1 Motivation und Einführung

Single- vs. Cross-Plattform:
Single: Codebasis für jede Plattform
Cross: Shared Code + Platform Code
Native-, Hybrid-, Web-Apps:
Native: Plattform(NativeApp(Binary))
Hybrid: Plattform(NativeApp(HTML))
Web: Plattform(Web Browser(HTML))
Vorteil Native Apps:

- Voller Funktionsumfang
- keine Tools/Einschränkungen von Drittanbietern

2 Grundkonzepte

Apps bestehen aus lose gekoppelten, wiederverwendbaren Komponenten (Activities, Content Providers, Services & Broadcast receivers).
Android hat die Kontrolle über ausgeführte Apps:

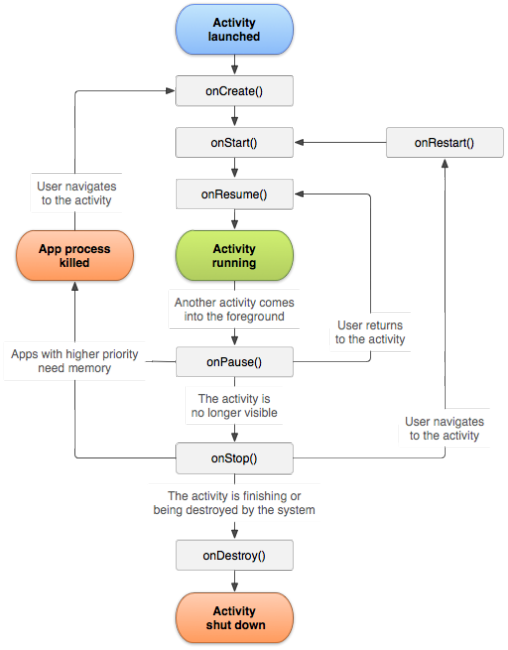
- Verwaltung des Lebenszyklus
- Kommunikation zwischen Komponenten
- Terminierung bei Bedarf (z.B. Speicherknappheit)

2.1 Activities

Beim App-Start wird die Main Activity von Android erzeugt und ausgeführt. Activities besitzen eine grafische Obefläche und verarbeiten Benutzereingaben.

```
public class MainActivity extends Activity {
   @Override
   protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_main);
   }
}
```

Activity Lebenszyklus & Zustände:
Android ruft beim Zustandwechsel Callback-Methoden auf der Activity auf. Diese Methoden können überschrieben werden.



2.1.1 Anwendungsfälle

- Erzeugung des GUI: onCreate()
- Datensicherun: onPause für schnelle Operationen, ansonsten onStop()
- Dienste wie Lokalisierung aktivieren/deaktivieren: onResume() und onPause()
- Zustand des GUI erhalten, z.B. bei Rotation: onSaveInstanceState() und onRestoreInstanceState()

2.2 Intents

- Die Kommunikation zwischen Komponenten erfolgt über Intents (Absicht, Vorhaben)
- Zwei Arten von Intents:
  - Explizit: Aufruf einer definierten Komponente (typischerweise für Komponenten der eigenen App)
  - Implizit: Aufruf einer passenden Komponente (typischerweise für Komponenten aus anderen Apps)
- Apps können sich im Android Manifest mit Intent Filters auf implizite Intents registrieren
- Intents werden stets von Android verarbeitet

```
// 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);
```

2.2.1 Beispiel

```
Button button = findViewById(R.id.buttonNavigate);
button.setOnClickListener(v -> {
   //Explizit
   Intent secondActivityIntent = new Intent(this,
       SecondActivity.class);
   startActivity(secondActivityIntent);
   //Implizit
   Intent intent = new Intent(Intent.ACTION_VIEW, Uri.
       parse("http://www.ost.ch"));
   startActivity(intent);
});
```

2.3 Intents mit Parametern

Zusätzliche Parameter können als Key-Value Paar (Bundle) mit putExtra()/putExtras() übergeben werden.

```
// MainActivity.java
Intent intent = new Intent(this, SecondActivity.class);
intent.putExtra("myKey", 42);
startActivity(intent);
// SecondActivity.java
Intent intent = this.getIntent();
String parameter = intent.getStringExtra("key");
```

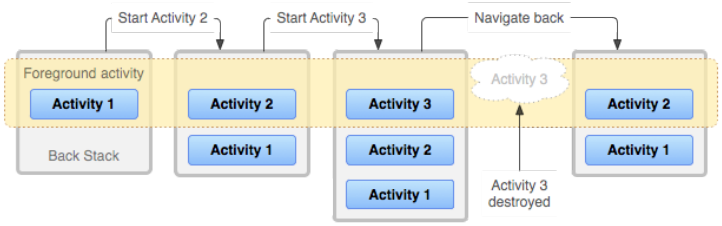
2.3.1 Hinweise

Mit Intents startet man andere Activities.
→ Ohne Rückgabewert: startActivity(Intent)
→ Mit Rückgabewert: startActivityForResult(Intent, int)
Implizite Intents müssen nicht immer einen Empfänger haben.
→ Darum immer überprüfen ob Intent einen Empfänger hat:

```
//MainActivity.java
if(intent.resolveActivity(getPackageManager()) != null){
   startActivity(intent);
}
//AndroidManifest.xml
<uses-permission android:name="android.permission.
    QUERY_ALL_PACKAGES" />
```

2.4 Back Stack (Task)

- Activities werden im Back-Stack verwaltet
- Activities eines Stacks können zu verschiedenen Apps gehören
- Dieselbe Activity kann mehrfach im selben Stack enthalten sein



Ein Back Stack wird auch Task genannt. Android verwaltet die Ausführung von Tasks. Bei Bedarf können Activities in neuen Tasks gestartet werden.

2.5 Tasks, Prozesse und Threads

- Alle Teile eines Apps werden in einer APK-Datei ausgeliefert
- Jedes APK wird mit einem eigenen Linux User installiert (Sandbox)
- Jedes APK wird in einem eigenen Linux Prozess ausgeführt
- Jeder Prozess hat mindestens einen Thread (Main Thread)

2.5.1 Main-Thread

- Automatisch erzeugt beim Start einer Applikation
- Blockierung des Main Threads führt zum ANR-Screen (Application Not Responding)
- Langlaufende Operationen immer in eigenen Threads ausführen (Runnable)
- **Achtung:** Nur der Main Thread darf das GUI aktualisieren, sonst Exception

2.6 GUI

Das GUI kann auf zwei Arten erstellt werden: Deklarativ (Beschreibung in XML) und Imperativ (Beschreibung im Quellcode).

2.7 Event Handling

Listener reagieren auf GUI-Ereignisse und werden bei GUI-Objekt registriert

```
final TextView textView = this.findViewById(R.id.
    text_example);
Button button = this.findViewById(R.id.button_example);
button.setOnClickListener(new View.OnClickListener() {
   @Override
   public void onClick(View view) {
      textView.setText("Button pressed");
   }
});
// Lambda
button.setOnClickListener(v -> { ... });
// XML
android:onClick="onExampleButtonClicked"
public void onExampleButtonClicked(View view)
```

2.8 Resources

Alle Dateien, die keinen Code enthalten, werden als Resources bezeichnet. colors.xml für Farbwerte, dimens.xml für Dimensionen, strings.xml für Texte, styles.xml für Styles. Veränderliche Werte immer in passenden Files definieren und referenzieren. Der Zugriff erfolgt jeweils über die Resource ID. → Zugriff via **R-Klasse**

2.9 Dimensionen

Android erlaubt die Verwendung folgender Dimensionen:

- dp: Density-independent Pixels
- sp: Scale-independent Pixels
- px: Pixel
- pt: Punkte (1/72 eines physikalischen)
- in: Inch
- mm: Millimeter

Empfehlung: Für Schriften immer in sp, Alles andere in dp

2.10 Qualifier

Resources können in unterschiedlichen Varianten hinterlegt werden:

- Texte für verschiedenen Sprachen
- Bilder für verschiedenen Auflösungen
- Layouts für unterschiedliche Gerätetypen

2.10.1 Mehrsprachigkeit

Kein Hardcoded Text sondern über String resource file. Mehrere values Ordner (values_en, etc.) mit strings.xml Dateien anlegen.

2.11 App Manifest

Das AndroidManifest.xml enthält essenzielle Informationen zur App.

- ID, Name, Version und Logo
- Enthaltene Komponenten
- Hard- und Softwareanforderungen
- Benötigte Berechtigungen

2.11.1 Application ID und Version

package: Eindeutige Identifikation der App, Definiert Namespace, Reversed Internet Domain Format (ch.ost.rj.helloworld)
versionName: Ein menschenlesbarer String, Typischerweise Semantic Versioning
versionCode: Ein positiver Integer für interne Verwendung, Je höher die Zahl, desto "neuer"die App, Unterschiedliche Ansätze zur Inkrementierung

2.11.2 Application-Element

- Parent der Komponenten ist der Application-Knoten
- Application ist auch eine Klasse, die den globalen Zustand der App hält
- Eigene Ableitung von Application kann registriert werden
- Application enthält LifeCycle-Methoden, die überschrieben werden können

2.11.3 API Level

- minSdkVersion gibt an, welche Version das Gerät mindestens haben muss
- maxSdkVersion gibt an, welche Version das Gerät maximal haben darf
- targetSdkVersion ist die Version, welche die App bei der Ausführung verwendet
- compileSdkVersion gibt an, mit welcher API das App kompiliert wird

3 GUI Programmierung

3.1 View und ViewGroup

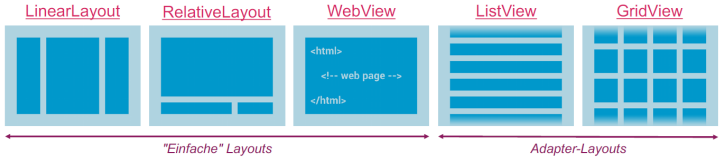
View ist die Basisklasse aller GUI Elemente. Es belegt einen rechteckigen Bereich und kümmert sich um die Darstellung und Event Verarbeitung. Die Ableitung ViewGroup enthält View-Objekte (Parent-Child Beziehung). ViewGroup-Klassen ordnen ihre Kinder nach einem Muster an, sind strukturierend und unsichterbar. Werden auch Layouts oder Container genannt.

3.2 Layouts Allgemein

Im onCreate der Activity wird das layout geladen:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
   super.onCreate(savedInstanceState);
   setContentView(R.layout.activity_main);
}
```

Layouts Übersicht:



Es gibt noch weitere Layouts und eigene können auch definiert werden. Layouts können beliebig verschachtelt werden, jedoch mit negativem Einfluss auf die Performance. → Am besten Flache, breite Hierarchie

3.2.1 Layout-Parameter

Verschachtelte Child View teilt Parent mit, wie sie angeordnet werden wollen. Child setzt auf sich selber diese Parameter.

```
<LinearLayout
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   android:orientation="vertical"
   android:gravity="center">
</LinearLayout>
```

Mögliche Werte: match_parent (so gross wie möglich), wrap_content (so klein wie Content) und Zahl (unüblich, meist in dp).

3.2.2 padding und Margin

Padding wird auf sich selbst gesetzt. Margin wird dem Parent übergeben, da ein Child nicht einfach den Platz dem Parent wegnehmen kann.

```
android:padding="20dp"
android:layout_margin="20dp"
```

3.3 Linear Layout

Vertikal oder horizontal angeordnet. Mit layout_weight kann die Grösse beeinflusst werden. → Verwendung in Kombination mit wrap_content

```
android:orientation="vertical"
android:orientation="horizontal"

// Beispiel mit weight:
| | |       | // Links: minimaler Platz (kein weight)
| | |       | // Mitte: android:layout_weight="1"
| | |       | // Rechts: android:layout_weight="3"
```

3.4 Frame Layout

Kinder werden übereinander angeordnet. z.B. Live-Kamerabild mit Auslöse-Button und Hilfslinien. → Anpassung der "Höhe"über dem Bild: Standardmässig gilt die Reihenfolge im XML. Manuelle Anpassung mit android:translationZ möglich

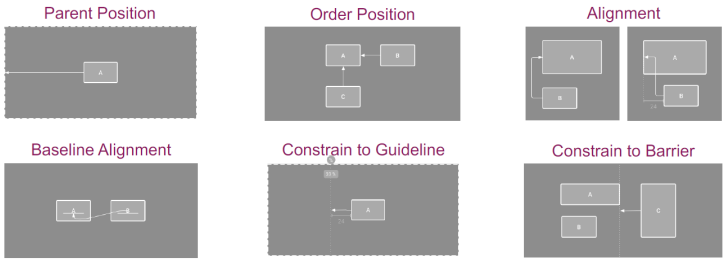
3.5 Relative Layout

Kinder werden relativ zueinander angeordnet. Identifizierung der anderen Kinder über Resource IDs. Mächtig, kann als effizienter Ersatz für verschachtelte Linear Layouts dienen.

```
// Beispiele
android:layout_alignParentTop="true"
android:layout_toStartOf="@id/..."
android:layout_alignStart="@id/..."
```

3.6 Constraint Layout

Das modernste und flexibelste Layout. Ist Teil von Jetpack/AndroidX. Grundidee: Definieren von Beziehungen zwischen Views. Pro View muss mindestens eine horizontale und vertikale Einschränkung definiert werden.



3.7 Widgets

Namespace: android.widget
Basisklasse: View

3.7.1 TextView und ImageView

TextView zur Anzeige von Text:

```
<TextView
   ...
   android:text="TextView"
   android:textSize="20sp"
   android:textStyle="bold"
   android:typeface="monospace"
   android:textColor="@android:color/white"
   android:background="@color/colorPrimaryDark"
   android:drawableEnd="@drawable/ic_emoji"
   android:drawableTint="@android:color/white" />
```

ImageView zur Anzeige von Bildern:

```
<ImageView
   ...
   android:layout_height="80dp"
   android:src="@drawable/ic_emoji"
   android:scaleType="fitCenter"
   android:tint="@color/colorPrimaryDark" />
```

3.7.2 Button und ImageButton

Buttons. Lösen via Listener Aktionen aus. Ableitung von TextView bzw. ImageView.

```
<Button
   ...
   android:text="Button"
   android:drawableEnd="@drawable/ic_emoji"
   android:drawableTint="@color/colorPrimary"/>
```

3.7.3 EditText

EditText dient als Eingabefeld für Texte und Zahlen. android:inputType beeinflusst Verhalten und aussehen (auch Keyboard).

```
android:inputType="textPassword"
android:inputType="date"
android:inputType="textMultiLine"
// Auch kombinierbar
android:inputType="textCapCharacters|textAutoCorrect"
```

Bei der Texteingabe kann auf Ereignisse reagiert werden. Dazu muss ein TextWatcher als Listener registriert werden. Folgende 3 Methoden können überschrieben werden:

- beforeTextChanged
- onTextChanged
- afterTextChanged

```
myEditText.addTextChangedListener(new TextWatcher() {
   public void afterTextChanged(Editable editable) {
      if (editable.length() < 8) {
         passwordInput.setError("Passwort zu kurz.");
      }
   }
})
```

Weitere, häufig verwendete Widgets:
Checkbox, Picker, Floating Action Button, Radio Buttons, Seek Bar, Spinner

3.7.4 UI-Elemente ohne XML

Werden direkt aus dem Code heraus erzeugt. Anpassbarkeit ist oft eingeschränkt (Farben, Texte, etc.)
Toasts: Einfache Rückmeldung zu Vorgang (Pop Up)
Snackbars: Wie Toast, aber mit Interaktion.
Dialoge: Erzwingen Aktion von Benutzer
Notification: Mitteilung ausserhalb aktiver Nutzung. NotificationCompat in AndroidX verwenden.
Manus: Existieren in verschiedenen Varianten. Options Menu, Contextual Menu, Popup Menu. → Wird generell als Resource in res/menu definiert.

3.8 ScrollView

- Ist ein spezielles Layout mit nur einem Kind-Element
- Erlaubt das vertikale Scrolling des Inhalts
- Horizontal nur mit HorizontalScrollView
- Alternative in AndroidX: NestedScrollView (erlaubt beide Richtungen)

```
<ScrollView
   android:layout_width="match_parent"
   android:layout_height="match_parent">
   <!-- Genau ein Kind hier -->
</ScrollView>
```

3.9 ListView und ArrayAdapter

Gut für Darstellung von Collections. Ein Adapter vermittelt zwischen der Darstellung und der Datenquelle.

```
// main_activity.xml
<?xml version="1.0" encoding="utf-8"?>
<ListView xmlns:android=" ... "
   android:id="@+id/list_example"
   android:layout_width="match_parent"
   android:layout_height="match_parent">
</ListView>

// MainActivity.java
setContentView(R.layout.activity_main);
String[] data = new String[] { ... };
ArrayAdapter<String> adapter = new ArrayAdapter<>(
   this,
   android.R.layout.simple_list_item_1,
   android.R.id.text1,
   data);
ListView listView = findViewById(R.id.list_example);
listView.setAdapter(adapter);
```

3.10 RecyclerView

Die RecyclerView ist eine moderne Alternative zu ListView und GridView. Ist Teil von AndroidX und erzwingt die Verwendung von View Holdern.

```
// main_activity.xml
<?xml version="1.0" encoding="utf-8"?>
<androidx.recyclerview.widget.RecyclerView
   android:id="@+id/recycler_view"
   android:layout_width="match_parent"
   android:layout_height="match_parent">
</androidx.recyclerview.widget.RecyclerView>

// MainActivity.java
setContentView(R.layout.activity_recyclerview);
RecyclerView recyclerView = findViewById(R.id.
    recycler_view);
RecyclerView.LayoutManager layoutManager;
layoutManager = new LinearLayoutManager(this);
recyclerView.setLayoutManager(layoutManager);
ArrayList<User> data = UserManager.getUsers();
UsersAdapter adapter = new UsersAdapter(data);
recyclerView.setAdapter(adapter);

// UsersAdapter.java
public class UsersAdapter extends RecyclerView.Adapter<
    ViewHolder> {
   private ArrayList<User> users;

   @Override
   public ViewHolder onCreateViewHolder(ViewGroup parent
       , int vt) {
      Context context = parent.getContext();
      LayoutInflater inflater = LayoutInflater.from(
          context);

      View view = inflater.inflate(
         android.R.layout.simple_list_item_2,
         parent,
         false);

      return new ViewHolder(
         view,
         view.findViewById(android.R.id.text1),
         view.findViewById(android.R.id.text2));
   }
   @Override
   public void onBindViewHolder(ViewHolder holder, int
       position) {
      User user = this.users.get(position);
      holder.text1.setText(user.name);
      holder.text2.setText(user.age + " Jahre");
   }
   @Override
   public int getItemCount() {
      return this.users.size();
   }
}
```

4 Strukturierung, Material Design und Styling

4.1 Fragments

Activities können nicht kombiniert werden, Fragments aber schon. Ein Fragment ist ein modularer Teil in einer Activity mit eigenem Lebenszyklus.
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

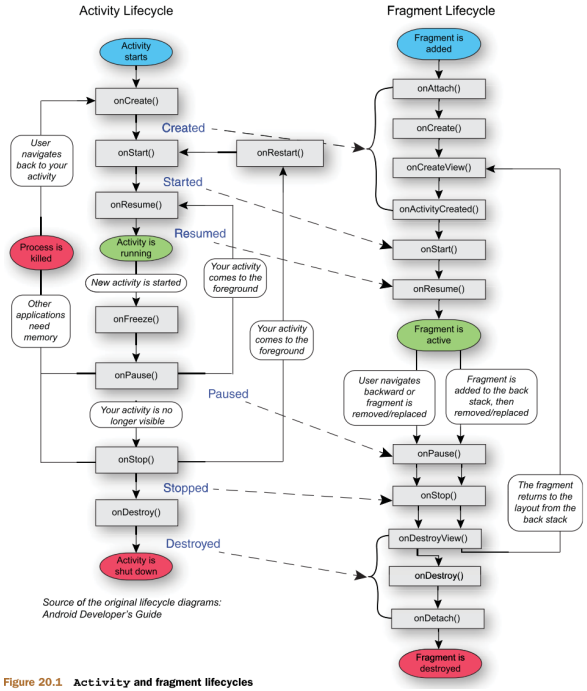


Figure 20.1 **Activity and fragment lifecycles**

4.1.1 Dynamische Einbindung

```
// activity_main.xml
<LinearLayout xmlns:android="( ... )"
   android:layout_width="match_parent"
   android:layout_height="match_parent">

   <FrameLayout android:id="@+id/main_fragment_container
       "
      android:layout_width="match_parent"
      android:layout_height="match_parent" />
</LinearLayout>

//MainActivity.java
public class MainActivity extends AppCompatActivity {
   @Override
   protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_main);

      FragmentManager mgr = getSupportFragmentManager();
      FragmentTransaction trans =mgr.beginTransaction();

      OutputFragment fragment = new OutputFragment();
      trans.add(R.id.main_fragment_container, fragment);
      trans.commit();
   }
}
```

4.1.2 Activity-Fragment Kommunikation

Fragments sollen wiederverwendbar sein. Einbindung in verschiedene Activities und keine direkten Abhängigkeiten zu Activities haben.
Best Practices:
Activity → Fragment: Parameter und Methoden
Activity ← Fragment: Callback-Interfaces

4.1.3 parameter und Methoden

```
// Vorher:
OutputFragment fragment = new OutputFragment();

// Mit Parameter:
OutputFragment fragment;
fragment = OutputFragment.create("Initial Value");
// In fragment.java: public static OutputFragment create
    (String text) { ... }

// Zusätzlich: Event-Listener
Button button = findViewById(R.id.main_button);
button.setOnClickListener(v -> {
   fragment.updateText("Updated value");
});
```

4.1.4 Callback-Interfaces

```
// Callback.java
public interface OutputFragmentCallback {
   void onTextTapped(String text);
}

// MainActivity.java
public class MainActivity extends AppCompatActivity
    implements OutputFragmentCallback {
   @Override
   public void onTextTapped(String text) {
      // Callback behandeln
   }
}

// OutputFragment.java
public class OutputFragment extends Fragment {
   private OutputFragmentCallback callback;
   @Override
   public void onAttach(Context context) {
      super.onAttach(context);
      try {
         callback = (OutputFragmentCallback) context;
      } catch (ClassCastException e) {
         throw new ClassCastException(" ...");
      }
   }
   @Override
   public View onCreateView( ... ) {
      View fragment = inflater.inflate( ... );
      textOutput = fragment.findViewById(R.id.
          output_text);
      textOutput.setOnClickListener(v -> {
         callback.onTextTapped("");
      });
      return fragment;
   }
}
```

4.1.5 Fragmente austauschen

Fragmente sind austauschbar und Übergänge können animiert werden. (XML-Beschreibung der Animation (res/anim))

```
fragmentManager.beginTransaction()
   .setCustomAnimations(
      R.anim.slide_in, // Einblendung neues Fragment
      R.anim.fade_out, // Ausblendung altes Fragment
      R.anim.fade_in, // Einblendung altes Fragment(Pop)
      R.anim.slide_out)//Ausblendung neues Fragment(Pop)
   .replace(R.id.main_fragment_container, newFragment)
   .addToBackStack(null)
   .commit();
});
```

4.1.6 Fragmente verschachteln

Sind verschachtelbar, gleiches Vorgehen bei der Einbindung. Unterschied: getChildFragmentManager() anstelle von getSupportFragmentManager()

4.2 Material Deisgn

Eine Designlanguage ist eine Hilfestellung für den Designprozess. Klare Regeln oder Empfehlungen zu Farbschema, Icons, Schriften, Abständen, etc. Martial Design ist die Design Language von Google.

4.2.1 Grundprinzipien

Material is the metaphor:

- Inspiriert von der physischen Welt
- Oberflächen erinnern an Papier und Tinte
- Materialien reflektieren Licht & werfen Schatten

Bold, graphic, intentional:

- Basiert auf Prinzipien von Print-Medien
- Hierarchie, Raster, Schriften, Farben, etc.

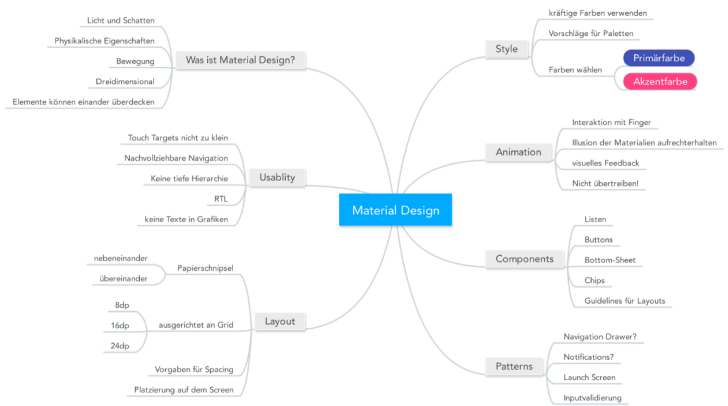
Motion provides meaning:

- Bewegung bedeutet Aktion
- Zurückhaltende, subtile Verwendung

4.2.2 Vorgaben

- Material ist immer 1dp dick ("Papier")
- Material wirft Schatten
- Material hat eine unendliche Auflösung
- Inhalt hat keine Dicke und ist Teil des Materials
- Material kann sich verändern
- Material kann sich bewegen

4.2.3 Zusammenfassung



4.3 Styling

Widgets werden über XML Attribute gestyled. Mögliche Probleme bei umfangreichen Apps wie Code-Duplizierung, Inkonsistenzen, Unübersichtlichkeit. Styles können wiederverwendbar gemacht werden.

4.3.1 Styles

Styles werden in der styles.xml resource definiert. Styles können aber auch mit der .Notation geerbt werden:

```
// layout.xml
<TextView
   android:layout_width="match_parent"
   android:layout_height="wrap_content"
   android:text="Element 1"
   style="@style/HeaderText" />
<TextView
   android:layout_width="match_parent"
   android:layout_height="wrap_content"
   android:text="Element 2"
   style="@style/HeaderText.Big" />

// styles.xml
<style name="HeaderText">
   <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>
```

4.3.2 Themes

Themes sind spezielle Styles, die für eine ganze App oder einzelne Activities gelten. Definition wie normale Styles in Resources:

```
// styles.xml
<resources>
   <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>

// Anwenden: Manifest oder Activity:
// AndroidManifest.xml
<application ... android:theme="@style/AppTheme">
   <activity ... android:theme="@style/AnotherAppTheme"
       />
</application>
// MainActivity.java
@Override
protected void onCreate(Bundle savedInstanceState) {
   setTheme(R.style.AnotherAppTheme);
   setContentView(R.layout.activity_styling);
}
```

4.3.3 Material Components Library

Damit erhält man, ausser den Themes, auch Zugriff auf Material Design-Widgets.

5 Berechtigungen, Persistenz und Hardwarezugriff

5.1 Berechtigungen

5.1.1 Grundlagen

Apps dürfen nur Aktionen ausführen, die andere Dienste nicht negativ beeinflussen (Sandbox). Vor riskanten Operationen müssen Berechtigungen eingeholt werden:

- Zugriff auf System APIs (Internet, WiFi, Bluetooth, etc.)
- Zugriff auf sensitive Daten (Telefonanruf, Kontaktliste, Kalender, etc.)
- Zugriff auf bestimmte Hardware (Kamera, Lokalisierung, etc.)

Es gibt zwei Arten von Berechtigungen:
Normal: Werden durch das System erteilt.
Gefährlich: Werden durch den Benutzer erteilt.

Berechtigungen erteilen:
Bis API 22 (Android 5.1.1): Während Installation der App. Kein selektives Ablehnen möglich.
Ab API 23 (Android 6.0): Während Nutzung der App. Selektives Ablehnen möglich.
Ab API 30 (Android 11.0): Weiterhin während Nutzung. Dialog ergänzt um ËinmaligOption

Berechtigungen verwalten:
Berechtigungen können vom Benutzer zurückgezogen werden. Seit API 30 können Berechtigungen vom System zurückgesetzt werden. **Darum Empfehlung:** Kein Flag zum Abfragen der Berechtigung setzten in der App, sondern immer Zugriff überprüfen. Sonst wird Berechtigung vom System entzogen aber Flag im Code ist gesetzt. SecurityException: Wenn keine Berechtigung, gut zum Debuggen wenn App abstürzt.

Best Practices:

- Nur anfordern, was auch benötigt wird
- Im Kontext der Verwendung anfordern
- Transparente Erklärungen
- Abbruch ermöglichen
- Verweigerung berücksichtigen

5.1.2 Berechtigungen XML/Code

Manifest: Benötigte Berechtigungen müssen im Manifest deklariert werden. Knoten <uses-permission>
Hinweise auf benötigte Features für Filterung im Google Play Store. Knoten <uses-feature> → Ist optional, aber empfohlen

```
// AndroidManifest.xml
<?xml version="1.0" encoding="utf-8"?>
<manifest>
<!-- Berechtigungen -->
<uses-permission
   android:name="android.permission.CALL_PHONE"/>
<uses-permission
   android:name="android.permission.CAMERA"/>
<uses-permission
   android:name="android.permission.
       WRITE_EXTERNAL_STORAGE"
   android:maxSdkVersion="28" />
<!-- Feature-Hinweise -->
<uses-feature
   android:name="android.hardware.location" />
<uses-feature
   android:name="android.hardware.camera"
   android:required="true" />
<uses-feature
   android:name="android.hardware.bluetooth"
   android:required="false" />
</manifest>
```

Anforderung im Code:

```
// MainActivity.java
private static final int CALLBACK_CODE = 1;
String permission = Manifest.permission.CALL_PHONE;
// Aktuellen Status prüfen (AndroidX)
int status = ContextCompat.checkSelfPermission(
   this,
   permission);
if (status != PackageManager.PERMISSION_GRANTED) {
   if (shouldShowRequestPermissionRationale(permission))
        {
      // Erklärung für Benutzer anzeigen (Wozu nötig?)
   }
   // Berechtigung beim Benutzer anfordern
   requestPermissions(
      new String[]{ permission },
      CALLBACK_CODE);
}

@Override
public void onRequestPermissionsResult(
   int requestCode, String[] permissions, int[] results)
        {
   if (requestCode != CALLBACK_CODE)
      return;
   if (results.length == 0)
      return; // Anfrage abgebrochen
   if (results[0] == PackageManager.PERMISSION_GRANTED)
       {
      // Berechtigung erteilt
   } else {
      // Berechtigung verweigert
   }
}
```

Hinweis:
falls der benutzer "Nicht mehr fragen"auswählt, werden keine Dialoge mehr angezeigt. Ab API 30 gelten wiederholte Ablehnungen automatisch als "Nicht mehr fragen".

5.2 Persistenz

5.2.1 Speicherarten

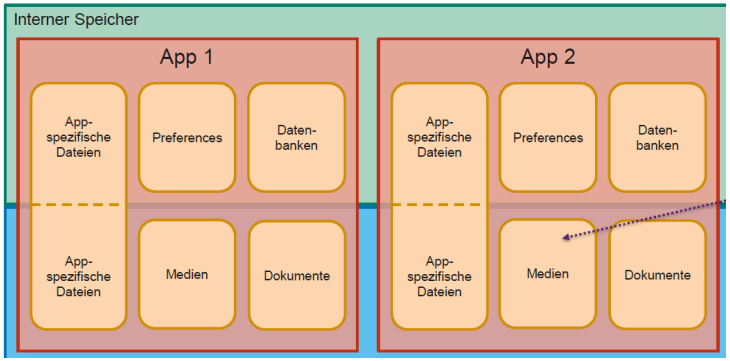
Interner Speicher:

- Stets verfügbar
- Geschützter Speicherbereich pro App
- Speicherplatz begrenzt
- Für app-interne Daten

Externer Speicher:

- Nicht immer verfügbar
- Oft ein Wechseldatenträger
- Emulation durch Android möglich
- Speicherplatz begrenzt (aber meist grösser)
- Primär für geteilte Daten

Bis API 28 konnte eine App auf beliebige Medien im externen Speicher zugreifen, brauchte dazu aber Berechtigungen.
Seit API 29 kann eine App eigene Medien im externen Speicher ohne Berechtigungen lesen und schreiben. Der Zugriff auf Medien von fremden Apps ist nur noch lesend möglich und braucht eine Berechtigung.



5.2.2 App-spezifische Dateien

- Eigene, proprietäre Datenformate
- Interner oder externer Speicher
- Bei Deinstallation der App gelöscht
- Geschützt vor fremdem Zugriff
- Keine Berechtigungen nötig (seit API 19)
- Zugriff via File-API und Context
  - Context.getFilesDir()
  - Context.getCacheDir()
  - Context.getExternalFilesDir()
  - Context.getExternalCacheDir()

```
// Schreiben
File folder = getFilesDir();
File file = new File(folder, "my_file.txt");
String input = "MGE Beispiel";
FileOutputStream outputStream = new FileOutputStream(
    file);
outputStream.write(input.getBytes());
outputStream.close();

// Dateien anzeigen
for(File fileInFolder : folder.listFiles()) {
   Log.d("MGE.V05", "File: " + fileInFolder.getName());
}

// Lesen
int length = (int) file.length();
byte[] bytes = new byte[length];
FileInputStream inputStream = new FileInputStream(file);
inputStream.read(bytes);
inputStream.close();
String output = new String(bytes);
```

5.2.3 Preferences

- Key-Value-Paare
- Interner Speicher
- Bei Deinstallation der App gelöscht
- Keine Berechtigungen nötig
- Zugriff via SharedPreferences-Objekte
  - Context.getSharedPreferences(name, mode)
  - Activity.getPreferences(mode)
- AndroidX: EncryptedSharedPreferences

```java
String file = "ch.ost.rj.mge.v05.myapp.preferences";
String key1 = "my.key.1";
String key2 = "my.key.2";
String key3 = "my.key.3";
int mode = Context.MODE_PRIVATE;

// Objekt abholen
SharedPreferences preferences;
preferences = getSharedPreferences(file, mode);

// Schreiben
SharedPreferences.Editor editor = preferences.edit();
editor.putString(key1, "MGE Beispiel");
editor.putBoolean(key2, true);
editor.putInt(key3, 42);
editor.commit();

// Lesen
String value1 = preferences.getString(key1, "default");
boolean value2 = preferences.getBoolean(key2, false);
int value3 = preferences.getBoolean(key3, 0);
```

5.2.4 Content Providers

- Datenquelle für andere Apps
- Client-Server Modell
  - Client: Content Resolver
  - Server: Content Provider
- SQL-ähnliche, standardisierte Schnittstelle
- Android enthält diverse Provider: Kalender, Kontakte, Medien, Dokumente, Wörterbuch

5.2.5 Content Resolvers

- Zugriff auf Provider via ContentResolver
- Bietet Methoden für CRUD-Operationen
- Iteration über Ergebnisse via Cursor
- Berechtigung nötig für Zugriff auf Daten

```java
Cursor cursor = getContentResolver().query(
    uri,
    projection,
    selection,
    selectionArgs,
    sortOrder
);
```

5.2.6 Medien

- Teilbare Bilder, Videos und Musik
- Externer Speicher
- Bleiben bei Deinstallation der App erhalten
- Berechtigungen
  - bis API 28: für jegliches Lesen oder Schreiben
  - ab API 29: nur für das Lesen fremder Dateien

Zugriff via MediaStore (Content Provider)

```java
// Auslesen von Bildern sortiert nach Einfügedatum
String[] projection = new String[] {
    MediaStore.Images.Media.TITLE,
    MediaStore.Images.Media.DATE_ADDED
};
String order = MediaStore.Images.Media.DATE_ADDED + "
    DESC";
Cursor cursor = getContentResolver().query(
    MediaStore.Images.Media.EXTERNAL_CONTENT_URI,
    projection, // projection
    null, // selection
    null, // selectionArgs
    order // sortOrder
);
int ct = cursor.getColumnIndex(MediaStore.Images.Media.
    TITLE);
int cd = cursor.getColumnIndex(MediaStore.Images.Media.
    DATE_ADDED);
while (cursor.moveToNext()) {
    String title = cursor.getString(ct);
    long added = cursor.getLong(cd);
    // ... hier die Werte verwenden ...
}
cursor.close();
```

5.2.7 Dokumente

- Teilbare Dokumente wie PDF, ZIP, etc.
- Externer Speicher
- Bleiben bei Deinstallation der App erhalten
- Keine Berechtigungen nötig
- Zugriff via Storage Access Framework (Kombination aus Intents & Content Provider)
- Auswahl von Dokumenten via Picker"

5.2.8 Datenbanken

- Strukturierte Daten
- Interner Speicher
- Bei Deinstallation der App gelöscht
- Keine Berechtigungen nötig
- Zugriff auf zwei Arten: SQLite API, Room aus AndroidX/Jetpack (Wrapper um SQLite)

5.2.9 Datenbanken – Room

- Abstraktionsschicht über SQLite
- ORM (Object Relational Mapping)
  - Abbildung von DB-Tabellen auf Java-Klassen
  - Basiert auf Java-Annotations
- Vorteile
  - Weniger Code nötig (z.B. Mapping)
  - Überprüfung von Queries zur Compile-Zeit
  - Kompatibilität mit Jetpack-Komponenten

```java
// Entry.java
@Entity
public class Entry {
    @PrimaryKey(autoGenerate = true)
    public int id;

    @ColumnInfo
    public String content;
}

// EntryDao.java
@Dao
public interface EntryDao {
    @Query("SELECT * FROM entry")
    List<Entry> getEntries();

    @Insert
    void insert(Entry entry);
    @Delete
    void delete(Entry entry);
}

// EntryDatabase.java
@Database(entities = {Entry.class}, version = 1)
public abstract class EntryDatabase extends RoomDatabase
    {
    public abstract EntryDao entryDao();
}

// MainActivity.java
// Erzeugung DB-Objekt
EntryDatabase db = Room.databaseBuilder(this,
    EntryDatabase.class, "room.db")
    .build();
EntryDao dao = db.entryDao();

// Daten einfügen
Entry entry = new Entry();
entry.content = "MGE Vorlesung";
dao.insert(entry);

// Daten auslesen
List<Entry> entries = dao.getEntries();
for (Entry entry : entries) {
    Log.d(null, + entry.id + " | " + entry.content);
}

// Aufräumen
db.close();
```

5.3 Hardwarezugriff

5.3.1 Grundlagen

Smartphones besitzen diverse Aktoren & Sensoren. z.B: Display, Lautsprecher, Vibration, Kamera, Beschleunigung, Licht, Lage.

Sensor Framework:

- SensorManager als Einstiegspunkt
- Sensor als Repräsentant für realen Sensor
- SensorEvent enthält Werte des Sensors
- SensorEventListenerfür Callbacks

- Verzögerung beeinflusst Energieverbrauch
  - SENSOR_DELAY_FASTEST (0ms)
  - SENSOR_DELAY_GAME (20ms)
  - SENSOR_DELAY_UI (60ms)
  - SENSOR_DELAY_NORMAL (200ms)
- Änderung der Genauigkeit
  - SENSOR_STATUS_ACCURACY_HIGH
  - SENSOR_STATUS_ACCURACY_MEDIUM
  - SENSOR_STATUS_ACCURACY_LOW
  - SENSOR_STATUS_ACCURACY_UNRELIABLE
- Inhalte von values abhängig von Sensor

```java
// Bei Sensor für Änderungen registrieren

String service = Context.SENSOR_SERVICE;
int type = Sensor.TYPE_LIGHT;
int delay = SensorManager.SENSOR_DELAY_NORMAL;

SensorManager mgr = (SensorManager)getSystemService(service);
Sensor sensor = mgr.getDefaultSensor(type);
mgr.registerListener(this, sensor, delay);



// Implementierung von SensorEventListener

@Override
public void onSensorChanged(SensorEvent sensorEvent) {
    float lux = sensorEvent.values[0];
    Log.d(null, lux + " lux");
}

@Override
public void onAccuracyChanged(Sensor sensor, int i) {
}
```

MainActivity.java

5.3.2 Vibration

Verwendung der Klasse Vibrator. Berechtigung nötig (VIBRATE)
Ab API26: createOneShot() (erlaubt Stärke der Vibration) und createWaveform() (für komplexe Muster).
Ab API29: createPredefined() für Standard-Effekte.

```
String service = Context.VIBRATOR_SERVICE;
int maxAmplitude = 255;
Vibrator vibrator = (Vibrator) getSystemService(service)
    ;
// Ab API 1
vibrator.vibrate(500);
Vibrator.cancel();
// 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);
```

5.3.3 Connectivity Internet

Über Klasse ConnectivityManager. Üblich sind zwei Kanäle (Wifi & Mobile). Android nutzt automatisch besten Kanal (Höhere Geschwindigkeit, Bessere Signalqualität, Kein Roaming). Berechtigung nötig: ACCESS_NETWORK_STATE.

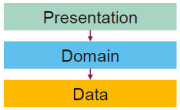
```
String service = Context.CONNECTIVITY_SERVICE;
ConnectivityManager manager;
manager = (ConnectivityManager) getSystemService(service
    );

// Aktive Verbindung prüfen
NetworkInfo activeNetwork = manager.getActiveNetworkInfo
    ();
if (activeNetwork != null) {
   int type = activeNetwork.getType();
   Log.d(null, "Active connection: " + type);
}
// Verbindungen prüfen
for (Network network : manager.getAllNetworks()) {
   NetworkInfo info = manager.getNetworkInfo(network);
   boolean state = info.isConnected();
   if (info.getType() == ConnectivityManager.TYPE_WIFI)
       {
      Log.d(null, "WiFi is connected: " + state);
   }
   if (info.getType() == ConnectivityManager.TYPE_MOBILE
       ) {
      Log.d(null, "Mobile is connected: " + state);
   }
}
```

6 Architektur und fortgeschrittene Themen

6.1 Schichtenarchitektur

Wichtig ist, dass keine zyklischen Abhängigkeiten entstehen. Domain sollte zum Beispiel nie von View abhängen.
Beispiele für Möglichkeiten zur Entkopplung an Grenzen:
Observer Pattern, Interfaces und Dependency Injection

- **Presentation**
  - Darstellung und Interaktion mit Benutzer
  - Typischerweise stark an UI-Toolkit gebunden
  - Zugriff auf Domain-Schicht
- **Data**
  - Speicherung der Daten
  - Stellt Daten der Domain zur Verfügung
  - Auch Persistenz oder Datenhaltung genannt
- **Domain**
  - Businesslogik und Domänenklassen
  - Keine UI-Funktionalität
  - Wenig externe Abhängigkeiten
  - Einfach zu testen

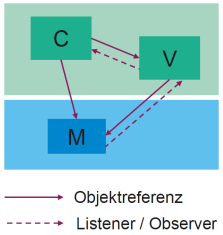


6.2 Observer Pattern

Zwei Rollen:
Sobject: Wird beobachtet, z.B. Model-Klasse
Observer: Beobachter, z.B. View-Klasse
→ Observer registrieren sich beim Subject für Statusänderungen.
→ Observer Pattern wird oft mit MVC kombiniert

6.3 Model-View-Controller

Pattern für die Organisation der Presentation:
Model beinhaltet die Daten. (Java-Klassen)
View liest die Daten des Modells und zeigt diese an. (View, Adapter)
Controller erhält Events der View und manipuliert das Model. (Activity, Fragment)
Kritik: Viel Code im Controller.



6.4 Application Methoden

onCreate(): Einmalig nach dem Start der App. Vor Erzeugung anderer App-Komponenten
onTerminate(): Wird auf realen Geräten nie aufgerufen. Reminder: Apps können stets beendet werden
onConfigurationChanged(newConfig): Bei Änderungen der System-Konfiguration. Parameter enthält neue Konfiguration. Beispiel: Rotation des Gerätes
onLowMemory(): Bei Speicherknappheit des Systems. Hinweis auf mögliche Terminierung der App
onTrimMemory(level): Bei geeigneten Momenten für Aufräumaktionen. Parameter gibt Hinweise auf Auslöser. Beispiel: App geht in den Hintergrund (TRIM_MEMORY_UI_HIDDEN)
→ Via Application-Klasse kann der Lebenszyklus aller Activities überwacht werden. Implementierung und Registrierung von Application.ActivityLifeCycleCallbacks

6.5 Context

Context ist eine abstrakte Klasse der SDK mit über 50 Ableitungen, beispielsweise Activity und Application. Sie erlaubt den Zugriff auf Dienste und Ressourcen der App. z.B:

- Context.startActivity()
- new Intent(Context, Type)
- LayoutInflater.from(Context)
- Toast.makeText(Context, String, int)
- NotificationManagerCompat.from(Context)

6.6 Broadcasts

- Für den Austausch von meldungen zwischen Apps. (Publish-Subscribe-Pattern)
- Zwei Arten von Broadcasts: Lokal: innerhalb der App. Global: Innerhalb des ganzen Systems.
- Android sendet selbst globale Broadcasts (System gestartet, Netzwerkverbindung verloren, SMS empfangen, etc.)

6.6.1 Broadcasts empfangen

Dynamische Registrierung via Context: An Lebenszeit des Contexts gebunden. Abmelden von Events nicht vergessen.

```
// MyReceiver.java
public class MyBroadcastReceiver extends
     BroadcastReceiver {
   @Override
   public void onReceive(Context context, Intent intent)
       {
      Log.d(null, "Broadcast: " + intent.getAction());
   }
}

// AndroidManifest.xml
<application>
   <receiver android:name=".MyBroadcastReceiver">
      <intent-filter>
         <action android:name="android.intent.action.
             TIME_SET"/>
      </intent-filter>
   </receiver>
</application>

// MainActivity.java
// Registrierung
BroadcastReceiver receiver = new MyBroadcastReceiver();
String action = ConnectivityManager.CONNECTIVITY_ACTION;
IntentFilter filter = new IntentFilter(action);
registerReceiver(receiver, filter);
// Abmeldung
unregisterReceiver(receiver);
```

6.6.2 Broadcasts versenden

Broadcasts sind normale Intent-Objekte. Die Action im Intent definiert den Ereignistyp. Parameter sind als Extras möglich.

```
// AndroidManifest.xml
<application>
   <receiver android:name=".MyBroadcastReceiver">
      <intent-filter>
         <action android:name="ch.ost.rj.mge.v06.
             MY_INTENT" />
      </intent-filter>
   </receiver>
</application>

// MainActivity.java
// Registrierung
BroadcastReceiver receiver = new MyBroadcastReceiver();
String action = "ch.ost.rj.mge.v06.MY_INTENT";
IntentFilter filter = new IntentFilter(action);
registerReceiver(receiver, filter);

// Impliziter Broadcast
Intent intent = new Intent();
intent.setAction(action);
sendBroadcast(intent);

// Expliziter Broadcast
Intent intent = new Intent(this, MyBroadcastReceiver.
    class);
intent.setAction(action);
sendBroadcast(intent);
```

6.6.3 Best Practices

- Dynamische Registrierung bevorzugen
- Lokale Broadcasts bevorzugen
- Keine sensitiven Daten in Broadcasts übermitteln
- App ID in eigene Broadcast-Actions integrieren
- Schnelle Rückkehr aus onReceive()

6.7 Services

- Für die Ausführung von Aktionen im Hintergrund
  - Laden von Daten über Netzwerk
  - Streaming von Musik
  - Rechenintensive Aufgaben
- Lebenszyklus unabhängig von App
- Kein oder reduziertes UI (Notification)
- Werden auf Main-Thread ausgeführt
  - Service: Aufgabe von Activity entkoppeln
  - Thread: Aufgabe von Main-Thread entkoppeln

6.7.1 Started Services

- Für einmalige Aktionen
- Laufen potentiell endlos weiter
  - Beendigung durch Service selbst: stopSelf()
  - Beendigung durch eine Applikation: stopService()
  - Beendigung durch Android
- Spezialvarianten
  - IntentService für Ausführung in Background-Thread und automatischem Stopp
  - JobIntentService als modernere Alternative für IntentService
  - Foreground Services mit Notifications als UI

6.7.2 Bound Services

- Für Aufgaben über längere Zeitdauer
- Client-Server ähnliche Kommunikation
  - Innerhalb eines Apps: via Interface
  - App-übergreifend: via Messenger-Klasse
- Austausch von Daten fester Bestandteil
- Mehrere Clients gleichzeitig möglich
- Nach Verbindungsende zu letztem Client wird der Service automatisch gestoppt

6.8 Build & Deployment

Apps werden aus APK-Dateien installiert. APKs aus dem Play Store sind signiert.

APK Splitting/Expansion Files:

- Limit des Google Play Store: 100 MB
  - Keine technische Grenze des APK-Formats
  - Schutz der Infrastruktur und User
- Optimierung 1: APK Splitting
  - Aufteilung in verschiedene kleinere APKs
  - Kriterien: CPU-Architektur, Gerätetyp, ...
- Optimierung 2: APK Expansion Files
  - Für speicherintensive Ressourcen (z.B. Videos)
  - Play Store erlaubt max. 2 x 2 GB zusätzlich
  - ZIP-Archive mit Dateiendung .OBB

ABB (Android App Bundle):

- Nachfolger des APK-Formats
- Container mit jeglichen Inhalten eines Apps
- Dynamische Erzeugung von APKs
  - Optimierte Dateigrösse
  - Play Feature Delivery für Feature-Module
  - Play Asset Delivery als Nachfolger von OBB
- Konsequenzen
  - Google in Besitz des Signaturschlüssels
  - Neues Limit von 150 MB für ABB-Datei
- Ab 2021 zwingend im Google Play Store

7 Android Jetpack

- Android Jetpack ist eine Sammlung von Libraries von Google
- Die Libraries vereinfachen die Entwicklung von Android-Apps
- Die Weiterentwicklung erfolgt unabhängig von der Android Plattform
- Jetpack-Klassen sind unter androidx.* definiert
- AndroidX ersetzt Android Support Libraries

7.1 View Binding

Vereinfacht den Zugriff auf View-Elemente. (Kein findViewById()) mehr, Typ- und Null-Sicherheit). Erzeugung von Binding-Klassen beim Build (Aktivierung über Gradle)

Namensgebung: Layout-Name als Camel Case + Binding.

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

// activity_main.xml
<LinearLayout
   android:layout_width="match_parent"
   android:layout_height="match_parent">
   <Button
      android:id="@+id/button_hello"
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:text="Hello World!" />
</LinearLayout>

// MainActivity.java
public class MainActivity extends AppCompatActivity {
   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 -> { });
   }
}
```

7.2 Data Binding

Erlaubt im XML Zugriff auf Objekte (Layouts als Observer der Daten, Einfache Logik direkt im XML möglich). Basiert auf Binding-Klassen (Aktiviert über Gradle, Generiert beim Build).

```
// build.gradle
android {
   buildFeatures {
      dataBinding true
   }
}

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

// activity_main.xml
<layout xmlns:android=" ... ">
   <data>
      <variable name="user" type="ch.ost.rj.mge.v07.User
          "/>
   </data>
   <LinearLayout
      android:layout_width="match_parent"
      android:layout_height="match_parent">
      <TextView
         android:id="@+id/first"
         android:layout_width="wrap_content"
         android:layout_height="wrap_content"
         android:text="@{user.firstName}" />
      <TextView
         android:id="@+id/last"
         android:layout_width="wrap_content"
         android:layout_height="wrap_content"
         android:text="@{user.lastName}" />
      <TextView
         android:layout_width="wrap_content"
         android:layout_height="wrap_content"
         android:text="@{first.text + ' ' + last.text}"
             />
   </LinearLayout>
</layout>

// MainActivity.java
public class MainActivity extends AppCompatActivity {
   private ActivityMainBinding binding;
   @Override
   protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      binding = DataBindingUtil.setContentView(this, R.
          layout.activity_main);
      User user = new User("Joel", "Schaltegger");
      binding.setUser(user);
   }
}
```

7.2.1 Expression Language

Die Bindings im Layout werden in einer Expression Language definiert.

- Mathematical (+ , - , / , * , %)
- String concatenation
- Logical (&& , ||)
- Vergleich (== , Grösser/Kleiner als usw.)
- ...

```
// Beispiele:
android:text="@{String.valueOf(index + 1)}"
android:visibility="@{age > 13 ? View.GONE :View.VISIBLE
    }"
```

7.2.2 Event Handling

- Auch Events können gebunden werden
- **Method References** verweisen direkt auf Methoden mit passender Signatur
- **Listener Bindings** erlauben die Verwendung von Ausdrücken vor dem Aufruf der Methode

```
<layout xmlns:android="…">
  <data>
    <variable name="handler" type="(…).EventHandler"/>
  </data>
  …
  <Button
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="Method Reference"
    android:onClick="@{handler::doSomething}"/>
  <Button
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="Listener Binding"
    android:onClick="@{v -> handler.doSomething(v, "...")}"/>
</layout>
```
activity_main.xml

```
public class EventHandler {
    public void doSomething(View view) {
    }
    public void doSomething(View view, String text) {
    }
}
```
EventHandler.java

Data Binding – Änderungen an Daten

Variante des Observer-Patterns

- Ein in Data Bindings verwendetes Objekt wird **nicht** automatisch observierbar
- Für die automatische Aktualisierung der View muss die Datenquelle angepasst werden
  - **Observable Fields** für einfache Datentypen
  - **Observable Objects** für eigene Klassen

```
// Observable Fields
public class User {
    public final ObservableField<String> firstName = new …();
    public final ObservableField<String> lastName = new …();
    public final ObservableInt age = new …();
}
```
User.java

```
// Observable Class
public class User extends BaseObservable {
    private String firstName;
    private String lastName;
    private int age;

    @Bindable
    public String getFirstName() { return this.firstName; }

    @Bindable
    public String getLastName() { return this.lastName; }

    @Bindable
    public int getAge() { return this.age; }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
        notifyPropertyChanged(BR.firstName);
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
        notifyPropertyChanged(BR.lastName);
    }

    public void setAge(int age) {
        this.age = age;
        notifyPropertyChanged(BR.age);
    }
}
```
User.java

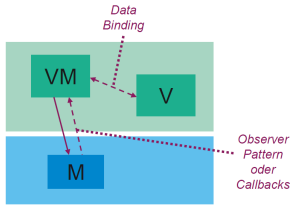
7.2.3 Two-Way-Bindings

In vielen Fällen ist Two-Way nötig: z.B. Login-Checkbox in den Übungen
Notation: = vor der Binding Expression

7.3 MVVM (Model, View, View-Model)

Durch Data Binding können schlanke, besser testbare Activites/Fragemente erstellt werden.

Risiken:
- Model mit Android-Details verunreinigt
- Zu viel Logik im Layout (Expression Language)
- Bei Fehlern erschwertes Debugging
- Erhöhter Zeitbedarf für Kompilierung
- Gefahr für "unsichtbare Observer"



Bestandteile:
- Model: enthält Daten- und Domänenklassen (Businesslogik)
- View: umfasst die grafische Benutzeroberfläche & Benutzereingaben
- ViewModel: enthält die Logik des UI und vermittelt zwischen Model und View

Vorteile:
- Das ViewModel ist einfach zu testen, da es keine UI-Klassen enthält
- Die View kümmert sich um rein visuelle Aspekte (keine Logik)
- Änderungen am Model haben keine Direkten Auswirkungen auf die View

7.3.1 MVVM im Eigenbau

```
// UserActivity.java
public class UserActivity extends AppCompatActivity {
    private ActivityUserBinding binding;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        User user = new User("Joel", "Schaltegger", 23);

UserViewModelFactory factory = new UseViewFactory(user);
        UserViewModel viewModel = new ViewModelProvider(
            this, factory).get(UserViewModel.class);

        binding = ActivityUserBinding.inflate(...);
        binding.setVm(viewModel);
        binding.setLifecycleOwner(this);

        setContentView(binding.getRoot());
    }
}

// UserViewModel.java
public class UserViewModel {
    private final User user;

    public final MutableLiveData<String> name = new M
        ...<>();
    public final MutableLiveData<Integer> age = new M
        ...<>();

    public ViewModelObservableFields(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();
    }
}

// Factory.java
public class UserViewModelFactory 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);
    }
}
```

7.4 ViewModel und Fragments

- Die Interaktion zwischen Fragments kann sehr komplex werden
  - Viele Callback Interfaces
  - Indirekte Kommunikation über Parent-Activity
- View Models können helfen
  - Ein View Model pro Activity
  - Fragments verwenden Teile des View Models
  - Nachteil: Fragments verlieren Unabhängigkeit