# grundkonzepte

**Single- vs. Cross-Plattform:**

Diagram

Description automatically generated

**Native-, Hybrid- und Web-Apps:**

Diagram

Description automatically generated

**Vor-/Nachteile:**

Table

Description automatically generated

**SP**: Singel-Plattform **CP**: Cross-Plattform

(1) Hybrid und Web Apps erreichen nativen Look & Feel über visuelles Styling

(2) Hybrid Apps greifen über native Libraries auf Device Features zu

**Fragmentierung:** Riesige Vielfalt an unterschiedlichen Geräten, Herstellern und OS-Versionen (mangels Updates wegen herstellerspezifische UIs). Dagegen kann man Compatibility Klassen von AndroidX verwenden, Version-Checks im Code oder Alternative Resources.

Apps bestehen aus lose gekoppelten, wiederverwendbaren Komponenten. Komponenten sind Activities, Content Providers, Services und Broadcast Receivers. Android hat die Kontrolle über ausgeführte Apps.

**Activity:** Besitzen eine grafische Oberfläche und verarbeitet Benutzereingaben.

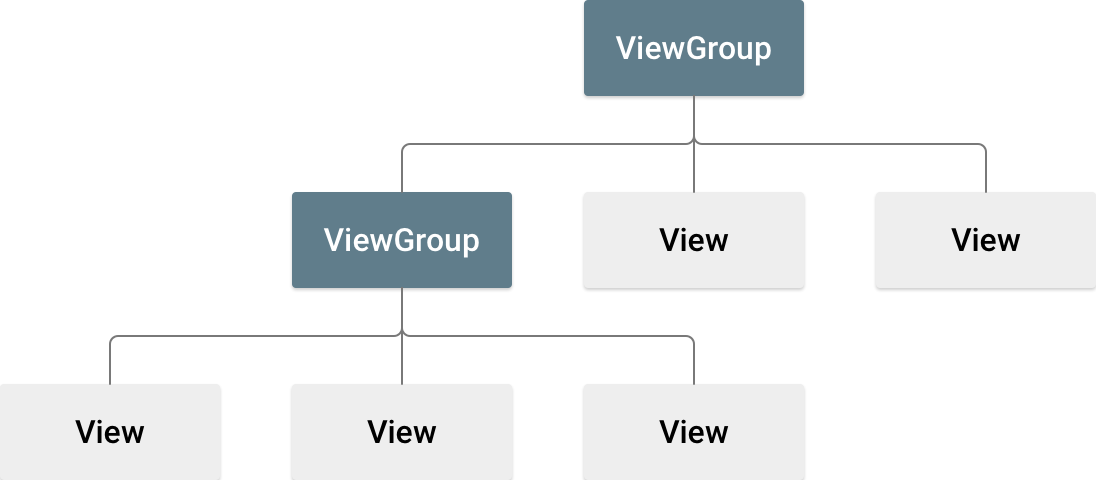
**Lifecycle:** Apps können verschiedene Zustände annehmen. Android ruft bei Zustandswechsel die Callback-Methoden der aktiven Activity auf.

Diagram

Description automatically generated

|  |  |
| --- | --- |
| Methode | Nutzen |
| onCreate() | Erzeugung des GUI |
| onPause(), onStop() | Datensicherung: onPause() für schnelle Operation |
| onResume, onPause() | Dienste wie Lokalisierung aktivieren/deaktivieren. Erst hier kann Activity User Events empfangen |
| onSaveInstanceState(), onRestoreInstanceState() | Zustand des GUI erhalten (z.B bei Rotation) |

**GUI:** Kann als Ressource im XML oder im Code beschrieben werden. GUI-Elemente werden hierarchisch angeordnet.

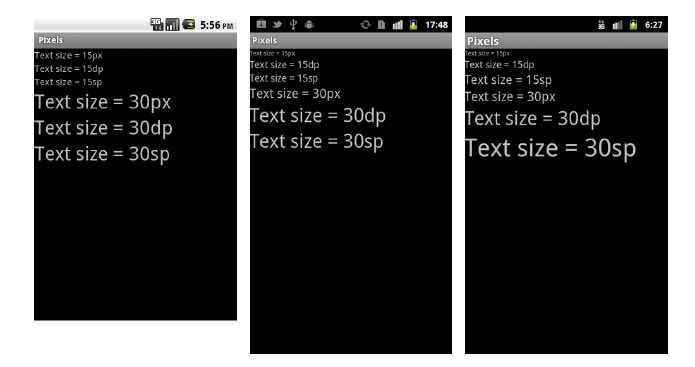


**Resources:** Alle Dateien, die keinen Code enthalten, werden als Resources bezeichnet. Zugriff erfolgt über Resource ID (int) via R-Klasse (vom Buildsystem generiert)

Button button = this.findViewById(R.id.button\_example);  
button.setText(R.string.example\_button\_2);

**Event-Handling:** Listener reagieren auf GUI-Ereignisse und werden bei GUI-Objekten registriert.

**Qualifier:** Die Auslagerung in XML-Dateien dient nicht nur der sauberen Trennung. Resources können in unterschiedlichen Varianten hinterlegt werden. (Text für verschiedene Sprachen, Bilder für verschiedene Auflösungen etc)

**Dimensionen:** Android erlaubt die Verwendung verschiender Dimensionen

**Android Manifest:** Das Manifest enthält essenzielle Informationen zur App: ID, Name, Version, Logo, enthaltene Komponente, Hard- und Softwareanforderungen etc.

**Intent:** Kommunikation zwischen Komponenten erfolgt über Intents. Explizite Intents sind Aufrufe bestimmter Komponente (Komponente eigener App), über Typ identifiziert. Implizite Itents sind Aufrufe einer passenden Komponente (Komponente aus anderen Apps), über Typ identifiziert

Diagram

Description automatically generated

*// Expliziter Intent*Intent secondActivityIntent = new Intent(this, SecondActivity.class);  
startActivity(secondActivityIntent);  
*// Impliziter Intent*Intent sendIntent = new Intent();  
sendIntent.setAction(Intent.*ACTION\_SEND*);  
sendIntent.setType("text/plain");  
sendIntent.putExtra(Intent.*EXTRA\_TEXT*, "Hey!");  
startActivity(sendIntent);

**Back Stack:** Ausgeführte Activities werden im Back Stack (Task) verwaltet. Mittels Overview Screen kann zwischen Tasks gewechselt werden.

Diagram

Description automatically generated

**Rückwärtskompatibilität:** API Levels ist die Zahl, die die Android Version identifiziert. Höhere API Level bedeutet weniger Geräte die unterstützt werden und umgekehrt.

**Android Jetpack:** Sammlung von Libraries, welche die App-Entwicklung vereinfachen. Übernimmt Rückwartskompatibilität bis zu API 8.

# gui

Die Basisklasse aller GUI-Elemente ist View. Belegt einen rechteckigen Beriech und kümmert sich um Darstellung un Event-Verarbeitung. Die Ableitung ViewGroup enthält View-Objekte. ViewGroup-Klassen ordnen Kinder nach einem Muster (Layout oder Container) und sind strukturierend, selber aber unsichtbar. Die View-Klassen implementieren das Composite Design Pattern.

## layouts

Einfache Layouts enthalten eine endliche Menge von Inhalten und können komplett im XML gestaltet werden. Höhe und Breite müssen auf jedem Element zwingend definiert sein.

**Layout Parameter:** Child-Views teilen ihrem Parent-Layout mit, wie sie angeordnet werden möchten. Dazu setzen diese auf sich selber spezielle Parameter mit dem Prefix layout\_ (layout\_width, layout\_height, etc.).

Diagram

Description automatically generated

**Padding/Margin:** Padding definiert den Leerraum innerhalb eines Elements. Margin definiert den Leerraum ausserhab des Elements

A picture containing application

Description automatically generated

Beim Rendering bestimmt jedes Layout, wieviel Platz seine Kinder einnehmen dürfen. Deshalb darf ein Element über den Platz innerhalb seiner Grenzen verfügen (Padding), nicht aber über den Platz ausserhalb (Margin).

A picture containing text, clipart

Description automatically generated**Linear Layout:** Kinder werden entweder vertikal oder horizontal angeordnet. Durch Gewuchte können die Grössen der Child-Elemente beeinflusst werden.

A picture containing logo

Description automatically generated**Frame Layout:** Kinder werden übereinander angeordnet. Nützlich für Live-Kamerabild oder Kartenansicht mit Zoom-Controls. Anpassung der Höhe über dem Bild. Standardmässig gilt die Reihenfolge im XML (grösser = höher)

Icon

Description automatically generated**Relative Layout:** Kinder werden relativ zueinander angeordnet. Identifizierung der referenzierten Kinder über Resource IDs. Sehr mächtig – dient oft als effizienter Ersatz für verschachtelte Linear Layouts. Jedoch negativer Einfluss auf Performance. Bevorzugt sind breite, flache Hierarchien.

**Constraint Layout:** Modernste und flexibelste Layout in Andoird. Grundidee ist ähnlich zum Relative Layout. Definieren von Beziehungen zwischen Views. Speziell für grosse, komplexe Layouts mit flacher Hierarchie entworfen.

Graphical user interface

Description automatically generated

**ScrollView:** Spezielles Layout mit nur einem Kind. Ergänzt mit einer Scrollbar und erlaubt das vertikale Scrolling. Horizontal Scrolling nur mit HorizontalScrollView möglich. Alternativ in AndroidX kann NestedScrollView verwendet werden.

**Darstellung von Collections:** Adapter vermittelt Darstellung und Datenquelle. Dank diesem Mechanismus bleibt Datenmodell frei von UI-Logik. View muss nicht neu erstellt werden, wenn die Datenquelle geändert wird, sondern nur der Adapter.

Diagram, text

Description automatically generated

**View Recycling:** Views werden recyclet, um Systemresourcen zu sparen.

Diagram

Description automatically generated

**RecyclerView:** Moderne Alternative zu ListView/GridView. Integriertes View-Recycling. Erzwungenge Verwendung von View Holdern. Weniger Overhead im eigenen Code und ist Teil von AndroidX. Die von der RecyclerView verwendeten Adapter implementieren View Recycling und View Holder Pattern in einer Basisklasse (RecyclerView.Adapter<T>). Die Basisklasse zwingt allen Ableitungen durch abstrakte Methoden und einen generischen Typparameter, die korrekte Implementierung auf. Diese gestaltet sich, danke der Vorarbeit der Basisklasse, allerdings sehr überschaubar.

## fragments

Je nach Formfaktor wäre es nützlich, mehrere Activities gleichzeitig anzugeigen. Dies funktioniert aber nicht mit Activities. Dafür gibt es Fragments. Ein Fragmeint ist ein modularer Teil einer Activity mit eigenem Lebenszyklus.

Chart

Description automatically generated

**Lifecycle:** Ähnlich wie Activity, aber umfangreicher. Zusätzliche Callbacks gegenüber Activity

* **onAttach:** Fragment an Activity angehängt
* **onCreateView:** UI des Fragments erstellen
* **onActivityCreated:** Activity wurde erzeugt
* **onDestroyView:** Gegenstück zu onCreateView
* **onDetach:** Gegenstück zu onAttach

Dynamisch eingebundene Fragmente sind austauschbar. Übergänge können animiert werden.

**Interaktion:** Wiederverwendbare Fragmente dürfen keine direkten Abhängigkeiten zur Activity haben. Interaktion von Activity mit Fragment über Parameter (Bundle-Objekt). Interaktion von Fragment mit Activity mit Callback-Interfaces.

Diagram

Description automatically generated

**Callback-Interfaces:**

**Diagram

Description automatically generated**

**Verschachtelung:** Fragmente sind verschachtelbar. Gleiches Vorgehen wie bei der Einbindung von Fragmenten in Activities. Zu viele Verschachtelungen kann zu Performance Problemen führen.

## Widget

Ein Sammelbegriff für visuelle Elemente des User Interfaces. Basisklasse ist View, nicht Widget. Verwechslungsgefahr mit Homescreen-Widgets.

**TextView:** Anzeige von Text mit verschiedenen Format-Attributen und Möglichkeit, integrierte Bilder zu verwenden

**ImageView:** Anzeige von Bildern. Source, Scale etc. können im XML angepasst werden.

**Button/ImageButton:** Lösen via Listener Aktionen aus. Ableitung von TextView bzw ImageView. Selbe Attribute/Methoden wie Basisklasen.

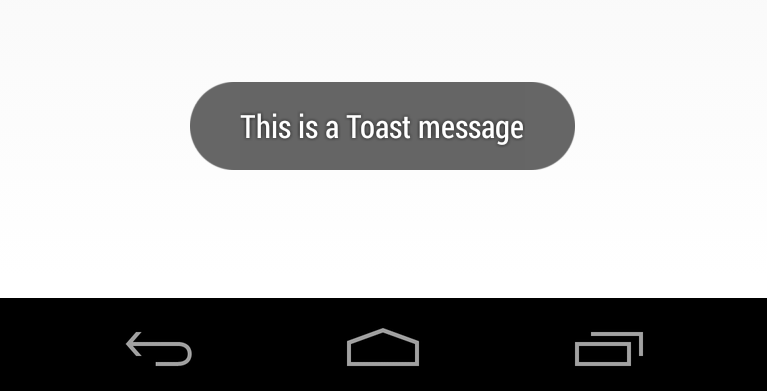
**EditText:** Dient der Eingabe von Text/Zahlen. Verhalten kann durch Attribute verändert werden (Format, Korrekturoptionen, Darstellung, etc.). Bei Texteingaben gibt es mehrere Möglichkeiten, auf Events zu reagieren.

Text

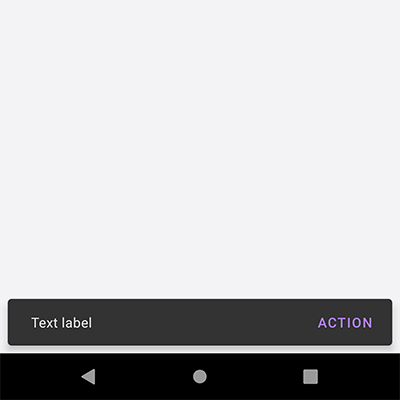
Description automatically generated

Android enthält UI-Widgets, welche direkt aus dem Code heraus erzeugt werden. Die Anpassbarkeit dieser Widgets ist oft eingeschränkt (Farben, Texte, etc.).

**Toasts:** Einfache Rückmeldung zu einem Vorgang. Darstellung in einem Popup-Fenster, Keine Interaktion für Benutzer möglich und verschwinden nach kurzer Zeit automatisch.



**Snackbar:** Selber Nutzen wie Toasts. Moderne Alternative für Toasts. Hier sind Interaktionen mit dem Benutzer möglich.



**Dialoge:** Dialoge erzwingen eine Aktion vom Benutzer. Füllen den Screen nicht vollständig und beinhaltet viele Anpassungsmöglichkeiten (Titel, Inhalt, Buttons, etc.)

Graphical user interface, text

Description automatically generated with medium confidence

**Notification:** Für Mitteilungen ausserhalb aktiver Nutzung.

Graphical user interface, text, application, chat or text message

Description automatically generated

**Menus:** Menus existieren in verschiedenen Varianten (Options Menu, App Bar, Contextual Menu etc.). Generell gilt für alle Menus, dass diese als Resource definiert werden.

## styling

Widgets können direkt im XML durch das Setzen von Attributen verändert werden. Mögliche Probleme bei umfangreichen Apps: Code-Duplizierung, Inkonsistenzen, Unübersichtlichkeit.

Mit Styles können wir Formatierungen wiederverwendbar machen.

<TextView *// layout.xml*  
 style="@style/HeaderText"  
 android:layout\_width="match\_parent"  
 android:layout\_height="wrap\_content"  
 android:text="Element 1" />  
  
<TextView  
 style="@style/HeaderText.Big"  
 android:layout\_width="match\_parent"  
 android:layout\_height="wrap\_content"   
 android:text="Element 2" />

<style name="HeaderText"> *// styles.xml*  
 <item name="android:textSize">24sp</item>  
 <item name="android:background">#ff9999</item>  
 <item name="android:padding">8dp</item>  
 <item name="android:layout\_margin">8dp</item>  
 <item name="android:gravity">center</item>  
</style>  
  
<style name="HeaderText.Big">  
 <item name="android:textSize">40sp</item>  
</style>

**Themes:** Themes sind spezielle Styles, die für eine ganze App oder einzelne Activities gelten. Themes können im Manifest für die ganze App (application) oder für einzelne Activites gesetzt werden (activity). Alternativ kann im Activity-Code ein Theme gesetzt werden.

<resources> *// styles.xml*  
 <style name="AppTheme" parent="…">  
 <item name="android:textViewStyle"> @style/MyText</item>  
 </style>  
  
 <style name="MyText">  
 <item name="android:textSize">24sp</item>  
 <item name="android:background">#ff9999</item>  
 <item name="android:padding">8dp</item>  
 <item name="android:layout\_margin">8dp</item>  
 <item name="android:gravity">center</item>  
 </style>  
</resources>

**Hierarchie von Styles:**

**Diagram

Description automatically generated**

**Material Design:** Eine Design-Language welche beim Designprozess helfen soll. Ziel ist ein konsistentes und benutzbares Look and Feel.

* **Material is the metaphor:** Inspiriert von der physischen Welt, Materialien reflektieren Licht und werfen Schatten (Verhalten von Controls, z.B Schatten)
* **Bold, graphic, intentional:** Basierend auf bewährten Prinzipien von Print-Medien (Layouts orientieren sich an einem 8dp-Raster)
* **Motion provides meaning:** Bewegung sollte zurückhaltend und nur zur Andeutung von Aktionen eingesetzt werden (Bestätigung von User Input, z.B Auswahl eines Elements)

# berechtigung

Apps dürfen keine Aktionen ausführen, die andere Dienste negativ beeinflussen könnten (Sandbox). Vor riskante Operationen müssen Berechtigungen angefragt werden (Kamera, ...).

Es gibt zwei Arten von Berechtigungen

* **Normal/Install-time:** Während der Installation beim System angefragt
* **Gefährlich/Runtime:** Während der Ausführung beim User angefragt

**Best Practice:** Nur anfordern, was benötig wird. Im Kontext der Verwendung anfordern, Transparente Erklärungen, Abbruch ermöglichen, Verweigerung berücksichtigen

**Manifest:** Alle benötigten Berechtigungen müssen im Manifest deklariert werden.

<uses-permission android:name="android.permission.CAMERA"/>

## persistenz

**Mechanismen :**

* **App-spezifische Dateien:** Beliebige Dateiformate für app-internen Gebrauch mit FileAPI (Domänenobjekte als JSON serialisiert)
* **Preferences:** Key-Value Paare für app-internen Gebrauch mit SharedPreferences (User Settings)
* **Datenbanken:** Strukturierte Daten für app-internen Gebrauch mit SQListe oder Room (Umfangreiche Domänenobjekte)
* **Medien:** Bilder, Dokumente, Musik und Videos. Austausch mit anderen Apps ohne UI-Dialoge mit MediaStore (Von App aufgenommene Fotos)
* **Dokumente:** Beliebige Dateiformate. Austausch mit anderen Apps mit UI-Dialogen mit Storage Access Framework (Von App erzeugte PDF)

**Interner Speicher:** Stets verfügbar. Geschützter Speicherbereich pro App mit begrenzten Speicher. Für app-interne Daten

**Externer Speicher:** Nicht immer verfügbar, da oft ein Wechseldatenträger. Emulation durch Android möglich. Primär für mit anderen geteilten Daten

**Graphical user interface, application

Description automatically generated**

## hardwarezugriff

Smartphones besitzen diverse Aktoren/Sensoren (Display, Kamera, Licht, Lage, etc.).

**Sensor Framework:** Framework für verschiedene Sensoren. Alle Sensoren werden gleich verwendet. SensorManager als Einstiegspunkt, Sensor als Repräsentant für realen Sensor, SensorEvent enthält Werte des Sensors und SensorEventListener für Callbacks.

**Beispiel:**

String service = Context.*VIBRATOR\_SERVICE*;  
int maxAmplitude = 255;  
Vibrator vibrator = (Vibrator) getSystemService(service);  
*// Ab API 26*long[] durs = new long[]{ 500, 500, 500, 500, 500 };  
int[] amps = new int[]{ 50, 100, 150, 200, 255 };  
VibrationEffect effect;  
effect = VibrationEffect.*createOneShot*(500, 255);  
effect = VibrationEffect.*createWaveform*(durs, amps, -1);  
vibrator.vibrate(effect);  
*// Ab API 29*int effectId = VibrationEffect.*EFFECT\_DOUBLE\_CLICK*;   
effect = VibrationEffect.*createPredefined*(effectId);  
vibrator.vibrate(effect);

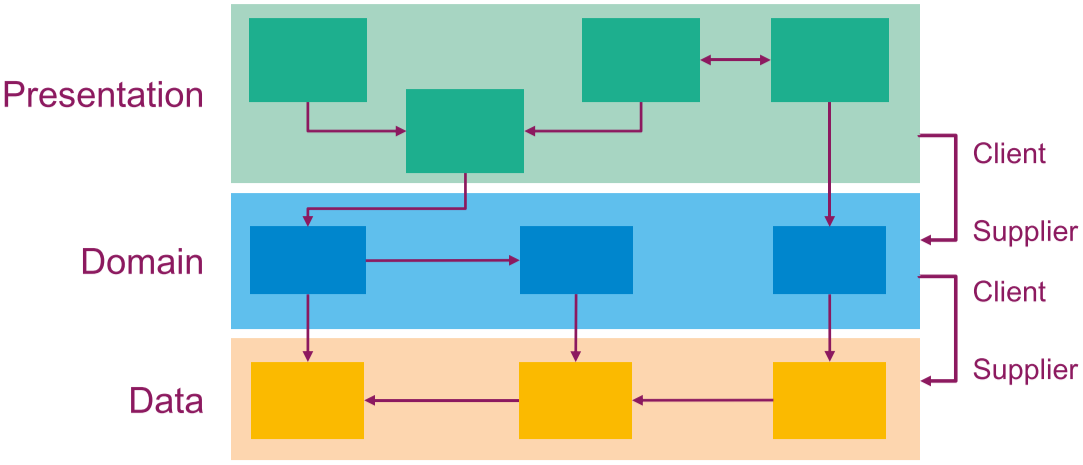
**REST-Kommunikation:** Für einfache REST-Calls existieren viele Varianten. HttpsURLConnection ist Teil der Android SDK, aber eher kompliziert. OkHttp und Retrofit sind effiziente Alternativen.

**Internet-Verbindung:** Im Gerät können viele Verbindungen gleichzeitig existieren. Android nutzt automatisch den besten Kanal. Statusänderungen kann via Broadcasts übermittelt werden.

# architektur

## schichtenarchitektur

Häufig angewendete Strukturierungshilfe für unsere Software. Schichten gruppieren zusammengehörige Konzepte. Keine Zyklen: Abhängigkeit nur auf darunterliegende Schicht.



**Presentation:** Darstellung und Interaktion mit Benutzer. Typischerweise stark an UI-Toolkit gebunden. Zugriff auf Domain-Schicht

**Domain:** Businesslogik und Domänenklassen, Keine UI-Funktionalität, Wenig externe Abhängigkeiten, Einfach zu testen

**Data:** Speicherung der Daten, Stellt Daten der Domain zur Verfügung, Auch Persistenz oder Datenhaltung genannt

Diagram

Description automatically generated

Schichtenarchitekturen finden sich oft in der Praxis, gelten aber als überholt. Moderne Literatur spricht von Ringen. Der Kern unserer Software ist die Domäne. Je zentraler, desto stabiler der Code. Je peripherer, desto volatiler der Code Technische "Details" liegen ganz aussen. Dazu gehört auch das User Interface.

## observer pattern

Wenn Schichtenarchitekturen keine Zyklen erlauben: Wie kann Domain dann die Presentation über Änderungen am internen Zustand informieren?

Diagram

Description automatically generated

Observer registrieren sich beim Subject für Statusänderungen. Observer kennt das Subject gut; umgekehrt weiss das Subject aber so gut wie nichts über seine Observer.

## model-view-controller (mvc)

Ein Pattern für die Organisation der Presentation. Model beinhaltet die Daten. View liest die Daten des Models und zeigt diese an. Controller koordiniert zwischen Model und View.

Text

Description automatically generated with low confidenceDiagram

Description automatically generated

**Kritik:** Viel Code im Controller und Controller wegen Referenz auf UI schwierig zu testen.

## fortgeschrittene themen

**Application:** Der Eltern-Knoten unserer Komponenten im Manifest heisst «application». Zu diesem Knoten wird beim Start eine Instanz der Application-Klase erzeugt. Eigene Ableitungen der Klasse möglich. Gut für einmalige Initialisierungen und Erzeugung von Singleton-Objekten.

<manifest xmlns:android="..." package="..."> *// Manifest*  
 <application android:name=".MyApplication">  
 *<!-- Unsere Komponenten -->* </application>  
</manifest>

*// MyApplication.java*

public class MyApplication extends Application {  
 @Override  
 public void onCreate() {  
 super.onCreate();  
 }   
 *// … mehr Callsbacks auf der nächsten Slide…*}

**Context:** Ermöglicht den Zugriff auf Dienste und Ressourcen der App. Context-Objekte haben eine eingeschränkte Lebensdauer. Angeforderte Ressourcen werden mit dem zugehörigen Context freigegeben

Table

Description automatically generated with medium confidence

**Broadcast:** Austausch von Meldungen zwischen Apps. Kann local oder global sein. Android sendet selbst globale Broadcasts (Internet verloren, SMS empfangen, etc.). Am besten keine sensitiven Daten in Broadcasts übermitteln.

**Services:** Ausführung von Aktionen im Hintergrund (Download, Streaming, Rechchenintensive Aufgabe, etc.). Lebenszyklus unabhängig von App.

**Started Service:** Für einmalige Aktionen und können potentiell endlos weiterlaufen. Beendigung durch Service selbst, Applikation oder Android

**Bound Service:** Für Aufgaben über längere Zeitdauer. Client-Server ähnliche Kommunikation. Austausch von Daten fester Bestandteile. Mehrere Clients gleichzeitig möglich.

# android jetpack

Android Jetpack ist eine Sammlung von Libraries von Google. Jetpack-Klassen sind unter androidx.\* definiert (darum der Name AndroidX).

## view binding / data binding

**ViewBinding:** Vereinfacht den Zugriff auf View-Elemente. Kein findViewById() mehr. Ist Typ- und Null-Sicherheit. Erzeugung von Binding-Klassen beim Build. Aktivierung über Gradle und kann für einzelne Layouts deaktiviert werden.

android **{** *// build.gradle*buildFeatures **{** viewBinding true  
 **}  
}**

<LinearLayout android:layout\_height="match\_parent" *// xml*  
 android:layout\_width="match\_parent">  
 <Button  
 android:id="@+id/button\_hello"  
 android:layout\_height="wrap\_content"  
 android:layout\_width="match\_parent"  
 android:text="Hello World!" />  
</LinearLayout>

public class MainActivity extends AppCompatActivity { *// Java*  
 private ActivityMainBinding binding;  
 @Override  
 protected void onCreate(Bundle savedInstanceState) {  
 super.onCreate(savedInstanceState);  
 LayoutInflater inflater = getLayoutInflater();  
 binding = ActivityMainBinding.inflate(inflater);  
 setContentView(binding.getRoot());  
 binding.buttonHello.setOnClickListener(v -> {});  
 }  
}

**Data Binding:** Erlaubt im XML den Zugriff auf Objekte. Layout als Observer der Daten. Einfache Logik direkt im XML. Basiert ebenfalls auf Binding-Klassen. Aktivierung über Gradle und wird beim Build generiert. Die Bindings im Layout werden in einer Expression Language definiert.

public class User { *// User.java*  
 public String firstName;  
 public String lastName;  
  
 public User(String firstName, String lastName) {  
 this.firstName = firstName;  
 this.lastName = lastName;  
 }  
}

public class MainActivity extends AppCompatActivity { *// Main*  
 private ActivityMainBinding binding;  
  
 @Override  
 protected void onCreate(Bundle savedInstanceState) {  
 super.onCreate(savedInstanceState);  
 LayoutInflater inflater = getLayoutInflater();  
 binding = ActivityMainBinding.inflate(inflater);  
 User user = new User("Thomas", "Kälin");  
 binding.setUser(user);  
 }  
}

<layout xmlns:android="…"> *// xml*  
 <data>  
 <variable  
 name="user"  
 type="ch.ost.rj.mge.v07.User" />  
 </data>  
 <LinearLayout  
 android:layout\_height="match\_parent"  
 android:layout\_width="match\_parent">  
  
 <TextView  
 android:id="@+id/first"  
 android:layout\_height="wrap\_content"  
 android:layout\_width="wrap\_content"  
 android:text="@{user.firstName}" />  
  
 <TextView  
 android:id="@+id/last"  
 android:layout\_height="wrap\_content"  
 android:layout\_width="wrap\_content"  
 android:text="@{user.lastName}" />  
  
 <TextView  
 android:layout\_height="wrap\_content"  
 android:layout\_width="wrap\_content"  
 android:text="@{first.text + ' ' + last.text}" />  
 </LinearLayout>  
</layout>

**Data Binding Event Handling:** Auch Events können verbunden werden. Listener Binding erlauben die Verwendung von Ausdrücken vor dem Aufruf der Methode

public class EventHandler { *// EventHandler.java*  
 public void doSomething(View view, String txt) {}  
 public void doSomething(View view) {}  
}

<Button  
 android:onClick="@{handler::doSomething}" *// xml*  
 android:text="Method Reference" />  
  
<Button  
 android:onClick='@{v -> handler.doSomething(v, "...")}'  
 android:text="Listener Binding" />

**Data Binding Änderungen:** Ein in Data Bindings verwendetes Objekt wird nicht automatisch observierbar. Für die automatische Aktualisierung der View muss die Datenquelle angepasst werden. Observable Fields/Classes für einzelne Werte/ganze Klassen. Normale Java-Observable sind inkompatibel.

public class User {  
 public final ObservableField<String> firstName = new …();  
 public final ObservableField<String> lastName = new …();  
 public final ObservableInt age = new …();  
}

**Gemeinsamkeiten:** Aktivierung via Gradle, Verwendung von Binding-Objekten und Vereinfachter Zugriff

## lifecycle-aware components

**Lifecycle-Aware Components:** Reagieren auf die Statusänderung einer anderen Komponente. LifecycleOwner verwalten ihren Status in einem Lifecycle-Objekt. LifecycleOberserver beobachten Lifecycle-Objekte. Dadurch verschiebt sich die Zustandslogik vom Owner bis Observer.

Diagram

Description automatically generated

Diagram

Description automatically generated

**LifeData:** Ist ein Lifecycle-Aware Observable. Alternative zu Observable Fields und Observable Classes, Als Datenquelle für Data Bindings verwendbar • Observer werden benachrichtigt, sofern diese aktiv sind (STARTED, RESUMED) und entfernt, wenn diese gestoppt werden

(DESTROYED)

public class UserActivity extends AppCompatActivity {  
 private ActivityUserBinding binding;  
 @Override  
 protected void onCreate(Bundle savedInstanceState) {  
 super.onCreate(savedInstanceState);  
 User user = new User("Thomas", "Kälin", 36);  
 UserViewModel viewModel = new UserViewModel(user);  
 binding = ActivityUserBinding.inflate(…); binding.setVm(viewModel);  
 binding.setLifecycleOwner(this); *// Setzen das LifecycleOwner für die Data Bindings in XML*  
 setContentView(binding.getRoot());  
 }  
}

public class UserViewModel {   
 private final User user;  
 public final MutableLiveData<String> name = new M…<>();  
 public final MutableLiveData<Integer> age = new M…<>();  
  
 public UserViewModel(User user) {  
 this.user = user;  
 name.setValue(user.name);  
 age.setValue(user.age);  
 }  
  
 public void incrementAge() {  
 int newAge = age.getValue() + 1;  
 age.setValue(newAge);  
 }  
  
 public void save() {  
 user.name = name.getValue();  
 user.age = age.getValue();  
 }  
}

**ViewModel:** Die abstrakte Klasse ViewModel löst das letztes Problem: die Rotation des Gerätes. ViewModel-Objekte hängen von LifecycleObjekten ab. Mehrfache Erzeugung des ViewModel mit demselben Lifecycle-Objekt liefert Singleton. Erst bei Zerstörung des Lifecycle-Objektes wird auch das ViewModel-Objekt freigegeben. An App-Lebenszeit geknüpfte View Models sind möglich (via Application-Objekt).

public class UserActivity extends AppCompatActivity {  
 private ActivityUserBinding binding;  
  
 @Override  
 protected void onCreate(Bundle savedInstanceState) {  
 super.onCreate(savedInstanceState);  
 User user = new User("Thomas", "Kälin", 36);  
 UVMFactory factory = new UVMFactory(user);  
 UserViewModel viewModel = new ViewModelProvider(this, factory).get(UserViewModel.class);  
 binding = ActivityUserBinding.inflate(…); binding.setVm(viewModel);  
 binding.setLifecycleOwner(this);  
 setContentView(binding.getRoot());  
 }  
}

public class UVMFactory implements ViewModelProvider.Factory {  
 private final User user;  
  
 public UserViewModelFactory(User user) {  
 this.user = user;  
 }  
  
 @Override  
 public <T extends ViewModel> T create(Class<T> class) {  
 return (T) new UserViewModel(user);  
 }  
}

public class UserViewModel { *// Wie oben* }

**ViewModel/Fragments:** Die Interaktion zwischen Fragments kann komplex werden. ViewModels können helfen, da pro Activity nur 1 ViewModel erlaubt ist und Fragmente Teile des ViewModels verwenden.

Table

Description automatically generated

**Vorteil MVVM:** Bei MVVM wird der «Controller» durch ein «View Model» ersetzt, welches keine Abhängigkeiten zu UI-Elementen enthält und dadurch einfacher zu testen ist.

**Nachteil MVVM:** Data Binding in Android basiert auf Code-Generierung (langsamere Compile-Time und schwerer zu debuggen). Ausserdem besteht durch die «Expression Language» die Gefahr, dass die Logik in der View platziert wird.

## repository pattern

Ergänzung zu Room, liefert LiveData-Objekte. Direkte Verwendung im ViewModel möglich. Das Repository Design Pattern dient dazu, die verwendete Persistenz-Technologie vor dem Verwender zu verstecken. Vorteile sind Austauschbarkeit der Persistenz, Bessere Testbarkeit des ViewModels, Internes Caching möglich und Erweiterbarkeit

**Diagram

Description automatically generated**

# .net maui

**NuGet:** Packet Manager im .NET-Umfeld

**XAML:** Beschreibungssrpache von Microsoft zur Gestaltung grafischer Oberflächen. Basiert auf XML und ist hierarchisch strukturiert. Code wird damit von Layout getrennt.

## funktionsweise

**Shared Code:** (1)Eigener Code verwendet nur abstrakte Elemente aus MAUI. (3) MAUI bildet diese abstrakten Elemente auf plattform-spezifische Elemente ab.

**Plattform-Code:** (2)Eigener Code verwendet Elemente aus einem der darunterliegenden UI-Frameworks. Dieser Code wird x-fach geschrieben

Graphical user interface, website

Description automatically generated

**Ausführung Android/IOS:**

Graphical user interface, application

Description automatically generated

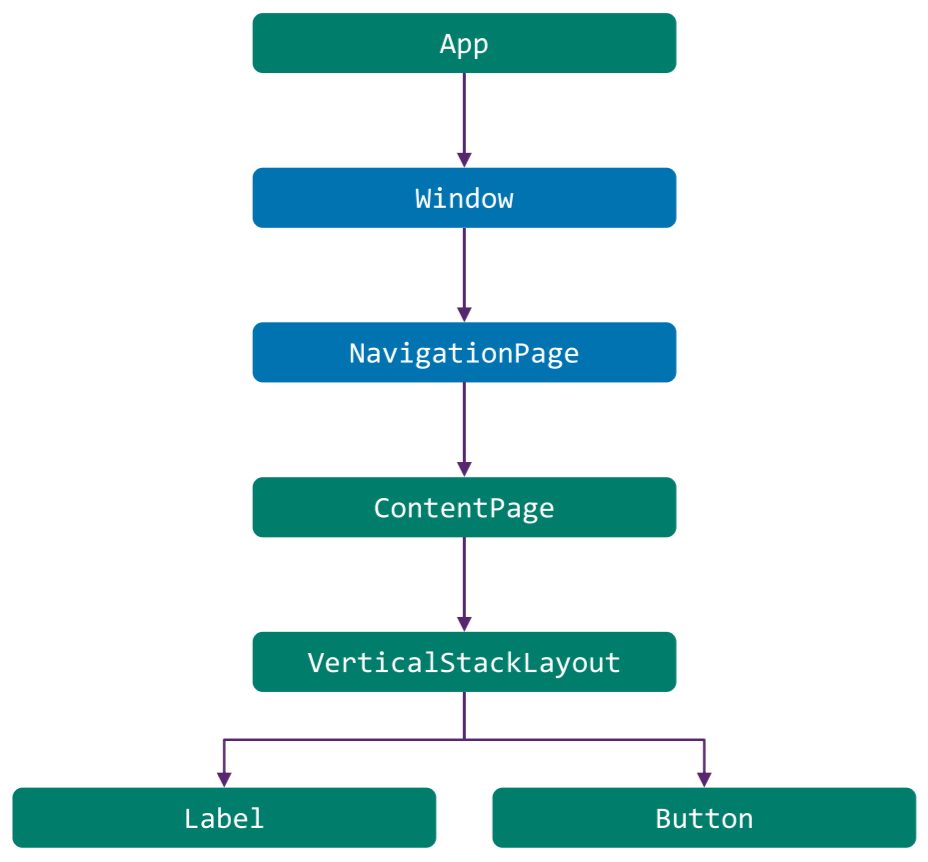
# xaml

XAML (Extensible Application Markup Language) ist die Beschreibungssprache von Microsoft zur Gestaltung grafischer Oberflächen (XML-basiert, hierarchisch strukturiert). Die XAML-Baustrukture kann auch in C# definiert werden.

**Vorteile von XAML:** Oft leichtgewichtiger, kürzer und lesbarer. Grafischer Designer inklusive Vorschau. Layout durch spezifische Tools erstellbar.

**Visual Tree (grün + blau):** Vollständiger, visuell dargesteller Baum. Enthält Knoten, die wir nicht selber definiert haben

**Logical Tree (grün):** Vereinfachung des vollen Baums, umfasst nur durch uns definierte Knoten



**Namespaces:** Mit xmlns werden XML-Namespaces definiert. Ohne Doppelpunkt: Standard-Namespace (Elemente können ohne Präfix verwendet werden). Mit Doppelpunkt: Benannter Namespace (Elemente müssen mit Präfix verwenden)

<Application xmlns="http://schemas.microsoft.com/dotnet/2021/maui"  
xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"  
x:Class="Vorlesung\_09.App">   
</Application>

Verwendung des Attributs «Class» aus dem Namespace «x». Hiermit wird die zugehörige Code Behind-Klasse definiert.

**Named Elements:** Elemente können benannt werden. Ermöglicht Zugriff im Code Behind. Attribut führt zu Property in generierter Klasse

<Label x:Name="NamedElement" /> *// XAML*

this.NamedElement.Text = "..."; *// Code Behind*

**Event Handler:** Reaktion auf Ereignisse der UI-Controls. Registrierung der Methode im XAML und Implementation im Code Behind.

<Button Clicked="Button\_OnClicked" Text="Click Me!" /> *// XAML*

private void Button\_OnClicked(object sender, EventArgs args) {  
 *// Wird beim Drücken des Buttons aufgerufen*}

**Content Property:** Jedes XAML-Element kann genau eine Eigenschaft als seinen Inhalt definieren. Dieser Inhalt kann in verkürzter Syntax «in das Element hinein» geschrieben werden. Fördert die Lesbarkeit von Parent/Child-Beziehung

<Label Text="Inhalt"/>  
<Label>Inhalt<Label/>

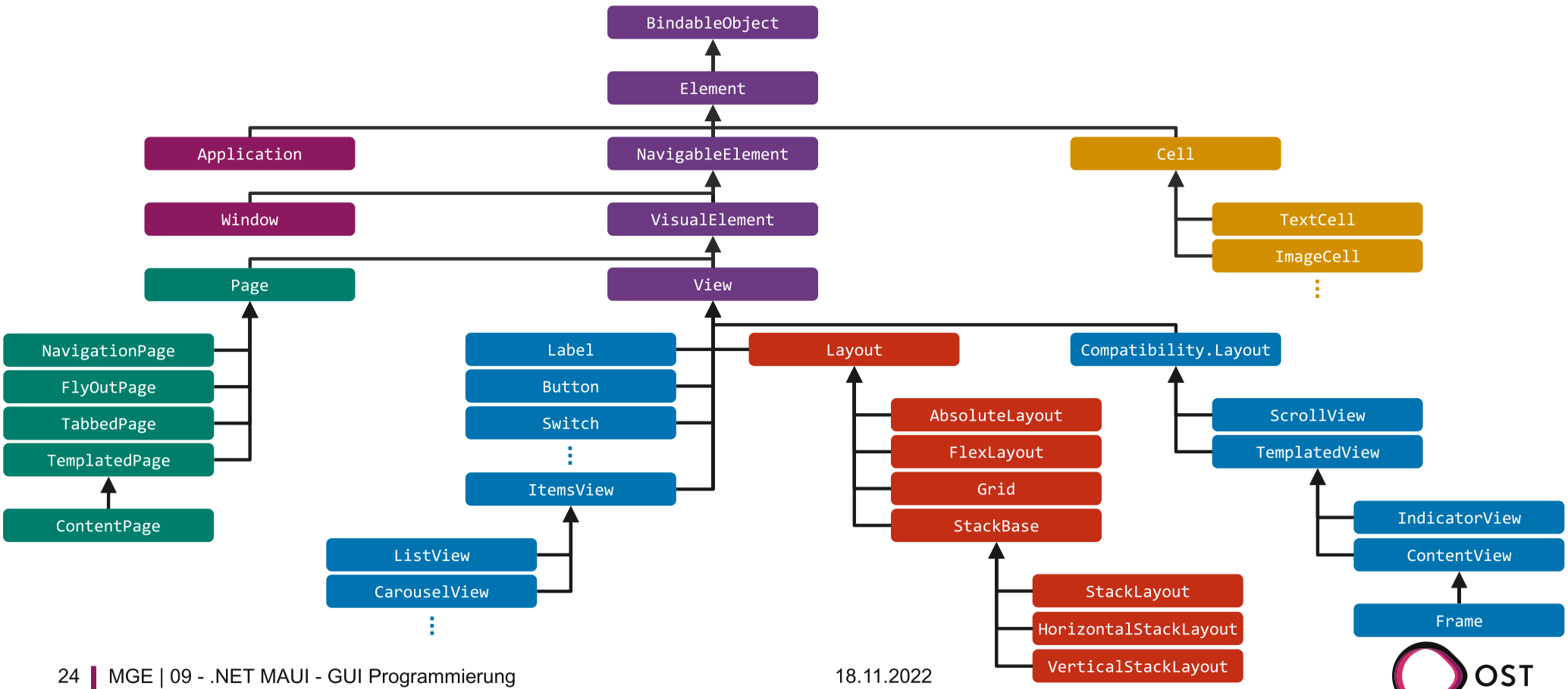
**Attached Properties:** Setzen einer Eigenschaft auf einem Element, welches zu einem anderen Element gehört. Die Eigenschaft wird einem anderen Element angehängt.

<Grid> *// XAML*  
 <Grid.RowDefinitions>  
 <RowDefinition Height="30" />  
 <RowDefinition Height="20" />  
 <RowDefinition Height="10" />  
 </Grid.RowDefinitions>  
 <Label Grid.Row="0" x:Name="R" Background="Re" />  
 <Label Grid.Row="1" x:Name="G" Background="Green" />  
 <Label Grid.Row="2" x:Name="B" Background="Blue" />  
</Grid>

**Markup Extensions:** Erlauben die Erweiterung des XAMLs mit zusätzlicher Logik.Die aufzurufende Funktion wird zwischen geschweiften Klammern platziert: { … }. Häufige Verwendung für Styling/Data Binding. Eigene Extensions auch möglich (Koordinaten)

<Label Text="{local:LocationExtension Lat=10,Long=20}" />

## gui Grundelemente

****

**Basisklassen:** Die abstrakten Basisklassen «im Zentrum» ergänzen schrittweise weitere Funktionalität. Diese Funktionalitäten stehen allen ableitenden GUI-Elementen zur Verfügung.

**Application:** Legt via MainPage den ersten im Hauptfenster angezeigten Screen fest. Erzeugung und Verwaltung von Fenstern. Erlaubt die Verarbeitung von LifecycleEvents in überschreibbaren Methoden, bei Multi-Window gilt das für alle Fenster.

**Window:** Die Klasse Window repräsentiert ein Fenster innerhalb der Anwendung. Fenster stellen Page-Objekte visuell dar. Hauptfenster: initialisiert durch MainPage. Zusatzfenster: initialisiert durch Konstruktor. Zusatzfenster werden über das Application-Singleton angezeigt

**App Lifecycle:** MAUI leitet wichtige Lifecycle-Events der darunterliegenden Plattformen weiter

Diagram

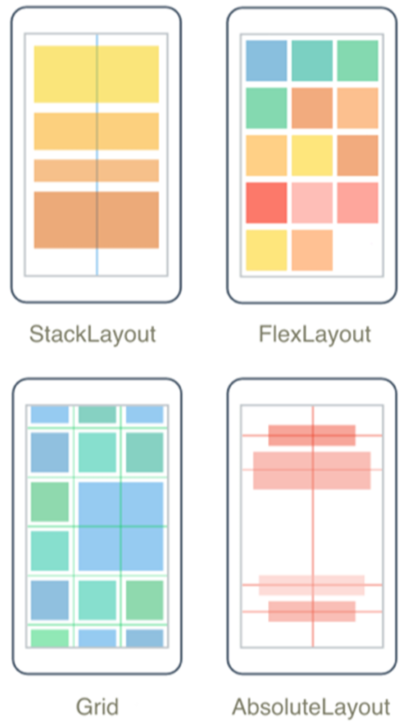
Description automatically generated

## gui pages, layout und views

**Pages:** Elemelnte zur Strukturierung und Gestaltung ganzer Screens

* Shape

  Description automatically generated ContentPage: Leere Screen ohne Zusatzelemente
* FlyoutPage: Slide-in Menu von links
* NavigationPage: Hierarchische Navigation mit Toolbar
* TabbedPage: Wechsel zwischen Inhalten mit Tabs

**Layouts:** Elemente zur Ausrichtung von Gruppierung von Views

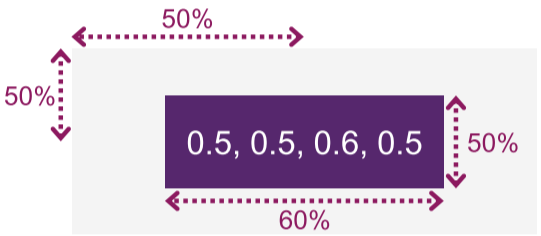
**StackLayout:** Horizontale oder vertikale Anordnung

**FlexLayout:** Ähnlich wie Stack, mit Wrapping und mehr Gestaltung

**Grid:** Anordnung in Zeilen und Spalten

**AbsoluteLayout:** Absolute oder proportionale Anordnung im Layout

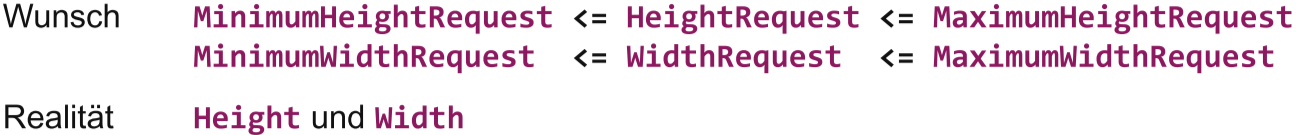
**AbsoluteLayout:**

A picture containing calendar

Description automatically generated

**Views:** Elemente, die für den Benutzer „interagierbar“ sind wie Label, Image, Entry, Button etc.

**Grössenangaben:** MAUI verwendet immer Device-Independent Units. 50 ist immer doppelt so gross wie 25. Grössenangaben sind Wünsche an die Rendering-Engine. Ohne Grössenangaben werden Views «so gross wie nötig» dargestellt



**Margin/Padding:** Margin und Padding sind nicht immer vorhanden

* **Pages** Padding
* **Layouts** Margin und Padding
* **Views** Margin und Padding

Mögliche Grössenangaben sind:

* **n** Selber Wert für alle Seiten
* **x, y** X für horizontal, Y für vertikal
* **l, t, r, b** Links, Oben, Rechts, Unten

## collections

Collections enthalten mehrere Elemente desselben Typs (Array<T> oder List<T>)

|  |  |
| --- | --- |
| ListView | Einfache Listen |
| TableView | Gruppierte Listen |
| Picker | Auswahl: 1 bis N |
| CarouselView | Horizontales Swiping |

public class User { *// User.cs*   
 public string FirstName { get; set; } = "Thomas";  
 public string LastName { get; set; } = "Kaelin";  
}

<ContentPage> *// XAML*   
 <ListView ItemsSource="{Binding}">  
 <ListView.ItemTemplate>  
 <xaml:DataTemplate x:DataType="model:User">  
 <TextCell Text="{Binding FirstName}" Detail="{Binding LastName}" />  
 </xaml:DataTemplate>  
 </ListView.ItemTemplate>  
 </ListView>  
</ContentPage>

public class MainPage : ContentPage{ *// Code Behind*   
 private ObservableCollection<User> \_users;  
  
 public MainPage(){  
 InitializeComponent();  
 *// Die Collection müsste hier noch befüllt werden…* \_users = new ObservableCollection<User>();  
 this.BindingContext = \_users;  
 }  
}

Text

Description automatically generated**ItemTemplate:** Das Item Template kann als Resource im Resource Dictionary definiert werden. Das Template wird so wiederverwendbar und der verwendende XAML-Code schlanker. Die Auslagerung ist insbesondere bei eigenen Cell-Definitionen sinnvoll. Durch das Attribut x:DataType ist IDE-Support gewährleistet

# view

## bilder

A picture containing application

Description automatically generatedBilddatei zum Projekt hinzufügen

|  |  |
| --- | --- |
| **Ordner** | Resources/Images |
| **Build Action** | MauiImage |
| **Formate** | JPG, PNG oder SVG |
| **Dateiname** | Lower Case, alphanumerisch + Underscore |



Aspect für Kontroller der Skalierung

|  |  |
| --- | --- |
| **AspectFit** | Bildverhältnis beibehalten (Standard) |
| **AspectFill** | Fläche füllen, Bildverhältnis beibehalten |
| **Fill** | Fläche füllen, Bildverhältnis ignorieren |
| **Center** | Bild gemäss Originalgrösse zentriert darstellen |

Text

Description automatically generatedImage-View besitzt einige weitere, nützliche Funktionen wie Anzeigen von GIFs inkl. Animation, Laden von Bildern aus Web, DLL/Assemblies und Byte Array

**App Icon/Splash Screen:** Auch ein App Icon und ein Splash Screen können als so genannte Resource Files einfach integriert werden. Nach Platzierung der Bild-Datei im Resources-Verzeichnis werden passende, plattform-spezifische Elemente erzeugt. Aufgrund der Skalierbarkeit werden auch hier SVG-Dateien bevorzugt. In der .csproj-Datei können Parameter konfiguriert werden (z.B. Hintergrundfarbe des Splash Screens)

## schriften

Die Integration von Schriften basiert ebenfalls auf Resource Files

|  |  |
| --- | --- |
| **Ordner** | Resources/Fonts |
| **Build Action** | MauiFont |
| **Formate** | TTF oder OTF |
| **Dateiname** | Keine Einschränkungen |

Nach einer Registrierung im Builder ist die Verwendung via FontFamily möglich.

var builder = MauiApp.CreateBuilder(); *// MauiProgram.cs*  
builder.UseMauiApp<App>().ConfigureFonts(fonts =>  
{  
 fonts.AddFont("Roboto-Bold.ttf", "RobotoBold");  
 fonts.AddFont("Roboto-Italic.ttf", "RobotoItalic");  
 fonts.AddFont("Roboto-Regular.ttf", "RobotoRegular");  
});

<Label Text="Roboto (Regular)" FontFamily="RobotoRegular" />  
<Label Text="Roboto (Bold)" FontFamily="RobotoBold" />  
<Label Text="Roboto (Italic)" FontFamily="RobotoItalic" />

Font Icons sind eine schlanke, gut skalierende Alternative zu Bildern. Integration als Schriftart – eine Schrift enthält tausende verschiedener Icons

<Image>  
 <Image.Source> *<!-- &#xf6e2; steht für \uf6e2 -->*  
 <FontImageSource FontFamily="FontAwesomeSolid"   
 Glyph="&#xf6e2;"   
 Color="White"   
 Size="50" />  
 </Image.Source>  
</Image>

## farben

Viele Views haben mehrere Varianten, wie Farben definiert werden können. Mittels Attribute auf Labels kann der Background vom Typ Brush und BackgroundColor vom Typ Color geändert werden.

Color-Objekte enthalten Farben, Brush-Objekte Farben oder Farbverläufe und werden durch Type Converter in SolidColorBrush-Objekte umgewandelt.

Shape

Description automatically generated

**Background via Type Converter:**

<Label Background="#6E1C50" />

**Background via Brush:**  
<Label.Background>  
 <SolidColorBrush Color="#6E1C50" />  
</Label.Background>

**Linear Gradient Brush:**  
<Label.Background>  
 <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">  
 <GradientStop Offset="0.0" Color="#6E1C50" /> *// A*  
 <GradientStop Offset="0.5" Color="#AE5C90" /> *// B*  
 <GradientStop Offset="1.0" Color="#6E1C50" /> *// C*  
 </LinearGradientBrush>  
</Label.Background>

Background pattern

Description automatically generated with low confidenceGraphical user interface, text, application

Description automatically generated

## rahmen / schatten

Zwei ähnliche Optionen:

* **Border** is a container control that draws a border, background, or both, around another control.
* **Frame** is used to wrap a view or layout with a border that can be configured with color, shadow, and other options.

Border ist mächtiger und moderner als Frame mit Formen, Farbverläufen, Linienmuster, Padding, etc.

Graphical user interface

Description automatically generated<Border StrokeThickness="4"   
 StrokeShape="RoundRectangle 20,20,20,20" *<!--Shape-Objekte ermöglichen die Beschreibung von geometrischen Formen.-->* Background="#D0A9D0">  
 <Border.Stroke>   
 <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">  
 <GradientStop Color="#6E1C50" Offset="0.0" />   
 <GradientStop Color="#56276D" Offset="1.0" />  
 </LinearGradientBrush>  
 </Border.Stroke>  
 <Border.Shadow>  
 *// Ableitungen von VisualElement besitzen die  
 // Shadow-Eigenschaft. Das Attribut Brush unterstützt  
 // aktuell nur SolidColorBrush-Objekte!*  
 <Shadow Brush="White"   
 Offset="0,0"   
 Opacity="0.8"   
 Radius="25" />   
 </Border.Shadow>  
 <Label Text="MGE"   
 TextColor="Black"   
 Padding="60,8"   
 FontSize="24"   
 FontFamily="OpenSansSemibold" />  
</Border>

## animationen

Alle VisualElement-Objekte unterstützen Überblendung (Fading), Skalierung (Scale), Rotation (Rotate) und Verschiebung (Translate). Alle Animationen unterstützen variable Zeitdauer (Standard: 250ms) und Easing (Standard: Linear)

*// Ghost ist ein benanntes XAML-Element (x:Name)   
// Task.WhenAll() führt die Animationen parallel aus  
// Normale await-Aufrufe führt die Animationen sequentiel aus*await Task.WhenAll(  
 Ghost.TranslateTo(Ghost.X + 100, Ghost.Y + 100, 750),   
 Ghost.RotateTo(360, 750),   
 Ghost.ScaleTo(1.5, 750),  
 Ghost.FadeTo(0.7, 750)  
);

## plattformspezifische anpassungen

Wunsch: eine UI-Definition, die auf allen Plattformen und Gerätetypen gut aussieht. In Realität ist das nicht immer möglich – manchmal müssen wir optimieren.

**Extensions:**

* **OnPlatform**: Werte pro Betriebssystem
* **OnIdiom**: Werte pro Gerätetyp

<Label Text="{OnPlatform '???', Android='Android', WinUI='Windows'}" />  
<Label Text="{OnIdiom '???', Phone='Phone', Desktop='Desktop'}" />

## resources

Eine Resource ist ein beliebiges Objekt, das in XAML definiert werden kann (Brush, Color, String etc.). Besitzt eine eindeutige Identifikation, zugewiesen mit XAML-Attributs x:Key.

**Verwendung von Resourcen:** Ziel ist, Objekte einmal definieren und x-fach wiederverwenden

A picture containing line chart

Description automatically generated

**Resource Dictionary:** Container zur Speicherung von Resources (Key-Value Speicher).

Attributs x:Key wird als Schlüssel verwendet.

In allen VisualElement-Ableitungen als Property Resources enthalten lokale Ressourcen

Ebenfalls auf der Application-Klasse als Property Resources definiert globale Ressourcen

**Beispiel:**

<ContentPage>  
 <ContentPage.Resources>  
 <SolidColorBrush x:Key="OSTBrush" Color="#6E1C50" />  
 </ContentPage.Resources>  
 <VerticalStackLayout>  
 <Label Text="Variante 1" TextColor="White">  
 <Label.Background> *// Property Element Syntax*  
 <StaticResource Key="OSTBrush" />   
 </Label.Background>  
 </Label>  
 <Label Text="Variante 2"   
 TextColor="White"   
 Background="{StaticResource Key=OSTBrush}"/>*//\**  
 <Label Text="Variante 3"   
 TextColor="White"   
 Background="{StaticResource OSTBrush}"/>*//\**

*// \*Attribute Syntax mit Markup Extension*  
 </VerticalStackLayout>  
</ContentPage>

Diagram

Description automatically generated**Auflösung von Resources:** Suchreihenfolge ist zuerst aktuelles Element und alle Parent-Elemente (entlang dem Visual tree in Richtung der Wurzel), danach in Application.Resources. Die Suche bricht beim ersten Treffer ab. Falls Schlüssel nicht gefunden wird, so wird der Fehler ignoriert (XAML) oder eine Exception geworfen (C#)

**Statische/Dynamische Resourcen:**

|  |  |
| --- | --- |
| Statisch | Dynamisch |
| Einmalige Auswertung der Resource | Mehrfache Auswertung der Resource |
| Auswertung bei Kompilierung | Auswertung bei Ausführung |
| Unveränderlich zur Laufzeit | Veränderlich zur Laufzeit |
| Extension: {StaticResource Key} | Extension: {DynamicResource Key} |

<ContentPage>  
 <ContentPage.Resources>  
 <SolidColorBrush x:Key="OSTBrush" Color="#6E1C50" />  
 </ContentPage.Resources>  
 <VerticalStackLayout>  
 <Label Text="OK"   
 TextColor="White"   
 Background="{DynamicResource OSTBrush}" />  
 <Button Text="Update" Clicked="UpdateResource" />  
 </VerticalStackLayout>  
</ContentPage>

private void UpdateResource(object sender, RoutedEventArgs e){  
 Resources["OSTBrush"] = new SolidColorBrush(Colors.Blue);  
}

**Eigenständige Resource Dictionaries:** Separate .xaml-Datei mit XML-Root <ResourceDictionary>. Code Behind-Datei ist optional. In andere Dictionaries als so genannte Merged Dictionaries integrierbar. Prioritäten bei Schlüsselkollisionen: 1. Lokale Resources

2. Merged Resources (Später definierte Merges überschreiben frühere Merges!)

<ResourceDictionary>  
 <SolidColorBrush x:Key="OSTBrush2" Color="#6E1C50" />  
</ResourceDictionary>

<ContentPage>  
 <ContentPage.Resources>  
 <ResourceDictionary>*//Bei “Merges” zwingend sonst optional*  
 <SolidColorBrush x:Key="OSTBrush" Color="#6E1C50"/>  
 <ResourceDictionary.MergedDictionaries>  
 <ResourceDictionary Source="MyDictionary.xaml"/>  
 </ResourceDictionary.MergedDictionaries>  
 </ResourceDictionary>  
 </ContentPage.Resources>  
 <Label Text="Externe Resource"   
 TextColor="White"   
 Background="{StaticResource OSTBrush2}" />  
</ContentPage>

**Beliebige Typen:** Ein Resource Dictionary nimmt alle Elemente auf, die in XAML definierbar sind. Häufig werden auch Basistypen benötigt (Strings, Zahlen, etc.). Diese können in XAML definiert werden.

Text

Description automatically generated with medium confidence

**x:Static:** Gelegentlich ist es nötig, auf statische Werte im C#-Code zuzugreifen. Konstanten aus .NET oder aus eigenem Code. Keine XAML-Resources, Werte definiert in C#-Code

namespace Resources.Examples; *// C#*   
public static class MyRes {  
 public static readonly SolidColorBrush OSTBrush3 = new(Color.FromArgb("6E1C50"));  
}

<ContentPage xmlns:local="clr-namespace:Resources.Examples">  
 <Label Text="Verwendung von x:Static"   
 Background="{x:Static local:MyRes.OSTBrush3}" />  
</ContentPage>

## styles

**Grenzen von Resources:** Mit Resources können mehrfach verwendete Werte zentral verwaltet werden. Aber: wir müssen die Werte weiterhin bei jedem Element referenzieren Das ist besonders ungünstig, wenn Werte für alle Elemente einer Applikation gelten sollen.

<ContentPage>  
 <ContentPage.Resources>  
 <Color x:Key="BlackColor">Black</Color>   
 <Color x:Key="BlueColor">LightBlue</Color>  
 <system:Double x:Key="MediumLine">2</system:Double>  
 </ContentPage.Resources>  
 <HorizontalStackLayout>  
 <Button TextColor="{StaticResource BlackColor}"   
 BackgroundColor="{StaticResource BlueColor}"  
 BorderColor="{StaticResource BlackColor}"   
 BorderWidth="{StaticResource MediumLine}"   
 Text="OK" />  
 <Button TextColor="{StaticResource BlackColor}"   
 BackgroundColor="{StaticResource BlueColor}"  
 BorderColor="{StaticResource BlackColor}"   
 BorderWidth="{StaticResource MediumLine}"  
 Text="Abbrechen" />  
 </HorizontalStackLayout>  
</ContentPage>

**Gestaltung mit Styles:**

**Diagram

Description automatically generated**

**Explizite Styles:** Explizit darum, weil wir die Resource explizit den verwendenden Views zuweisen müssen

<ContentPage>  
 <ContentPage.Resources>  
 <Style x:Key="MyButtonStyle"   
 TargetType="Button"> *// Typ der View, auf welcher der Style angewendet werden soll*  
 <Setter Property="TextColor"   
 Value="Black" />   
 <Setter Property="BackgroundColor"   
 Value="LightBlue" />  
 <Setter Property="BorderColor"   
 Value="Black" />   
 <Setter Property="BorderWidth"   
 Value="2" />  
 </Style>  
 </ContentPage.Resources>  
 <HorizontalStackLayout>  
 <Button Style="{StaticResource MyButtonStyle}" *// \**  
 Text="OK" />  
 <Button Style="{StaticResource MyButtonStyle}" *// \**  
 Text="Cancel" />

*// \*Zuweisung des Styles an alle verwendenden Views*  
 </HorizontalStackLayout>  
</ContentPage>

**Implizite Styles:** Ohne Key wirkt der Style automatisch für alle Controls des angegeben Typs

<ContentPage>  
 <ContentPage.Resources>  
 <Style TargetType="Button">  
 <Setter Property="TextColor"   
 Value="Black" />   
 <Setter Property="BackgroundColor"   
 Value="LightBlue" />  
 <Setter Property="BorderColor"   
 Value="Black" />   
 <Setter Property="BorderWidth"   
 Value="2" />  
 </Style>  
 </ContentPage.Resources>  
 <HorizontalStackLayout>  
 <Button Text="OK" />*// Kein Style-Attribute mehr nötig*  
 <Button Text="Cancel" />  
 </HorizontalStackLayout>  
</ContentPage>

**Styles erweitern:** Styles sind mit Inline-Attributen kombinierbar. Eignet sich für einmalige Anpassungen, z.B. das Hervorheben eines bestimmten Buttons. Styles können auch vererbt werden. Eine clevere Hierarchie kann den Umfang der Resourcen stark reduzieren

<ContentPage.Resources>  
 <Style x:Key="NormalButton" TargetType="Button">  
 ...  
 </Style>  
 <Style x:Key="DangerButton"   
 TargetType="Button"   
 BasedOn="{StaticResource NormalButton}">  
 <Setter Property="BackgroundColor" Value="Red" />  
 </Style>  
 OK Abbrechen  
</ContentPage.Resources>  
...  
<Button Style="{StaticResource DangerButton}"   
 Text="Abrechen" />

**Komplexe Werte:** Komplexe Werte in Styles sind über die Property Element Syntax möglich. Setzen des Value-Attributs auf dem Setter-Objekt. Selbe Syntax wie im normalen XAML

**Beispiel:** Button mit einem Gradient als Hintergrund

<ContentPage.Resources>  
 <Style x:Key="BrushButton" TargetType="Button">  
 <Setter Property="Background">  
 <Setter.Value>  
 <LinearGradientBrush StartPoint="0,0"   
 EndPoint="0,1">  
 <GradientStop Offset="0"   
 Color="Red" />   
 <GradientStop Offset="0.5"   
 Color="Yellow" />  
 <GradientStop Offset="1"   
 Color="Red" />  
 </LinearGradientBrush>  
 </Setter.Value>  
 </Setter>  
 </Style>  
</ContentPage.Resources>  
<Button Style="{StaticResource BrushButton}" Text="Brush" />

## theming

.NET MAUI kennt kein spezifischen Theming. Der Mechanismus kann aber nachgebaut werden.

1. Mehrere Resource Dictionaries mit identischen Schlüsseln darin
2. Laden des Standard-Theme als Merged Dictionary
3. Zugriff auf alle Ressourcen via DynamicResource
4. Laden eines neuen Dictionaries beim Wechsel des Themes

**Beispiel:** Light/Dark Theme

Icon

Description automatically generated

Graphical user interface

Description automatically generated

**Einstellung Betriebsystem:** Mögliche Verbesserung für unseren Code: Berücksichtigung der OS-Einstellungen. Laden eines passenden Theme beim App-Start. Wechsel des Theme bei Änderungen im OS. Glücklicherweise kennt MAUI passende APIs:

* **Aktuelle Systemeinstellung auslesen:** Application.Current.RequestedTheme
* **Event für bei Änderungen:** Application.Current.RequestedThemeChanged

public class ThemingPage : ContentPage{  
 public ThemingPage(){  
 InitializeComponent();  
 LoadThemeBasedOnOS();  
 Application.Current.RequestedThemeChanged += (\_, \_) => { LoadThemeBasedOnOS(); };  
 }  
  
 private void LoadThemeBasedOnOS(){  
 var activeTheme = Application.Current.RequestedTheme;  
 var isLightThemeActive = activeTheme == AppTheme.**Light**;  
  
 if (isLightThemeActive){  
 ActivateTheme<LightTheme>();  
 }  
 else{  
 ActivateTheme<DarkTheme>();  
 }  
 }  
}

**MarkupExtension AppThemeBinding:** Unsere eigene Lösung hat Vorteile wie zum Beispiel Erweiterbarkeit um zusätzliche Themes und saubere Trennung der Resources nach Theme.

Für einfache Apps kann diese Lösung aber zu kompliziert sein wenn nur Dark-/Light-Themes benötigt oder ein gemeinsames Dictionary gewünscht ist.

In diesen Fällen kann die Markup Extension AppThemeBinding verwendet werden. Automatische Anpassung auf Basis des OS.

<Label Text="{AppThemeBinding   
 Dark='Theme: Dark',   
 Light='Theme: Light',  
 Default='Theme: ???'}" />

# data binding

**Data Binding in Android:**

Shape

Description automatically generatedGraphical user interface, text, application

Description automatically generated

**Data Binding in .NET MAUI:**

public class User{  
 public string FirstName { get; set; } = "Thomas";  
 public string LastName { get; set; } = "Kälin";  
}

<ContentPage>  
 <StackLayout>  
 <Label Text="{Binding FirstName}"/> *//Data Binding mit*  
 <Label Text="{Binding LastName}"/>*// Markup Extension*  
 </StackLayout>  
</ContentPage>

public partial class MainPage : ContentPage{  
 private readonly User \_user;  
  
 public MainWindow() {  
 InitializeComponent();  
 \_user = new User();  
 this.BindingContext = \_user; *//Dataquelle setzen*  
 }  
}

**Bindings:** Path ist der Name der Quell-Eigenschaft, Objektpfad-Syntax möglich (x.y.z). Mode bestimmt die Richtung des Datenflusses. Standardwert unabhängig von Ziel-Eigenschaft. Converter ist Datenumwandlung zwischen Quelle und Ziel. Umwandlung in beide Richtung möglich.

*<!-- Attribute Syntax + Markup Extension -->*  
<Label Text="{Binding Path=FirstName,   
 Mode=**TwoWay**,   
 Converter={StaticResource MyCnv}}" />  
*<!-- Property Element Syntax -->*  
<Label>  
 <Label.Text>  
 <Binding Path="FirstName"   
 Mode="**TwoWay**"   
 Converter="{StaticResource MyCnv}" />  
 </Label.Text>  
</Label>

**Binding Mode:**

|  |  |
| --- | --- |
| OneTime | Einmalige Aktualisierung des Ziels beim Setzen der Quelle |
| OneWay | Ziel wird bei Änderungen der Quelle aktualisiert |
| OneWayToSource | Quelle Wird bei Änderungen des Ziels aktualisiert |
| TwoWay | Änderungen werden in beide Richtungen propagiert |
| Default | Wert abhängig von Ziel-Eigenschaft |

A screenshot of a computer

Description automatically generated with low confidence

**Binding Value Converter:** Hilfsobjekt zur Datenumwandlung zwischen Quelle und Ziel (bool zu Color oder Strings umformatieren)

*// Parameter der Methoden gekürzt zwecks Lesbarkeit*public class ReverseConverter : IValueConverter  
{  
 public object Convert(object value, …)  
 {  
 var stringValue = (string)value;  
 var reversedChars = stringValue.Reverse().ToArray();  
 var reversedString = new string(reversedChars);  
 return reversedString;  
 }  
  
 public object ConvertBack(object value, …)  
 {  
 return Convert(value, …);  
 }  
}

<ContentPage xmlns:converter="clr-namespace:(…)">  
 <ContentPage.Resources>  
 <converter:ReverseConverter x:Key="RConverter" />  
 </ContentPage.Resources>  
 <Entry Text="{Binding Path=FirstName, Mode=**TwoWay**,   
 Converter={StaticResource RConverter}}" />  
</ContentPage>

**Multi Binding:** Verwendung analog zu Binding (Path, Mode, Converter). Unterschiede sind beliebig viele Quell-Eigenschaften. Nur Property Element Syntax. Wichtig: Sofern die Ziel-Eigenschaft nicht vom Typ string ist, muss zwingend ein Multi Value Converter verwendet werden.

<Label>  
 <Label.Text>

*// {} nur nötig, wenn das erste Zeichen im Format- String eine geschweifte Klammer ist (Escaping)*  
 <MultiBinding StringFormat="{}{0} {1} ({2} Jahre)">  
 <Binding Path="FirstName" />   
 <Binding Path="LastName" />   
 <Binding Path="Age" />  
 </MultiBinding>  
 </Label.Text>  
</Label>

**Binding Context:** Property der Klasse BindableObject. Setzt die Standardquelle für Bindings. Falls undefiniert, traversiert den Logical Tree nach oben bis zum ersten Treffer. Jeder Path ist relativ zum BindingObject.

**Binding Context überschreiben:** Der Binding Context lässt sich für einzelne Elemente anpassen. Im Code Behind kann das Property BindingContext für das Element setzen oder über das Attribut im Binding setzen.

**RelativeSource**: Elemente im Visual Tree werden referenziert. Suche beginnt beim aufrufenden Element

<Label Text="{Binding Source={RelativeSource AncestorType=  
 {x:Type ContentPage}},   
 Path=Title}" />

**x:Reference:** Elemente werden über Namen referenziert. Namen müssen im gleichen Namespace vorliegen

<TextBox Name="MyText" Text="Hallo MGE" />  
<TextBox   
 Text="{Binding Source= {x:Reference Name=MyText},   
 Path=Text}" />

**Compiled Bindings:** Mit den bisherigen Mechanismen werden Bindings erst zur Laufzeit aufgelöst und kennt der XAML-Editor den Typ des Objektes im Binding Context (kein IDE Support). Beide Probleme können mit dem Attribut x:DataType adressiert werden. Vorteile sind die Auflösung der Bindings zum Zeitpunkt der Kompilierung. Bessere Performance beim Erstellen der Binding-Objekte. Auto Complete in IDE

## aktualisierung von daten

**POCO als Binding Context:** Als Datenquelle können beliebige Objekte verwendet werden, also auch POCO. Bindings funktionieren allerdings nur mit Einschränkungen.

<ContentPage>  
 <Label Text="{Binding Age}" /> *// Initialwert ist 38*  
 <Button Text="Alter ++" Clicked="Increment" />  
</ContentPage>

public partial class MainPage : ContentPage{  
 private readonly User \_user;  
  
 private void Increment(object sender, EventArgs e){  
 \_user.Age++;  
 }  
}

public class User{  
 public string FirstName { get; set; } = "Thomas";  
 public string LastName { get; set; } = "Kälin";  
 public int Age { get; set; } = 38;  
}

**INotifyPropertyChanged:** Interface mit genau einem Event. Name des geänderten Propery in EventArgs

public abstract class BindableBase : INotifyPropertyChanged{  
 public event PropertyChangedEventHandler PropertyChanged;  
  
 protected virtual void OnPropertyChanged(string name){  
 var eventArgs = new PropertyChangedEventArgs(name);  
 PropertyChanged?.Invoke(this, eventArgs);  
 }  
   
 protected bool SetProperty<T>(  
 ref T field,   
 T value,   
 [CallerMemberName] string name = null){

*// CallerMemberName wird zur Compile-Time in den Namen des Property umgewandelt*  
 if (Equals(field, value)) return false;  
 field = value;  
 OnPropertyChanged(name);  
 return true;  
 }  
}

public class User : BindableBase {  
 private string \_firstName = "Thomas";  
  
 public string FirstName {  
 get => \_firstName;  
 set => SetProperty(ref \_firstName, value);  
 }  
}

# mvvm

Ziel ist die Trennung von Präsentation und Logik.

Diagram

Description automatically generated

**Model:** Umfasst Domänen- und Businesslogik. Domänenmodelle mit oder ohne Verhalten. Bestandteile in C#-Klassen (ggf. Mit INPC oder INCC).

**View:** Kümmert sich um die Darstellung. Sollte möglichst dumm sein (Applikation sollte über View Models bedienbar sein). Logik und Zustände gehören ins View Model.

**View Model:** Enthält komplette Logik der Darstellung. Typische Aufgaben sind Formatierung von Model-Eigenschaftern, halten von Zuständen, Validierung von Benutzereingaben und Delegation von Benutzeraktion an Model.

## view models

**Klassisch:**

Graphical user interface, application

Description automatically generated

**Durchgriff:**

Graphical user interface, text, application

Description automatically generated

**Vergleich der Hauptvarianten:**

Table

Description automatically generated

**Aktionen in View Models:** Data Bidning erlaubt die Verknüpfung von Eigenschaften, nicht aber von Methoden. Lösung: Methoden in Objekte verpacken. ICommand definiert eine Schnittstelle für solche Objekte.

public sealed class RelayCommand : ICommand {  
 private readonly Action \_execute;  
 private readonly Func<bool> \_canExec;  
  
 public RelayCommand(Action execute, Func<bool> canExec){  
 \_execute = execute;  
 \_canExec = canExec;  
 }  
  
 public bool CanExecute(object parameter) => \_canExec();  
 public void Execute(object parameter) => \_execute();  
 public event EventHandler CanExecuteChanged;  
  
 public void RaiseCanExecuteChanged(){  
 CanExecuteChanged?.Invoke(this, EventArgs.Empty);  
 }  
}

public class UserViewModel : BindableBase{  
 public UserViewModel(User user){  
 DecreaseAgeCommand = new RelayCommand(OnDecreaseAge, CanDecreaseAge);  
 }  
  
 public int Age{  
 get => …; private set => …;  
 }  
  
 public ICommand DecreaseAgeCommand { get; }  
 private bool CanDecreaseAge() => Age > 0;  
  
 private void OnDecreaseAge(){  
 Age--;  
 DecreaseAgeCommand.RaiseCanExecuteChanged();  
 }  
}

**Relay Command:** Vorteile: ICommand Inteface einmalig implementiert. Universell verwendbar. Command-Code näher beim View Model. Nachteile des Relay Command sind keine wiederverwendbaren Command-Klassen

**Faustregel zur Zuteilung von Logik:**

Diagram

Description automatically generated

# architektur

## solution design

Graphical user interface

Description automatically generated

**Hauptgründe für Schnitte:** Fachliche, technische oder organisatorische Grenzen, Positiver Einfluss auf SW-Qualitätsmerkmale, Sorgen bei grossen Projekten für Überblick. Anzahl und Namen der Schnitte sekundär.

**Horizontale/Vertikale Schnitte:**

Diagram

Description automatically generated

**Horizontale Schnitte:** Traditioneller Ansatz, Geeignet für «Technologie Teams», Austausch von Technologien einfacher

**Vertikale Schnitte:** Modernerer Ansatz, Geeignet für «Feature Teams», Austausch von Technologien schwieriger

**Weitere Schnitte:**

**A picture containing text, calculator

Description automatically generated**

**Projektarten in Visual Studio:**

A picture containing calendar

Description automatically generated

## technologische grenzen

Model und View Model sollen technologie-neutral sein. Testbarkeit von Model und View Model. Wiederverwendbarkeit von Model und View Model. Austauschbarkeit von View

Graphical user interface, application, website

Description automatically generated

**SOLID:** Programmierung gegen Abstraktion (Abstrakte Klassen, Interfaces, Delegates)

**Beispiel Fehlermeldung anzeigen:**

Graphical user interface, text, application

Description automatically generated

**Beispiel mit Delegates:**

public class MauiPlatformSupport{ *// Platform Code*   
 public Task ShowErrorAsync(string title, string msg){  
 return MainPage.DisplayAlert(title, msg, "OK");  
 }  
}  
  
public partial class App : Application{ *// Glue Code*   
 public App(){  
 var platform = new MauiPlatformSupport();  
 var viewModel = new AsciiArtViewModel{  
 OnShowErrorAsync = platform.ShowErrorAsync  
 };  
 }  
}  
  
public class AsciiArtViewModel : BindableBase{ *// View Model*   
 public Func<string, string, Task> OnShowErrorAsync { … }  
  
 private async void OnCommandExecute(){  
 await OnShowErrorAsync("…", "…");  
 }  
}

**Beispiel mit Interfaces:**

public class MauiPlatformSupport : IErrorService{  
 public Task ShowErrorAsync(string title, string msg){  
 return MainPage.DisplayAlert(title, msg, "OK");  
 }  
}  
  
public partial class App : Application{  
 public App(){  
 var platform = new MauiPlatformSupport();  
 var viewModel = new AsciiArtViewModel(platform);  
 }  
}  
  
public interface IErrorService{  
 Task ShowErrorAsync(string title, string msg);  
}

public class AsciiArtViewModel : BindableBase{  
 private readonly IErrorService \_errorService;  
  
 public AsciiArtViewModel(IErrorService errorService){  
 \_errorService = errorService;  
 }  
  
 private async void OnCommandExecute(){  
 await \_errorService.ShowErrorAsync("…", "…");  
 }  
}

**Dependency Injection:** Client kennt Service nur als Interface, Injector erzeugt Service und Client, Injector injiziert Abhängigkeiten via Konstruktor, Methode und Property

Diagram

Description automatically generatedDiagram

Description automatically generated

**Vor- und Nachteile von DI:**

Graphical user interface, application

Description automatically generated

**DI-Container:** Nützlich bei grossen Projekten mit vielen, voneinander abhängigen Services. Kümmern sich um die Erzeugung der Clients mit ihren benötigten Services. Konfiguration des Containers nötig («Mapping von Interface zu Service»). Lebensdauer konfigurierbar (z.B. Singleton). Fehler in der Konfiguration wirken sich erst zur Laufzeit aus.

**Testing:** Unit Tests profitieren stark von DI. Test-Projekt als zusätzliche «View». Implementierung von «Test Doubles». Injektion von Test Doubles. Implementierung von Test Doubles (Selber implementieren oder Mocking Library verwenden)

*// Beispiel verwendet xUnit und FakeItEasy*public class AsciiArtViewModelTests {  
 [Fact]  
 public void Create\_WhenNoImageWasChosen\_ShowsError(){  
 *// Arrange* var fakeService = A.Fake<IErrorService>();  
 var testee = new AsciiArtViewModel(fakeService);  
 *// Act* testee.CreateCommand.Execute(null);  
 *// Assert* A.CallTo(() => fakeService.ShowErrorAsync("…", "…"))  
 .MustHaveHappened();  
 }  
}

## platform integration

**Umsetzung mit MAUI API:** .NET MAUI bietet für viele Anwendungsfälle bereits passende APIs. Einheitliche Schnittstelle für alle Plattformen. MAUI-interne Abbildung auf Plattform-APIs.

Diagram

Description automatically generated**Umsetzung mit Plattform-API:** Existieren keine APIs in MAUI, machen wir die plattform-spezifischen Services selber. Eine Implementierung pro Zielplattform und gutes Wissen über alle Zielplattformen nötig. Umsetzungsvarianten Conditional Compilation und Multi-Targeting

**Conditional Compilation:** Alle Implementierungen in einer Datei. Trennung der Plattformen über Direktiven: #if, #elif, #else und #endif. Führt schnell zu schlecht lesbarem Code

Graphical user interface

Description automatically generated with low confidence

**Multi Targeting:** Implementierung in mehreren Dateien. Trennung über Projekteinstellungen. Bevorzugter Weg in .NET MAUI

A screenshot of a computer

Description automatically generated with low confidence

## threading

**Main Thread in Android:** Beim Start einer Applikation erzeugt Android automatisch den Main Thread. Blockierung des Main Threads führt zum ANR- Screen (Application Not Responding).

Langlaufende Operationen immer in eigene Threads ausführen (Runnable)

**Main / Background Threads:** Viele UI-Frameworks kennen diese Trennung. Main Thread für die Aktualisierung des GUI und Background Thread für langlaufende Aktionen. Gilt auch für .NET MAUI, da es auf den plattform-spezifischen APIs aufsetzt

**Stolperfallen:**

1. Langlaufende Operationen auf dem MainThread blockieren die GUI
2. Aktualisierungen des GUI aus Background-Threads führen zu Exceptions

**Blockierung des GUI vermeiden:** .NET kennt verschiedene Threading-Mechanismen mit der Klasse Task, Keywords async/await, parallel LINQ. In MGE verwenden wir die Klasse Task, moderne Apps verwenden async/await.

**Aktualisierung des GUI vom Background-Thread:** MAUI enthält eine Platform Integration für das Problem. Via Klasse MainThread lässt sich Code an den Main Thread delegieren

public void RaiseCanExecuteChanged(){  
 CanExecuteChanged?.Invoke(this, EventArgs.Empty);  
 *// Main-Thread läuft hier \*parallel\* zum Task weiter C#* Task.Run(() => {  
 *// Ausführung auf Background-Thread  
 // Läuft \*parallel\* zum Main-Thread!* Thread.Sleep(2000);  
 MainThread.BeginInvokeOnMainThread(() =>{  
 *// Ausführung auf Main-Thread  
 // Hier darf das GUI aktualisiert werden* });  
 });  
}

**Auswirkung auf View Models:** Die gute Nachricht ist Data Binding erledigt die Delegation zum Main-Thread automatisch und Properties können also ohne Weiteres von einem Background-Thread verändert werden. Die schlechte Nachricht ist ICommand.CanExecuteChanged-Events müssen an den Main-Thread delegiert werden Tipp: RelayCommand entsprechend erweitern

*// IsCalculating ist ein einfaches ViewModel-Property*IsCalculating = true;  
Task.Run(() => {  
 *// Keine Delegation an Main-Thread nötig* IsCalculating = false;  
});

*// Einmalige Initialisierung in Application:*RelayCommand.Dispatch = MainThread.BeginInvokeOnMainThread;  
public sealed class RelayCommand : ICommand{  
 public static Action<Action> Dispatch { get; set; }

public void RaiseCanExecuteChanged(){  
 Dispatch(() => CanExecuteChanged?.Invoke(this, EventArgs.Empty));  
 }  
}

## mehrsprachigkeit

**Variante Resources:** Strings in Resource Dictionaries. Pro Sprache eine XAML-Datei. Zugriff im XAML über {DynamicResource …}. Zugriff in C# über Resources[]. Aktualisierung der Resources bei Sprachänderung aus dem Code Behind

<ResourceDictionary>  
 <system:String x:Key="Key1">Translation 1</system:String>  
</ResourceDictionary>

<ContentPage>  
 <Label Text="{DynamicResource Key1}" />  
</ContentPage>

public void LoadTranslations(string key){  
 var assembly = "MyApp";  
 var type = $"MyApp.Translations\_{key}";  
 var rd = Activator.CreateInstance(assembly, type).Unwrap();  
 foreach (var rdKey in rd.Keys) {  
 Resources[rdKey] = rd[rdKey];  
 }  
}

**Variante RESX:** Strings in RESX-Dateien. Pro Sprache eine RESX-Datei. Zugriff im XAML über {x:Static …}. Zugriff in C# über generierte Klasse. Anpassen der «Culture» bei Sprachänderung

in beliebigen C# Code

<?xml version="1.0" encoding="utf-8"?> *// Translation.resx*  
<root>  
 <data name="Key1" xml:space="preserve">  
 <value>Translation 1</value>  
 </data>  
</root>

<ContentPage xmlns:t="clr-namespace:MyApp.Translations">*//XAML*  
 <Label Content="{x:Static t:Translations.Key1}" />  
</ContentPage>

public void LoadTranslations(string key) { *// Code behind*  
*// Key kann z.B. "de" oder "en-US" sein Translations.Culture = new CultureInfo(key);*}

**Vergleich der Varianten:**

A picture containing table

Description automatically generated

## custom controls

Leere Ableitung von MAUI-View . Übernimmt alle Attribute der Basisklasse. Eigene Attribute sind möglich. Verwendung via Klassennamen nach Import des Namespaces im XAML. Präfix wie gewohnt frei wählbar. Auch als Typ für implizite Styles möglich

public class AlertLabel : Label { }

Graphical user interface

Description automatically generated<ContentPage xmlns:cc="clr-namespace:V\_14.CustomControls">  
 <ContentPage.Resources>  
 <Style TargetType="cc:AlertLabel">  
 <Setter Property="TextColor"   
 Value="White" />   
 <Setter Property="BackgroundColor"   
 Value="DarkRed" />  
 </Style>  
 </ContentPage.Resources>  
 <VerticalStackLayout Spacing="10">  
 <cc:AlertLabel Text="Label 1" />   
 <cc:AlertLabel Text="Label 2" />  
 </VerticalStackLayout>  
</ContentPage>

**Control Templates:** Die View soll neu abgerundete Ecken haben. Ein Label selbst kann das nicht, sondern muss in ein Border verpackt werden. Anders gesagt: wir müssen die visuelle Repräsentation der View steuern können.

**Bindable Properties:** Die View soll ausserdem ein Property mit dem Namen Message

bekommen. Benötigt so genannte Bindable Properties.

**Beispiel:**

public partial class AlertBox : ContentView{  
 public AlertBox(){  
 InitializeComponent();  
 }  
  
 public static readonly BindableProperty MessageProperty =   
 BindableProperty.Create(nameof(Message), *// Name* typeof(string), *// Datentyp* typeof(AlertBox), *// Besitzer* string.Empty); *// Initialwert* public string Message{  
 get => (string)GetValue(MessageProperty);  
 set => SetValue(MessageProperty, value);  
 }  
}

A picture containing text, screenshot, clipart

Description automatically generated<ContentView>  
 <ContentView.ControlTemplate>  
 <ControlTemplate>  
 <Border BackgroundColor="DarkRed"   
 StrokeShape="RoundRectangle 10,10,10,10"  
 Padding="5">  
 <Label Text="{TemplateBinding Message}"   
 TextColor="White"   
 HorizontalTextAlignment="**Center**" />  
 </Border>  
 </ControlTemplate>  
 </ContentView.ControlTemplate>  
</ContentView>

<ContentPage xmlns:cc="clr-namespace:V\_14.CustomControls">  
 <VerticalStackLayout Spacing="10">  
 <cc:AlertBox Message="Box 1"/>  
 <cc:AlertBox Message="Box 2"/>  
 </VerticalStackLayout>  
</ContentPage>

**Control Presenter:** Mit dem ContentPresenter kann man beliebige XAML-Elemente statt einfachem Text darstellen. Fungiert als Platzhalter, gibt den Inahlt der Content-Eigenschaft aus. Kein eigenes Bindable Property nötig.

<ContentView>  
 <ContentView.ControlTemplate>  
 <ControlTemplate>  
 <Border BackgroundColor="DarkRed"  
 StrokeShape="RoundRectangle 10,10,10,10"  
 Padding="5">  
 <ContentPresenter />  
 </Border>  
 </ControlTemplate>  
 </ContentView.ControlTemplate>  
</ContentView>

<ContentPage xmlns:cc="clr-namespace:V\_14.CustomControls">  
 <cc:AlertBorder>  
 <Grid>  
 <Image Source="…" />

<Label Text="Border 1" />  
 </Grid>  
 </cc:AlertBorder>  
</ContentPage>

**Handlers:** Bisher haben wir uns innerhalb der Grenzen von .NET MAUI bewegt. Was aber, wenn die native Repräsentation einer MAUI-View beeinflusst werden soll? Mit einem Hanlder kann man die MAUI-View auf die native Control abbilden. Anpassung oder eingene Handler möglich.

Diagram

Description automatically generated

## shell

Shell wird für die Vereinfachung von Anforderungen verwendet. Definiert eine Hierarchie in XAML, Navigation mit URI und Suchfelder für Listen. Verwendung ist optional.

A picture containing graphical user interface

Description automatically generated

**Strukturierung:** Hauptvarianten sind Flyouts, Tabs und die Kombination davon. Anhand der Struktur im XAML wird ein passendes UI erzeugt

Graphical user interface, text, application, Teams

Description automatically generated**Flyout:** Anzeige via Hamburger-Icon. Verfügt über 4 Bereiche

* **Flyout Items:** Seiten inkl. Verschachtelung
* **Menu Items:** Aktionen mit Commands
* **Header:** Sektion zu Beginn des Flyouts
* **Footer:** Sektion am Ende des Flyouts

**Möglichkeiten für visuelle Anpassungen:**

* Breite/Höhe des Flyouts
* Darstellung der Listeneinträge
* Farben, Schriften und Icons
* Scrolling der Inhalte
* Etc.

**Tabs:**

|  |  |
| --- | --- |
| Variante 1 | Variante 2 |
| Deaktiviert Flyout | Ermöglicht die Kombination mit Flyout |
| Ein Screen mit den darzustellenden Tabs | Auswahl des Flyout-Items navigiert zu Screen mit Tabs |

**Graphical user interface, text, application

Description automatically generated**

**Pages:** Pages werden in ShellContent gekapselt. LazyLoading via ContentTemplate, EarlyLoading via Content. Viele «Early Loading»-Elemente führen zu einem langsamem App-Start. Pages können via Attached Properties die Navigation beeinflussen

<Shell xmlns:p="clr-namespace:Vorlesung\_14.Pages">  
 *<!-- Lazy Loading bei Click -->*  
 <ShellContent Title="Seite 1"   
 ContentTemplate="{DataTemplate p:Page1}" />  
 *<!-- Early Loading bei App Startup -->*   
 <ShellContent Title="Seite 2">  
 <p:Page2 />  
 </ShellContent>  
</Shell>

Graphical user interface, text, application

Description automatically generated

**Navigation:** Automatische Navigation. Ein-/Ausblenden des Flyouts via Hamburger Icon. Wechsel von Inhalten bei Auswahl eines Items. Wechsel von Inhalten bei Auswahl eines Tabs. Optional Navigation mit URLs auch möglich

<Shell>  
 <FlyoutItem Route="fo1">  
 <Tab Route="tab1" />   
 <Tab Route="tab2" />  
 </FlyoutItem>  
</Shell>

*// Absolute Navigation*

Shell.Current.GoToAsync("//fo1/tab2");  
*// Eine Stufe zurück, d.h. zu «fo1»*await Shell.Current.GoToAsync("..");

**Hierarchische Navigation:** Hierarchische Navigation (Stack) erfolgt wie gehabt via Inavigation. Nützlich, wenn innerhalb eines Flyout-Items oder Tabs «in die Tiefe» navigiert wird

private async void ButtonClicked(object s, EventArgs e){  
 await Navigation.PushAsync(new SecondPage());*// First Page*  
}  
  
private async void ButtonClicked(object s, EventArgs e){  
 await Navigation.PopAsync(); *// Second Page*  
}

**Suchfelder:** Anzeige eines Suchfeldes in der Navigation Bar. Konzept ähnlich zu Adaptern im Android-Teil. Vorgehen für Umsetzung:

1. Ableitung von SearchHandler in C# implementieren
2. Search Handler im XAML via Attached Property
3. Item Template für Suchergebnisse in XAML gestalten, analog zu Collections

## xaml in anderen ui-Frameworks

A picture containing diagram

Description automatically generated

**MAUI vs WPF:**

|  |  |
| --- | --- |
| MAUI | WPF |
| XAML Features (Attribut Syntax etc) | WPF unterstützt nur Windows |
| Application-Klasse als Einstiegspunkt | Kein MauiProgram als Builder |
| MVVM inkl. Data Binding und Commands | Keine Platform-Ordner in Solution |
| Main- und Backgroundthreads | Keine Pages, nur Window |
| Mehrsprachigkeit mit Resources | Andere XAML-Namespaces |
|  | Navigation zwischen Windows |
|  | Kein Hot Reloading, dafür visueller Designer |

**Framework für Mobile:**

Diagram

Description automatically generated