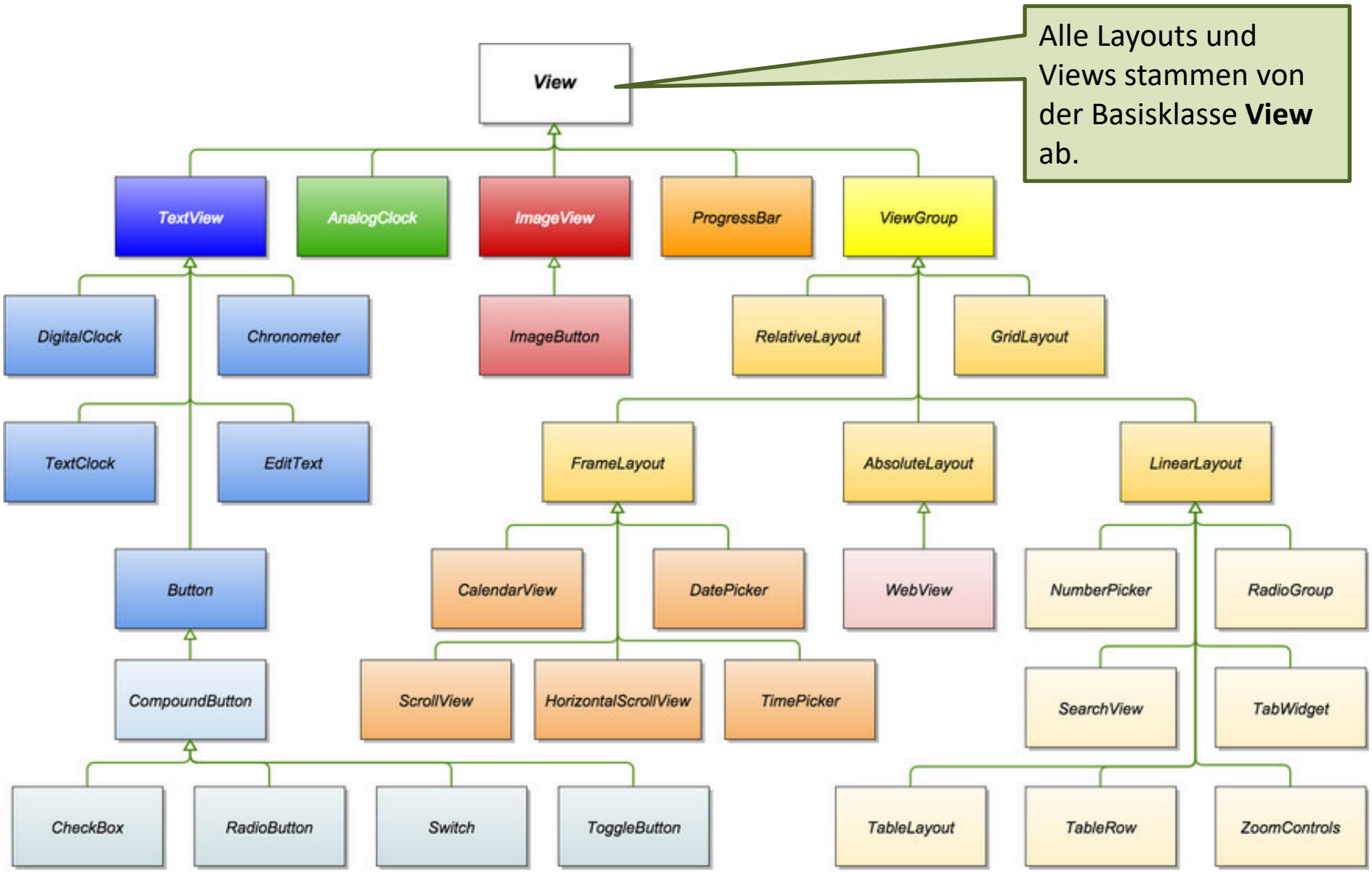
```
fun programmatic_layout_generic(context: Context): View {

    val LinearLayout_root = LinearLayout(context)
    LinearLayout_root.orientation = LinearLayout.VERTICAL

    for (i in 1..10) {
        val editTextPersonVorname = EditText(context)
        editTextPersonVorname.hint = "Vorname" + i
        val layoutParams_match_wrap = LinearLayout.LayoutParams(
            ViewGroup.LayoutParams.MATCH_PARENT,
            ViewGroup.LayoutParams.WRAP_CONTENT)
        editTextPersonVorname.layoutParams = layoutParams_match_wrap
        LinearLayout_root.addView(editTextPersonVorname)
    }

    return LinearLayout_root
}
```

5.1 Erstellung (Programmatisch - generisch) - Forts.



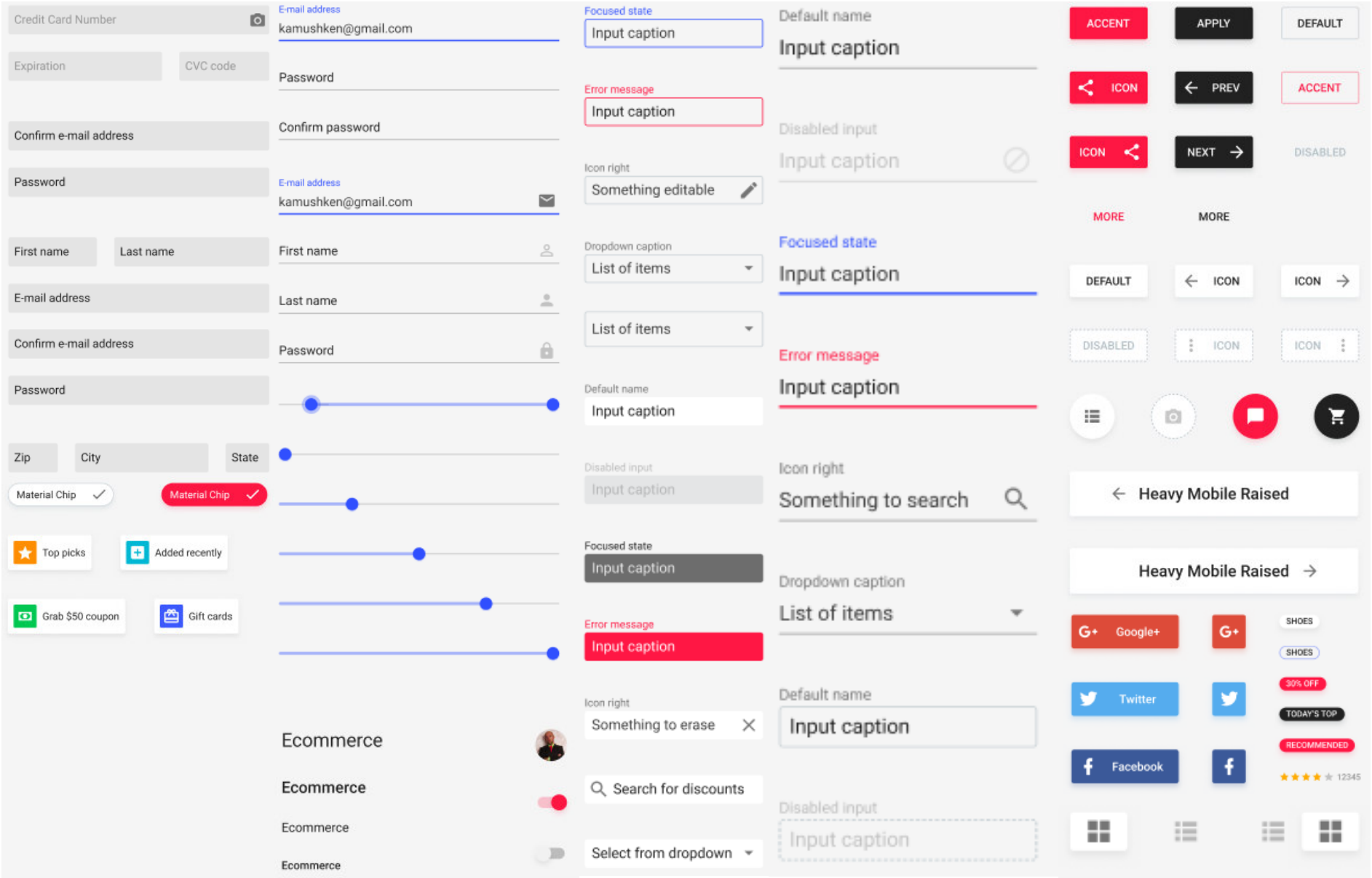
Programmatisch (generisch)

5.2 Hierarchie der GUI-Komponenten



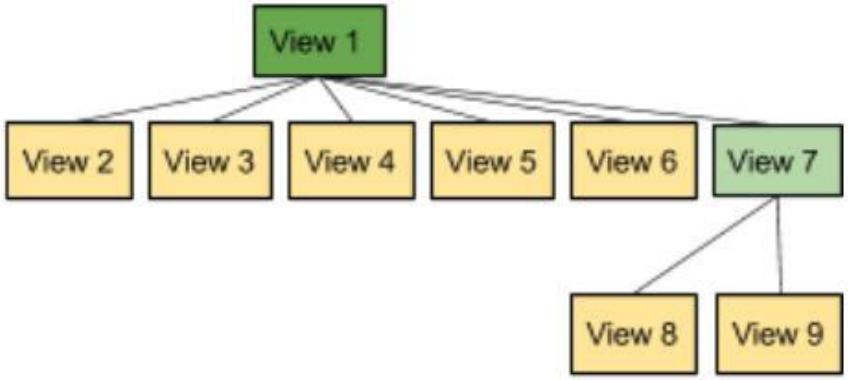
(https://o7planning.org/de/10423/anleitung-android-ui-layouts)

5.3 GUI-Komponenten

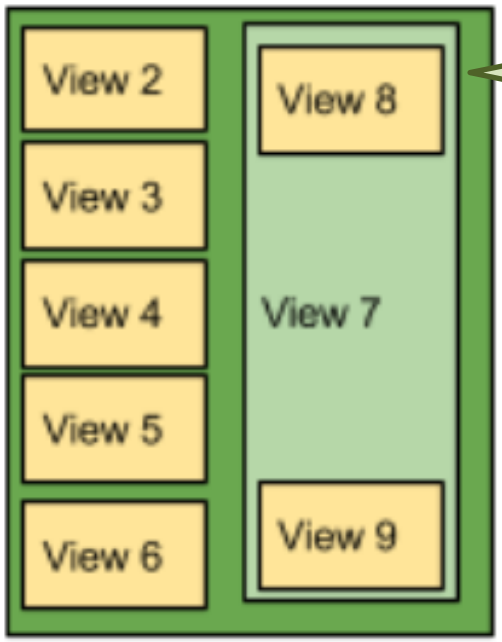


(https://medium.com/@kamushken)

5.4 Linear Layout (View-Hierarchie)

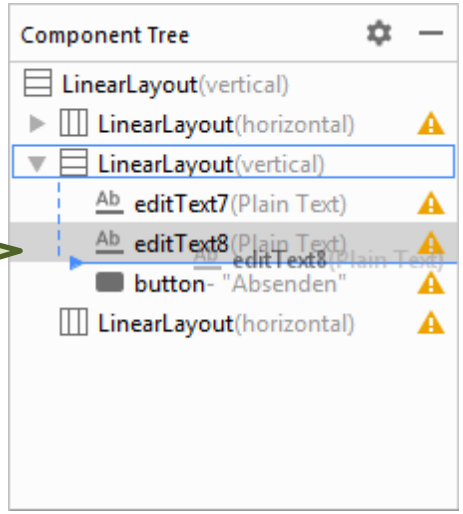


Views können beliebig geschachtelt werden, um ein bestimmtes Gesamtlayout zu erreichen.



Abhängig von den gewählten Layouts (und deren Attributen) ergibt sich dann eine Gesamtdarstellung.

Im ‚Component Tree' können die verschachtelten Views verschoben werden, um ein verändertes Layout zu erreichen.



(https://medium.com/androiddevelopers/simplify-complex-view-hierarchies-5d358618b06f)

5.4 Linear Layout

Das LinearLayout ordnet die eingebetteten Steuerelemente entweder horizontal (nebeneinander) oder vertikal (untereinander) an.

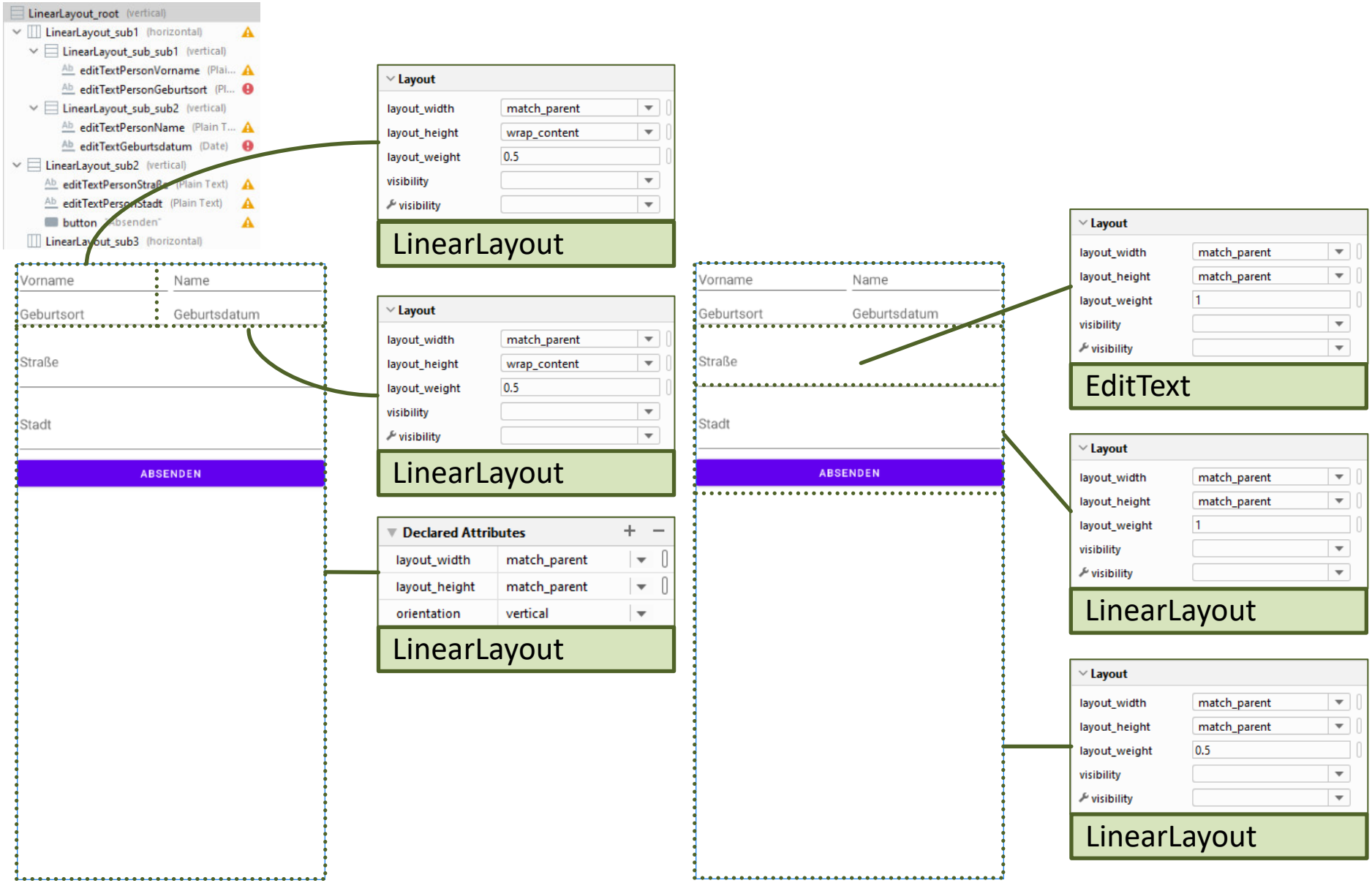
Die Ausrichtung wird über den XML Parameter `android:orientation` festgelegt. Mögliche Werte sind `horizontal` (Standard) und `vertical`.

Über die Parameter `android:layout_width` und `android:layout_height` kann bestimmt werden, ob sich das LinearLayout nur bis zur Größe der innenliegenden Elemente ausdehnt (`wrap_content`) oder den maximal verfügbaren Raum einnimmt (`match_parent`).

LinearLayouts lassen sich verschachteln.

Mit dem Attribut `layout_weight` können verschachtelte Layouts in eine Größenrelation gesetzt werden, indem eine Gewichtungsfaktor angegeben wird (z.B. 2:1).

5.4 Linear Layout (Beispiel)



5.5 Frame Layout

In einem FrameLayout werden die eingebetteten Layout-Elemente unabhängig voneinander und ggf. übereinander angezeigt.

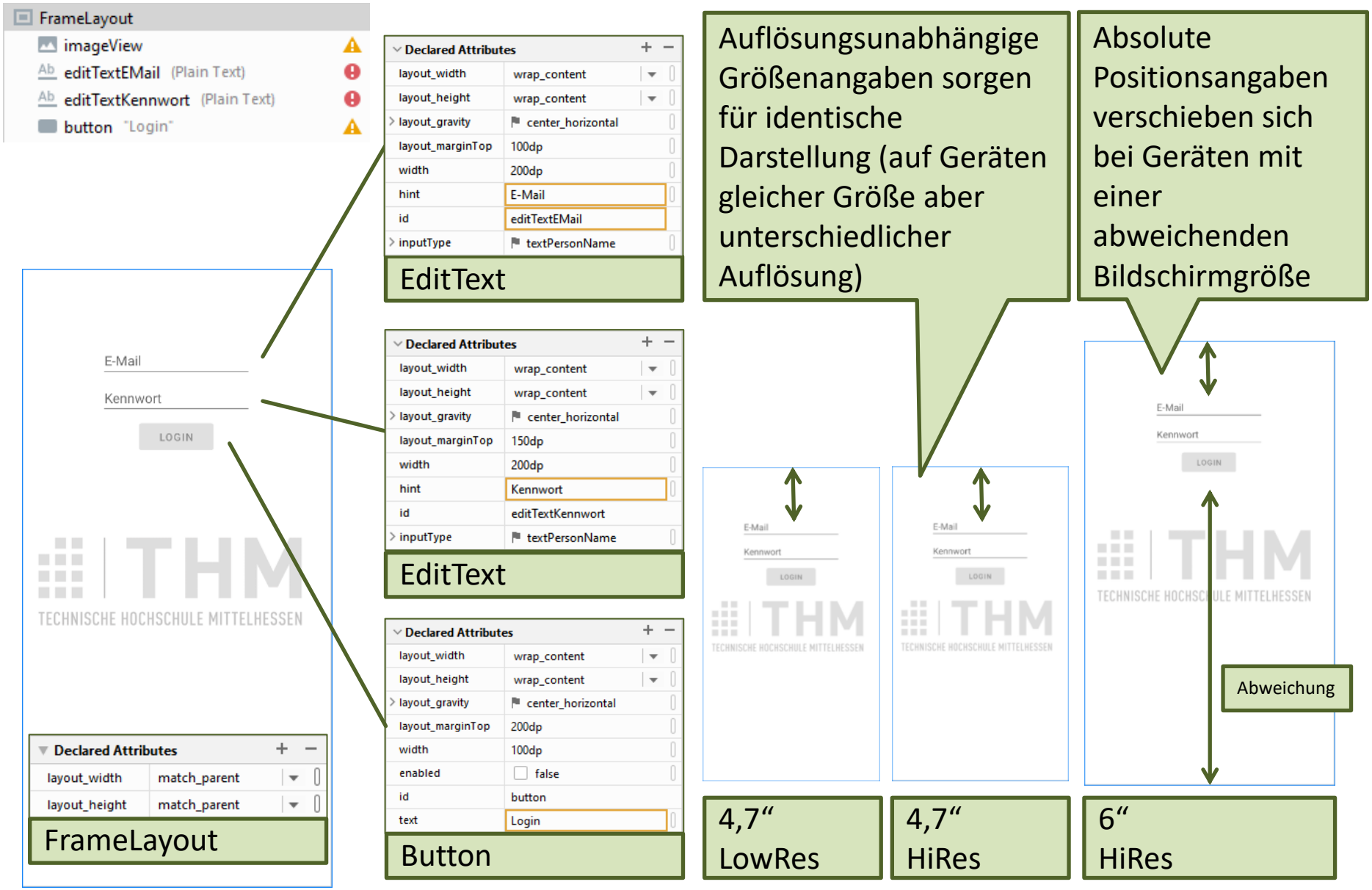
Die Überlagerung von Layout-Elementen ist in zwei Situationen sinnvoll: a) Layout-Elemente sollen auf bestimmten Hintergründen angezeigt werden, oder b) bestimmte Varianten von Layout-Elementen sollen erst zur Laufzeit aktiviert bzw. deaktiviert werden.

Die Größenangaben für die Layout-Elemente erfolgen analog zum LinearLayout (d.h. über die Attribute `android:layout_width` und `android:layout_height`)

Relative Positionierungen können über das Attribut `layout_gravity` erfolgen.

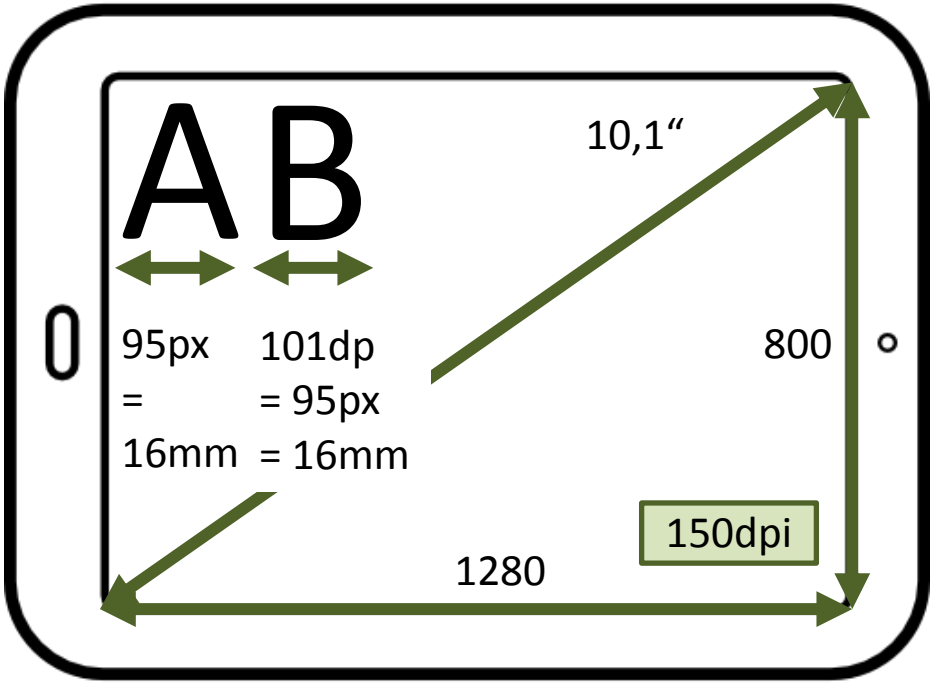
Absolute Positionierungen können über das Attribut `layout_Margin` erfolgen.

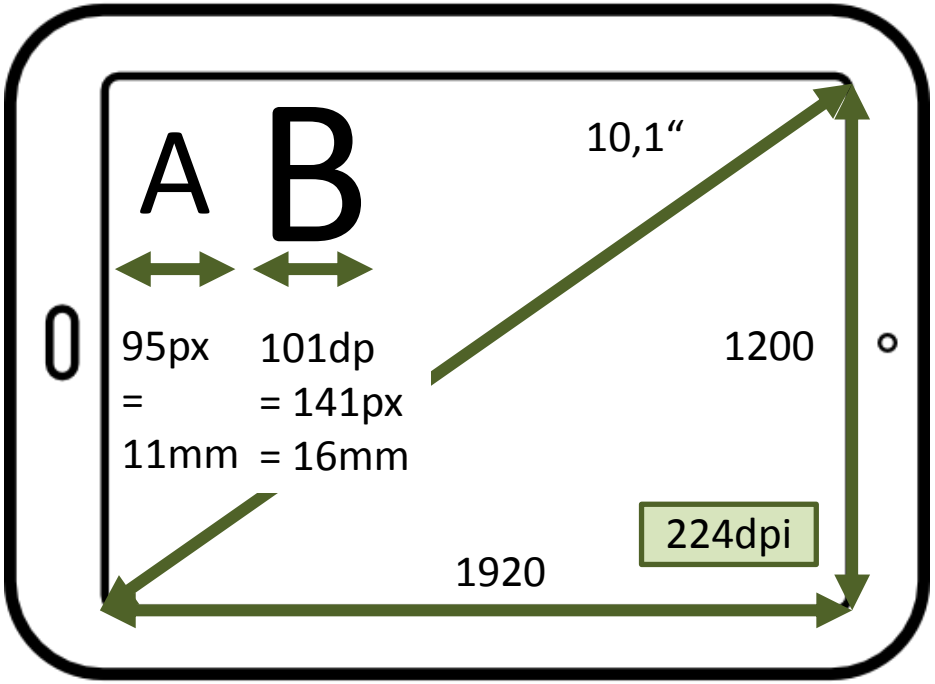
5.5 Frame Layout (Beispiel)



5.5 Frame Layout (Exkurs)

Ziel: Gerätunabhängige Darstellung eines Zeichens mit 16mm Breite.





16mm in px:

$800px^2 + 1280px^2 = \sqrt{2.278.400px} = 1509.43px$

$1509px/10{,}1" = 150.9\ dpi$

$2{,}54\ cm = 25{,}4\ mm = 1"$

$16\ mm = 16/254"$

$\underline{150.9\ dpi * 16/25{,}4 = 95\ \text{px}}$

$dp = px/(dpi/160)\ \ dp = 95/(150/160) = 101.3\ \text{dp}$

95px in mm:

$1200px^2 + 1920px^2 = \sqrt{5126400px} = 2264.15px$

$2264px/10{,}1" = 224.1\ dpi$

$1“/224.1\ dpi * 95px = 0.4239"$

$\underline{0.4239" * 25.4\ mm = 10{,}76\ mm}$

$px = dp * (\text{dpi}/160)\ px = 101 * (224/160) = 141.9px$

5.6 Constraint Layout

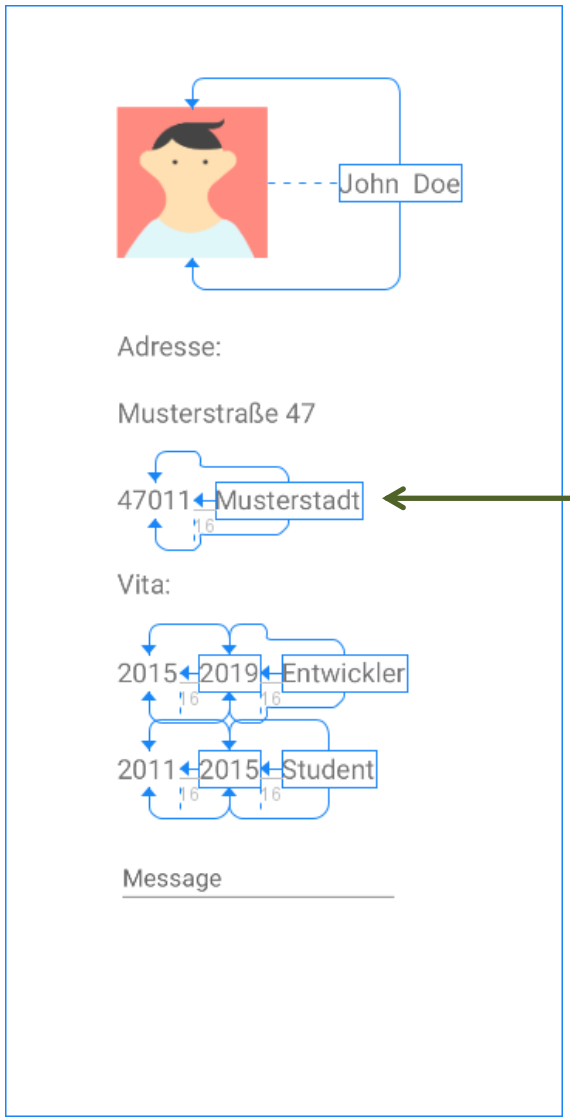
Problem: Die Verschachtelung verschiedener Layout-Typen wird als umständlich empfunden, da zum einen das Ziel-Layout nicht direkt erstellt werden kann (es werden Hilfskonstrukte benötigt), zum anderen gestalten sich Anpassungen umso schwieriger, je größer die Layouthierarchie ist.

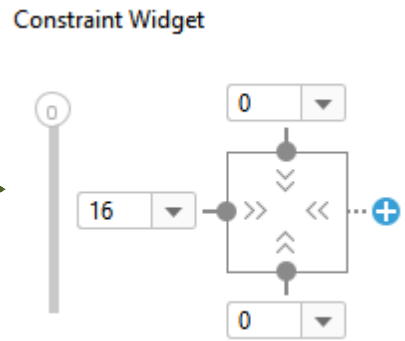
Lösung: Das Constraint Layout verzichtet auf eine Verschachtelung (obwohl möglich) und erlaubt eine flache Layouthierarchie. Dabei werden im Wesentlichen Bedingungen aufgestellt, welche die relative Anordnung der Layout-Elemente zueinander beschreiben.

Dieses Layout-Designparadigma ist ähnlich zu der vorhanden Vorgehensweise bei der Anwendungsentwicklung für iOS.

Das Constraint Layout ist Standard in Android Studio (Component Tree ▷ Convert view ...)

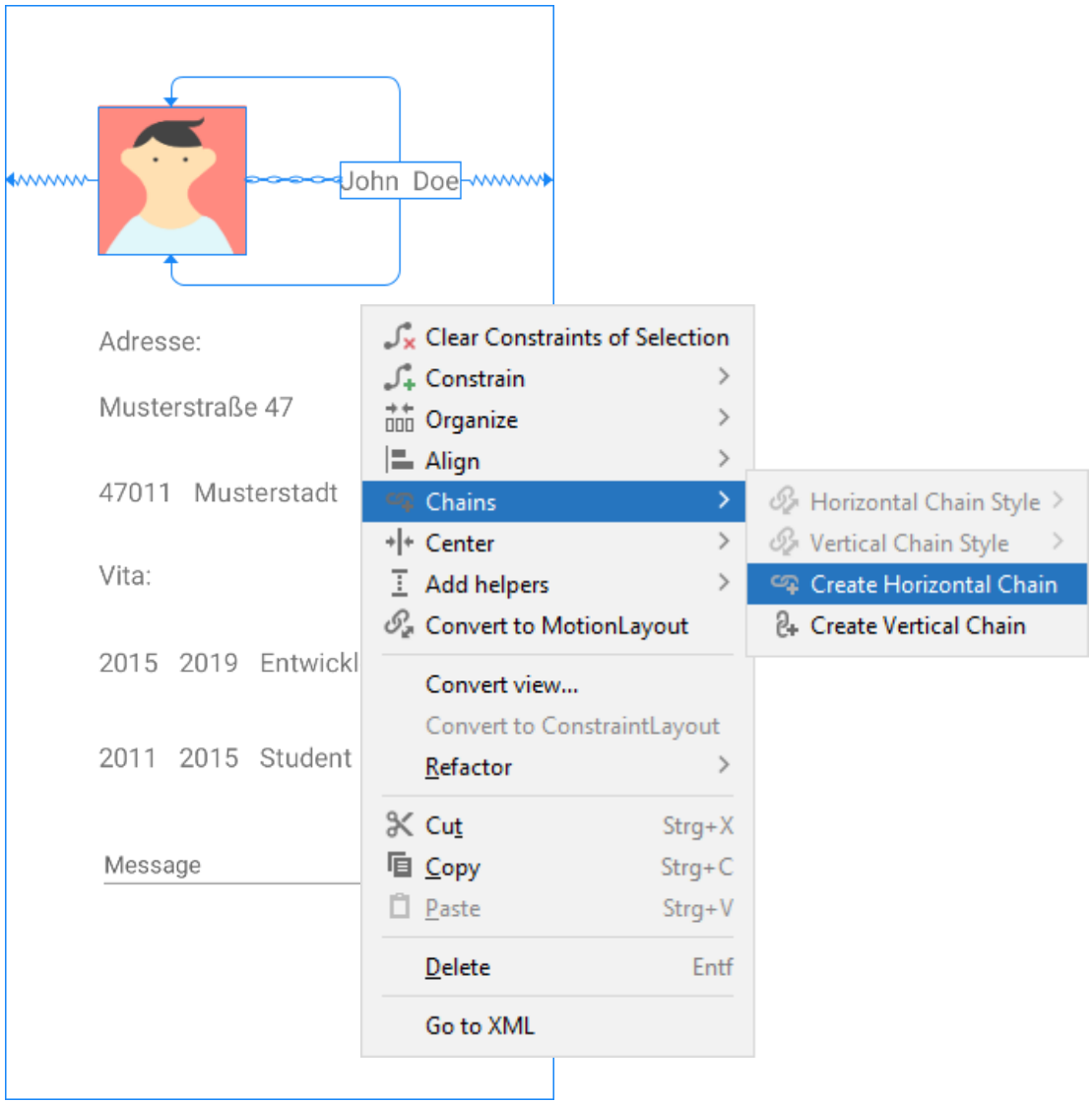
5.6 Constraint Layout (Beispiel)

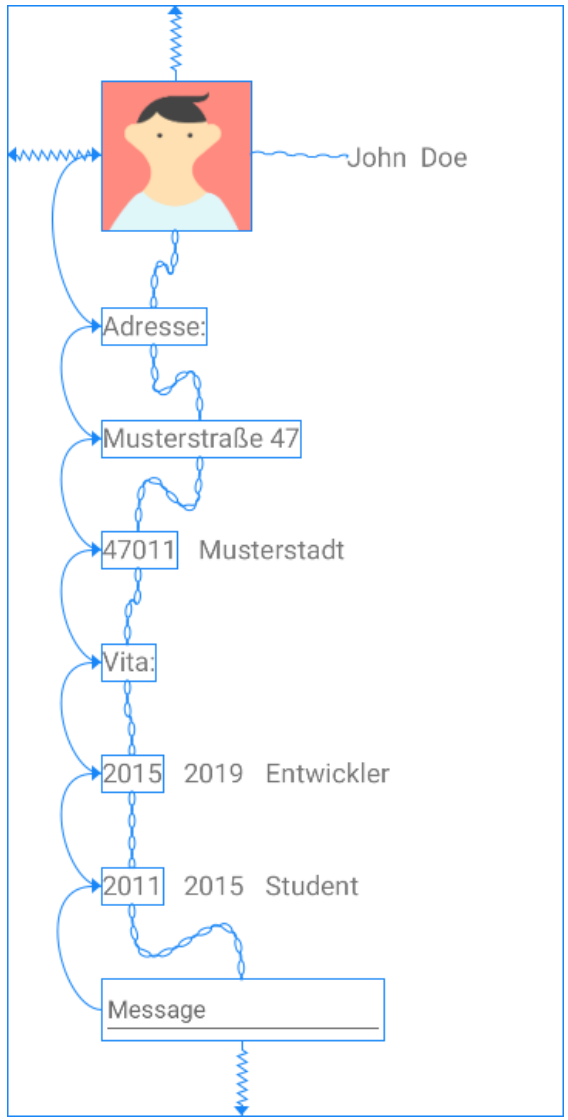




Festlegung der Constraints (Zeilen/Spalten)

5.6 Constraint Layout (Beispiel) - Forts.





Festlegung der Constraints (Vertikale/Horizontale Ketten)

5.6 Constraint Layout (Constraints)

Hinweis: Jedes Layout-Element muss mit ausreichenden Angaben zur Positionierung ausgestattet werden.

Im Hintergrund prüft eine Constraint-Checker, ob die Anforderungen an eine ausreichen Anzahl an Constraints erfüllt sind und markiert entsprechende Layout-Element die dies nicht erfüllen.

Werden die Hinweise ignoriert, positioniert das System die Layout-Element an der Position (0,0).

Es gibt unterschiedliche Constraints: Pfeil (Ausrichtung an weiterem Layout-Element mit konstanten Abstand), Feder (Ausrichtung zwischen zwei Layout-Elementen mit gewichteten Abstand), “Kette” (Vertikale o. horizontale Ausrichtung).

5.6 Constraint Layout (Empfehlungen)

Empfehlungen zur Erstellung von Constraint Layouts:

Jedes Layout-Element braucht mindestens einen horizontalen und einen vertikalen Bezugspunkt.

Zwei Bezugspunkte in der selben Dimension (vertikal o. horizontal) führen zur zentrierten (gewichteten) Anordnung.

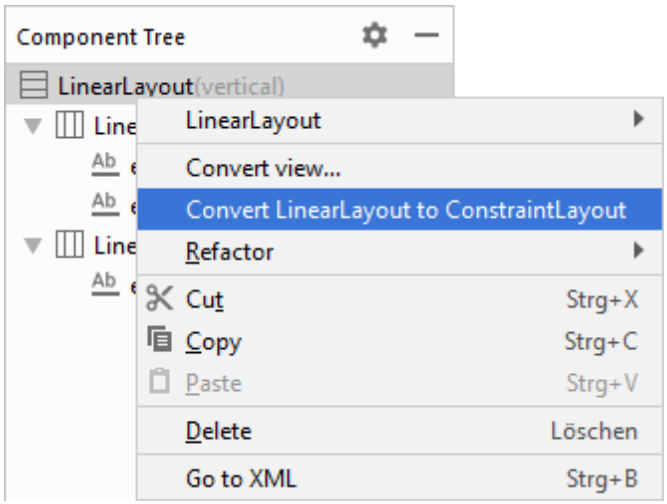
Mehrere Layout-Elemente können eine horizontale o. vertikale 'Kette' bilden. Die Elemente werden dann gleichmäßig verteilt. Achtung: Fehlt oder wird eine Verbindung in der Kette gelöscht verschiebt sich das gesamte Layout.

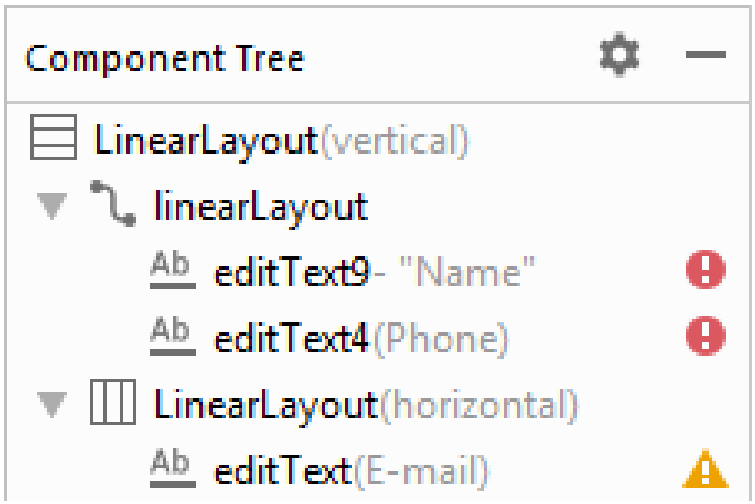
Für Höhen- und Breitenangaben gibt es drei Möglichkeiten:

- `wrap_content`
- eine absolute Größe (z.B. 100 dp)
- 0dp, dann orientiert sich die Größe an den Constraints

5.6 Constraint Layout (Konvertierung)

Android-Studio bietet die automatische Konvertierung verschiedener Layouttypen in andere Layouttypen an:

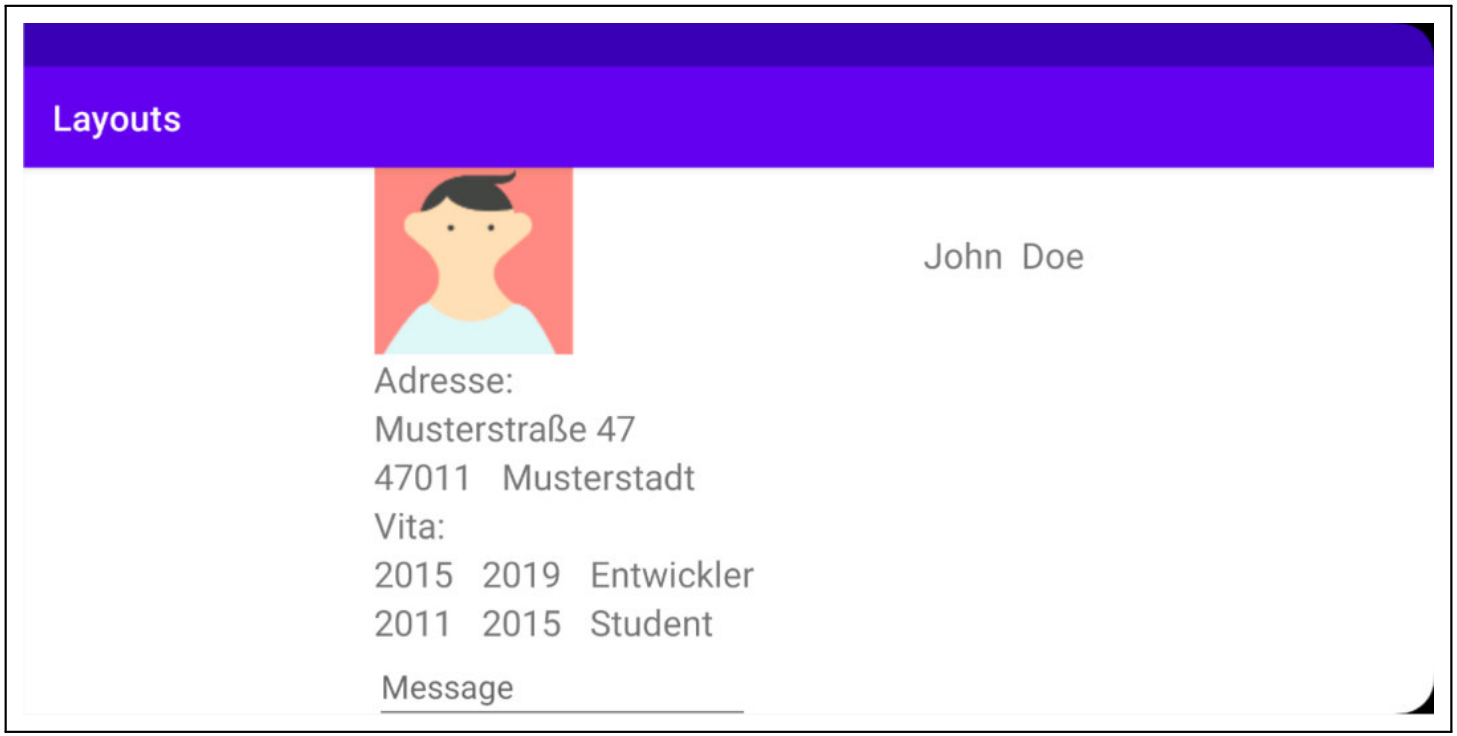




Die Funktion scheint allerdings noch Optimierungsbedarf zu haben.

5.7 Kontextspezifische Layouts (Problem)

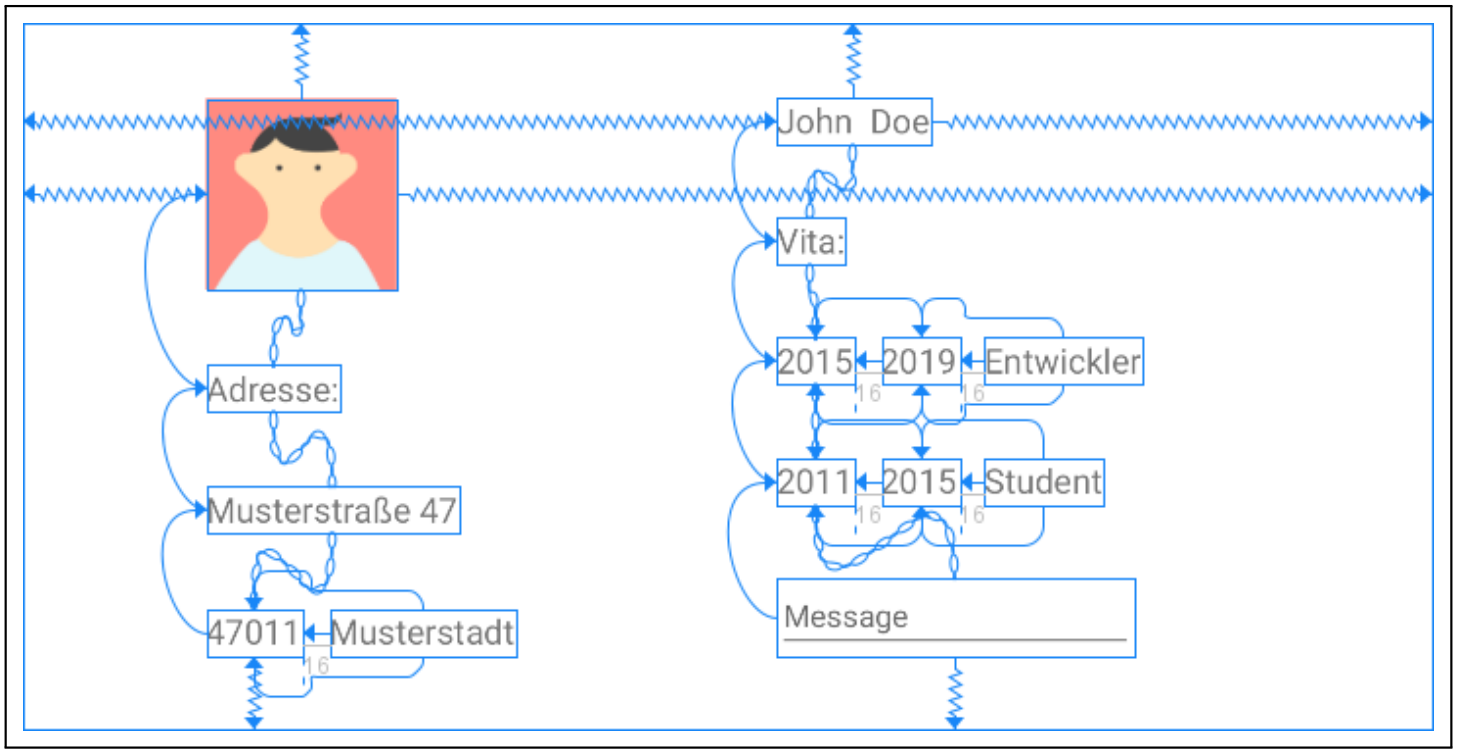
Die Darstellung im Landscape-Modus ist identisch, wobei die Zwischenräume der vertikalen Kette auf “0” reduziert werden.



Ungünstiges Layout im Landscape-Modus

5.7 Kontextspezifische Layouts (Lösung)

In dem Ressourcen-Ordner layout-land kann eine weitere Layout-Variante (mit identischen Namen z. B. `activity_main_constraintslayout.xml`) angegeben werden, die von Android automatisch bei einem Kontextwechsel (Portrait-Modus zu Landscape-Modus u. U.) geladen wird.



Layout-Variante für den Landscape-Modus

5.8 Ausblick

Neben den Linear-, Frame- und Constraint Layout gibt es noch weitere Layouttypen:

- Table Row Layout
- Relative Layout
- Table Layout
- Grid Layout

Die hier beschriebenen Technik beschreibt wie man ein Layout technisch realisieren kann.

Es gibt daneben noch umfassende Richtlinien zur Erstellung von Benutzerschnittstellen (z.b. EU-Richtlinie zur Barrierefreiheit von Websites und mobilen Anwendungen), die spezielle Anforderungen an die Benutzerschnittstellen stellen.

Block 5 – Zusammenfassung

- Es gibt verschiedene Arten in Android (Studio) Layouts anzulegen. Die gebräuchliste ist der **Android Design Editor**
- Es gibt einschlägige Layouttypen (z.B. **LinearLayout** o. **Frame Layout**) für unterschiedliche Zwecke. Diese lassen sich verschachteln
- Das **Constraint Layout** ist ein flexibler Layouttyp, der ohne eine Verschachtelungshierarchie auskommt
- Layouts sollten generell **relative** statt **absolute** Positionsangaben enthalten

Block 5 – Weitere Aufgaben

- Überführten Sie ein LinearLayout, FrameLayout oder ConstraintLayout in ein jeweils anderen Layouttypen. Nutzen Sie ggf. den Werkzeugunterstützung von Android Studio.
- Legen Sie für das gezeigte LinearLayout o. FrameLayout jeweils eine Layoutvariante für den Landscape-Modus an. Testen Sie den Mechanismus der Android Plattform die jeweils richtige Layoutvariante zu laden.
- Testen Sie, ob wie die Layouts auf unterschiedlichen (virtuellen) Geräten mit unterschiedlicher Anzeigegröße und Anzeigeauflösung dargestellt werden.

Block 5 – Literatur I