Mobile and GUI Engineering
Android

Oliviero Chiodo
Stefano Kals

Hochschule für Technik Rapperswil

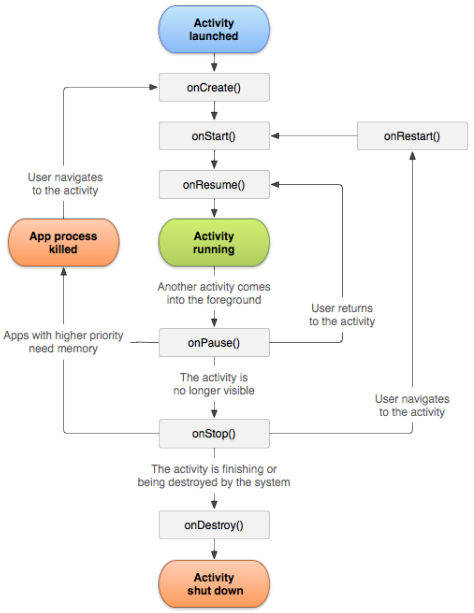
Herbstsemester 2015

Grundlagen der GUI Programmierung

Apps bestehen aus lose gekoppelten, wiederverwendbaren Komponenten. Diese sind Activities (für den Benutzer sichtbar) und Services, Content Provider, Broadcast Receivers (unsichtbar für Benutzer). Das System hat die Kontrolle über alle Applikationen und Verwaltet den Lebenszyklus, ist verantwortlich für die Kommunikation zwischen Komponenten und schliesst die Apps automatisch um Speicher zu sparen.

Activites Dies sind die Hauptbausteine in der App Entwicklung, die Activity interagiert mit dem Benutzer.

```
public class MainActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState)
      ↪ {
        super.onCreate(savedInstanceState);
        /* ... */
    }
}
```



Activities werden in einem Stack verwaltet, wobei die Activites eines Stacks zu verschiedenen Apps gehören können.
Eine Gruppe von Activities in einem Stack nennt man auch **Task**. Es können mehrere Tasks gleichzeitig existieren.

Activity Launch Modes Activities haben verschiedene Launch Modes

- **Standard:** Es wird eine neue Activity in den Stack gepusht
- **SingleTop:** Ist Activity A zuoberst auf dem Stack und sie wird nochmals gestartet, wird in Activity A `onNewIntent()` ausgeführt
- **SingleTask:** Das System erstellt einen neuen Task und instanziert die Activity als die Root des neuen Tasks. Wenn aber schon eine Instanz dieser Activity existiert, wird `onNewIntent()` in dieser aufgerufen.
- **SingleInstance:** Es darf nur eine Instanz der Activity geben

Intent Ein Intent beschreibt was gemacht werden soll und das System entscheidet wer zuständig ist. Apps können selbst wiederum Activites zur Verfügung stellen und bestehende Applikationen ersetzen. Andere Activities können explizit oder implizit aufgerufen werden:

```
// Explizit mit Klasse
Intent in = new Intent(this, CalculateActivity.class)
// Implizit mit Aktion
Intent in = new Intent(MediaStore.ACTION_IMAGE_CAPUTRE
    ↪ )
```

Andere Activities kann man mit `startActivity(intent)` oder `startActivityForResult(intent, myId)` starten. Um bei letzterem mit dem Rückgabewert arbeiten zu können muss man `onActivityResult` überschreiben.

```
@Override
protected void onActivityResult(int request, int
    ↪ result, Intent data) {
    if (result == Activity.RESULT_OK && request ==
      ↪ myId) {
        /* ... */
    }
}
```

Möchte man einem Intent Daten übergeben, kann man das entweder mit der `setData` Methode, welche eine URI entgegennimmt, oder mit `putExtra(MediaStore.EXTRA_OUTPUT, imageCaptureUri)`. Letzere ist eine Struktur mit Key-Value Paaren und nimmt nur primitive Daten, Strings und serialisierbare Datentypen an.

Manifest Das Manifest enthält Meta-Daten einer App. Es umfasst

- Komponenten der App
- Metadaten (Name, Icon, Versionsnummer)
- Permissions (Internet, kostenpflichtige Anrufe etc.)
- Anforderungen an die Geräte API

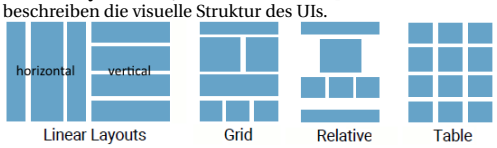
Das Manifest wird vom System verwendet um zu wissen, ob die App installiert werden kann, welche Permissions diese verwendet etc.

Android GUI

Aufbau Die Basisklasse um User Interfaces zu bauen ist die `View`. Eine View ist zuständig seinen Inhalt zu zeichnen und Events zu behandeln. Untergruppen der View sind Widgets und ViewGroups. Widgets ist ein Sammelbegriff für alle fix-fertigen Komponenten für User-Interfaces (Buttons, Images, Checkboxes etc.)

ViewGroup Die ViewGroup ist eine Unterklasse von View. Sie kann andere View beinhalten. Wenn die ViewGroup beinhaltende View anordnet, spricht man von einem Layout.

Basis Layout Layouts sind ViewGroups und beschreiben die visuelle Struktur des UIs.



Die Layout-Parameter beschreiben wie die Views angeordnet und dargestellt werden. Für alle ViewGroups gemeinsam sind `android:layout_width` und `android:layout_height`. Häufig benutze Werte sind `match_parent` (So gross wie mögich, also wie der Parent erlaubt) und `wrap_content` (so klein wie möglich, also wie die Kinder erlauben).
In einem **Linear Layout** werden die Elemente horizontal oder vertikal angeordnet. Mit `android:layout_weight` kann man Elementen ein Gewicht geben, da es selten sinnvoll ist, alle Elemente gleich gross zu lassen. Kinder ohne Weight bekommen minimalen Platz, auf die restlichen wird der verfügbare Platz nach Gewicht aufgeteilt.
Das **Relative Layout** ist das vielseitigse Layout welches Kinder relative zueinander anordnet.

```
<RelativeLayout xmlns:android="... >
    <TextView
        android:text="1. Platz"
        android:id=@+id/first"
        android:layout_centerHorizontal="true" />
    <TextView
        android:text="2. Platz"
        android:id="@+id/first"
        android:layout_below="@id/first"
        android:layout_toStartOf="@id/first" />
    <TextView
        android:text="3. Platz"
        android:id="@+id/textView3"
        android:layout_below="@id/first"
        android:layout_toEnfOf="@id/first" />
</RelativeLayout>
```

Das **Frame Layout** kann die Kinder übereinander anordnen (Hilfslinien bei einer Kamera-App).
Das Layout muss in der Activity deklariert werden. Dies wird in der Activit Klasse mit `setContentView(R.layout.activity_main)` gemacht.

Widgets

Button Vom Button gibt es einen normalen `Button` und einen `ImageButton` bei dem man mit dem `android:src` ein Bild einfügen kann.

Eingabefelder Der Typ des Eingabefeld bestimmt welche Tastatur verwendet wird. Dies kann man mit dem `android:inputType` bestimmen, es sind auch Kombinationen möglich. Innerhalb `EditText`

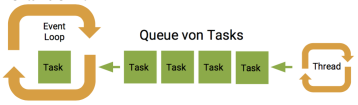
Referenzen und ID Möchte man GUI-Elemente referenzieren, gibt man ihnen einen ID-String, welche Strings sind, die mit `@` beginnen. Wenn man einen neue ID definieren will, macht man das mit `@+id/`. Das Android-Buildsystem sammelt alle diese IDs als Konstanten in der automatisch generierten Klasse R. Diese Klasse enthält alle Ressourcen als Konstanten. Weitere Resourcesn sind: `drawable` (Bilder), `menu` (Menüs), `mipmap` (Launcher Icon der App) und `values` (Strings und andere Konstanten). Mit Aussnahme der values, wird für jeden Ordner eine innere Klasse in R generiert.

Dimensionen in Ressourcen Konstanten in `dimens.xml` werden für Grössen in den Layouts benutzt.

```
<!-- Layout -->
<RelativeLayout xmlns:android="..." xmlns:tools="..."
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingLeft="@dimen/
      ↪ activity_horizontal_margin"
    android:paddingRight="@dimen/
      ↪ activity_horizontal_margin"
    android:paddingTop="@dimen/
      ↪ activity_vertical_margin"
    android:paddingBottom="@dimen/
      ↪ activity_vertical_margin"
    tools:context=".MainActivity">
<!-- dimens.xml -->
    <!-- Default screen margins, per the Android
      ↪ Design guidelines. -->
    <dimen name="activity_horizontal_margin">16dp</
      ↪ dimen>
    <dimen name="activity_vertical_margin">16dp</dimen
      ↪ >
</resources>
```

Grössenangeben von Views erfolgen in density-indepentent Pixels (dp oder dip). Bei Schriften verwendet man scale-independent Pixels (sp).

Events und Event Handling Das Android-Framework hat einen sogenannten Event-Loop (Looper). Dieser wartet bis ein Ereignis passiert und verarbeitet dieses dann. Nur der Main-Thread darf das GUI verändern.



Fast alle Listener können mit `set[EventName]Listener` registriert werden.

```
button.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        /* ... */
    }
});
```

Oder onClick Listener in XML deklariert

```
android:onClick="onButtonClicked"
```

Alternativ zur anonymen Klasse, kann die Activity auch das Interface implementieren. Bei der Verarbeitung von Texteingaben haben wir mehrere Möglichkeiten auf Events zu reagieren. Das zu implementierene Interface (`TextWatcher`) umfasst 3 Methoden:

- `beforeTextChanged`: Wird aufgerufen bevor der Text geändert wird
- `onTextChanged`: Wird aufgerufen sobald der Text geändert hat
- `afterTextChanged`: Nachdem der Text geändert wurde, kann man den Text noch anpassen (Loop-Gefahr)

```
editText.addTextChangedListener(new TextWatcher() {
    public void onTextChanged(CharSequence s, int
      ↪ start, int before, int count){ }
    public void beforeTextChanged(CharSequence s, int
      ↪ start, int count, int after){ }
    public void afterTextChanged(Editable s) { }
});
```

Bei Eingabefeldern kann man mit `setError` Meldungen anzeigen lassen. Dies eignet sich gut für eine Inputvalidierung. Bei jeder Änderung wird diese zurückgesetzt.

```
final EditText password = (EditText) findViewById(R.id
    ↪ .password);
password.addTextChangedListener(new TextWatcher() {
    @Override
    public void afterTextChanged(Editable s) {
        String pw = s.toString();
        if (s.length() < 8) {
        password.setError("Passwort muss mindestens 8
      ↪ Zeichen lang sein.");
        }
    }
    ...
});
```

Android Testing mit JUnit

Es gibt mehrere Arten von Tests für eine Activity. Die `ActivityUnitTest` manipulieren das UI im Code, die `ActivityInstrumentationTestCase2` schickt Clicks und Events an das UI. Um eine App mit mehreren Activities zu testen, gibt es das Espresso Framework, sowie UI Automator um App-übergreifend zu testen.

```
public class MainActivityLayoutTest extends
    ↪ ActivityUnitTestCase<MainActivity> {
    public MainActivityLayoutTest() {
        super(MainActivity.class);
    }
    @Override
    protected void setUp() throws Exception {
        super.setUp();
        ContextThemeWrapper context = new
            ContextThemeWrapper(getInstrumentation().
      ↪ getTargetContext(), R.style.AppTheme);
        setActivityContext(context);
        startActivity(
            new Intent(getInstrumentation().
      ↪ getTargetContext(), MainActivity.class),null,
      ↪ null);
}
```

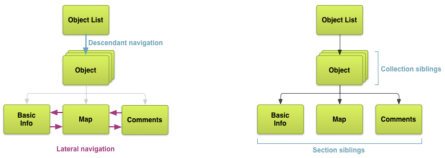
Funktionale Tests (ActivityInstrumentationTestCase2) testen eine Activity im echten Systemkontext. Der Test löst Events aus und prüft, ob diese zum erwünschten Resultat führen. Dazu gehören:

- Ändert sich das UI wie erwartet
- Überprüfen von Inputvalidierung
- Werden Lifecycle Events korrekt behandelt

```
public class MainActivityInteractionTest extends
    ↪ ActivityInstrumentationTestCase2<MainActivity
    ↪ > {
    public MainActivityInteractionTest() { super(
    ↪ MainActivity.class); }
    public void testSayHi() {
        MainActivity activity = getActivity();
        final EditText editText = (EditText) activity.
    ↪ findViewById(R.id.editText);
        getInstrumentation().runOnMainSync(new
    ↪ Runnable() {
            @Override
            public void run() {
                editText.requestFocus();
            }
        });
        getInstrumentation().waitForIdleSync();
        getInstrumentation().sendStringSync("Hello");
        getInstrumentation().waitForIdleSync();
        Button button = (Button) activity.findViewById
    ↪ (R.id.button);
        TouchUtils.clickView(this, button);
...
```

Navigation

Die Navigation in der App lässt sich durch eine gute Domainanalyse vorbereiten. Mit dieser können die Screens entwickelt werden, die es braucht um mit den Daten umgehen zu können.
Danach kann man Screens gruppieren. Dies hilft das Design an grosse und kleine Bildschirme anzupassen (siehe Reflow-Pattern, WED2). Danach kann man die Beziehung zwischen den Screens festlegen. Es gibt die **Parent-Child** Hierarchie, und die **Siblings**.



Bei der Rückwärtsbeziehung unterscheidet man ebenfalls zwischen 2 Navigationstypen. Die **Ancestral Navigation** führt zum hierarchischen Parent, die **Temporal Navigation** führt zum vorherigen Element. Ancestral Navigation geschieht über den Up oder Home Button, Temporal Naviagtion über den Back-Button.
Grundsätzlich geht man beim Navigation-Design folgendermassen vor:

1. Domain-Modell entwerfen
2. Screens ableiten
3. Screens in Beziehung bringen und gruppieren
4. Navigation zwischen den Screens festlegen
5. Wireframe/Storyboard für die Gesamtübersicht erstellen
6. Usability Test mit einem Papier-Prototypen des Wireframes

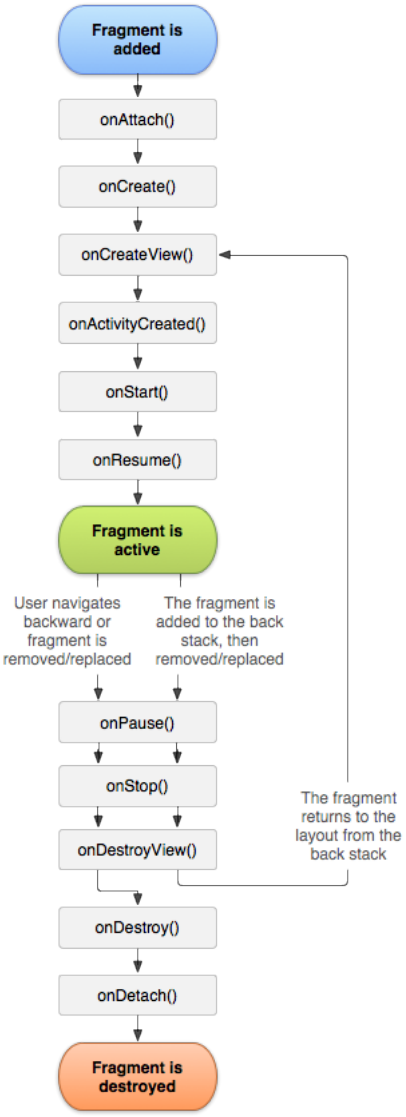
Fragment

Es ist nicht möglich mehrere Activities auf einen Screen zu tun. Mit Fragments kann man dies tun. Ein Fragment ist ein modularer Teil einer Activity mit eigenem Lebenszyklus.

```
public class MainActivityFragment extends Fragment {
    public MainActivityFragment() {}
    @Override
    public View onCreateView(LayoutInflater inflater,
    ↪ ViewGroup container,
                             Bundle savedInstanceState)
    ↪ {
        return inflater.inflate(R.layout.fragment_main
    ↪ , container, false);
    }
}
```

Ein Fragment kann im XML in eine Activity eingefügt werden.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.
    ↪ com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="horizontal">
    <fragment xmlns:android="http://schemas.android.
    ↪ com/apk/res/android"
        xmlns:tools="http://schemas.android.com/tools"
        android:id="@+id/fragment"
        android:name="com.example.
    ↪ myfragmentapplication.MainActivityFragment"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        tools:layout="@layout/fragment_main" />
</LinearLayout>
```



Ein Fragment wird einem `LayoutInflager` übergeben, der nimmt XML entgegen und instanziert die View Klassen. Oder dynamisch, mit dem FragmentManager im Java Code

```
public class MainActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState)
    ↪ {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        FragmentManager fm = getFragementManager();
        FragmentTransaction ft = fm.beginTransaction()
    ↪ ;
        MainActivityFragment f = new
    ↪ MainActivityFragment();
        ft.add(R.id.fragment_container, f);
        ft.commint();
    }
...
}
```

Best Practice ist, dass Fragments ein Interface zur Kommunikation definieren, welches Parent-Activity implementieren muss.

```
public class MainActivityFragment extends Fragement {
    public interface OnItemSelectedlistener {
        void onItemSelected(String item);
    }
    OnItemSelectedListener parentActivity;
    @Override
    public void onAttach(Activity activity);
        super.onAttach(activity);
        try {
            parentActivity = (OnItemSelectedListener)
    ↪ activity;
        } catch (ClassCastException e) {
            throw new ClassCastException("Exception");
        }
    }
}
```

Master-Detail Navigation

Die Unterscheidung zwischen den Anzeigevarianten wird über Layouts getroffen. Das Default-Layout der `ItemListActivity` beinhaltet nur das `ItemListFragment`. Das Tablet enthält ein Linear-Layout mit dem `ItemListFragment` sowie ein Platzhalter für das `ItemDetailFragment`. Das Kriterium für das Tablet-Layout ist *sw600dp*(smallest width 600dp).

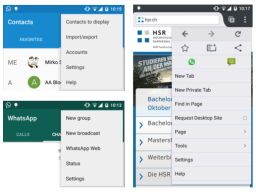
```
public clas ItemListActivity extends Activity
    ↪ implements ItemListFragment.Callbacks {
    private boolean twoPane;
    @Override
    protected void onCreate(Bundle savedInstanceState)
    ↪ {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_item_list);
        if(findViewById(R.id.item_details_container)
    ↪ != null) {
            twoPane = true;
        }
    }
    @Override
    public void onItemSelected(String id) {
        if(twoPane) {
            Bundle arguments = new Bundle();
            arguments.putString(ItemDetailFragment.
    ↪ ARG_ITEM_ID, id);
            ItemDetailFragment fragment = new
    ↪ ItemDetailFragment();
            fragment.setArguments(arguments);
            getFragmentManager()
                .beginTransaction()
                .replace(R.di.item_detail_container,
    ↪ fragment)
                .commit();
        } else {
            Intent detailIntent = new Intent(this,
    ↪ ItemDetailActivity.class);
            detailIntent.putExtra(ItemDetailFragment.
    ↪ ARG_ITEM_ID, id);
            startActivity(detailIntent);
        }
    }
}
```

Activities sollte man verwenden, wenn man einen Einstiegspunkt in einen Task braucht. **Wichtig:** Activities leben weiter, wenn wir eine neue Activity starten.

Menus

Options Menu

Das Options Menu ist teil der ActionBar. Es enthält Actions die generell für die App/Activity gedacht sind. Apps mit Navigation Drawer haben teilweise kein Options Menu mehr.



```
public boolean onCreateOptionsMenu(Menu menu) {
    menu.add(0, START_MENU_ITEM, 0, "Start");
    menu.add(0, SUBMIT_MENU_ITEM, 0, "Submit");
    return true;
}
public boolean onOptionsItemSelected(MenuItem item){
    switch(item.getItemId()){
        case START_MENU_ITEM:
            // start
            return true;
        case SUBMIT_MENU_ITEM:
            // submit
            return true;
    }
    return super.onOptionsItemSelected(item);
}
```

Der bessere Ansatz wäre die deklarative Implementation im XML.

```
<menu xmlns:android="http://schemas.android.com/apk/
    ↪ res/android"
xmlns:tools="http://schemas.android.com/tools"
    ↪ tools:context=".MainActivity">
    <item android:id="@+id/action_search"
        android:title="@string/action_search"
        android:icon="@drawable/ic_action_search"
        android:orderInCategory="100"
        android:showAsAction="never" />
    <item android:id="@+id/action_settings"
        android:title="@string/action_settings"
        android:orderInCategory="100"
        android:showAsAction="never" />
</menu>
```

```
public class MainActivity extends Activity {
    public boolean onCreateOptionsMenu(Menu menu) {
        getMenuInflater().inflate(R.menu.menu_main,
    ↪ menu);
        return true;
    }
    public boolean onOptionsItemSelected(MenuItem item
    ↪ ) {
```

```
        /* ... */
    }
}
```

Settings-Page Die Settings-Page erbt von der Klasse `PreferenceActivity`. Sie kann im `PreferenceScreen` Tag definiert werden.

```
<!-- preferences.xml -->
<PreferenceScreen xmlns:android="...">
        <CheckBoxPreference
        android:defaultValue="true"
        android:key="@string/sync_pref"
        android:title="Synchronize" />
    <MultiSelectListPreference
        android:entries="@array/languages"
        android:entryValues="@array/languages"
        android:key="@string/lang_pref"
        android:title="Languages" />
</PreferenceScreen>
<!-- resources.xml -->
<resources>
    <string name="lang_pref">language</string>
    <string name="sync_pref">synchronize</string>
    <string-array name="languages">
        <item>German (CH)</item>
        <item>English (UK)</item>
    </string-array>
</resources>
```

```
public class Preferences extends PreferenceActivity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        getFragmentManager()
            .beginTransaction()
            .replace(R.id.content, new PrefsFragment())
            .commit();
        PreferenceManager.setDefaultValues(this, R.xml
          ↪ .preferences, false);
    }
    public static class PrefsFragment extends
      ↪ PreferenceFragment {
        @Override
        public void onCreate(Bundle savedinstanceState
      ↪ ) {
            super.onCreate(savedInstanceState);
            addPreferencesFromResource(R.xml.
      ↪ preferences);
        }
    }
}
```

In der MainActivity mit dem Settings-Menueintrag brauchen wir bloss noch die neue Activity zu starten.

```
@Override
public boolean onOptionsItemSelected(MenuItem item){
    int id = item.getItemId();
    if(id == R.id.action_settings) {
        startActivity(new Intent(this, Prefernces.
      ↪ class));
        return true;
    }
    return super.onOptionSItemSelected(item);
}
```

Die Einstellung wird mittels dieser komplizierten Funktion ausgelesen.

```
SharedPreferences settings = PreferenceManager
    .getDefaultSharedPreferences(getAppliction());
settings
    .getBoolean(
        getString(
            R.string.sync_pref
        ), true);
```

Auch Fragments können Einträge dem Menu der Activity hinzufügen. Die Behandlung erfolgt analog entweder im Fragment oder in der Activity.

```
public class MainFragment extends Fragment {
    @Override
    public void onCreateOptionsMenu(Menu menu,
      ↪ MenuInflater inflater) {
        inflater.inflate(R.menu.menu_main, menu);
    }
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setHasOptionsMenu(true);
    }
}
```

Context und Popup Menu Ein Context Menü für Actions welche die selektierte View betreffen. Meist ändert sich bei einer Selektion die ActionBar.
Popup Menüs lassen sich z.B an einen beliebigen Button binden (Vergleichbar mit einem Spinner/Combobox).
Popup Menus eignen sich für Overflow-Style Menu für Actions die Context spezifisch sind (beispielsweise Weiterleiten und Antworten bei Mail).

```
<ImageButton
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:src="@drawable/ic_overflow_holo_dark"
    android:contentDescription="@string/
      ↪ descr_overflow_button"
    android:onClick="showPopup" />
```

```
public void showPopup(View v) {
    PopupMenu popup = new PopupMenu(this, v);
    MenuInflater inflater = popup.getMenuInflater();
    inflater.inflate(R.menu.actions, popup.getMenu());
    popup.show();
}
```

Action Bar Die ActionBar war der erste Versuch eine normierte "Schnittstelle" für Aktionen in einer App zu kreieren.

1. App Icon und ev. Up-/Home-Navigation
2. Name der App oder View-Switcher
3. Actions (Teil des Options Menu)
4. Action-Overflow mit dem Rest des Menus

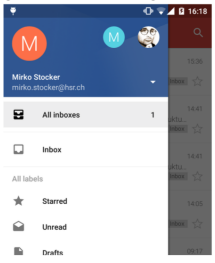
Toolbar Die Toolbar soll ab Android 5.0 die ActionBar ablösen. Sie ist flexibler aber noch nicht so weit verbreitet. Um sie in älteren Android Versionen verwenden zu können ist eine Support-Library nötig.


```
<RelativeLayout xmlns:android="..." xmlns:app="..."
    ↪ xmlns:tools="..."
    android:layout_width="match_parent"
    ↪ android:layout_height="match_parent"
    tools:context=".MainActivity">
    <android.support.v7.widget.Toolbar
    android:id="@+id/toolbar"
    android:layout_width="match_parent"
    android:layout_height="wrap_content">
    </android.support.v7.widget.Toolbar>
    <fragment
        ...
        android:layout_below="@+id/toolbar"
        tools:layout="@layout/fragment_main" />
</RelativeLayout>
```

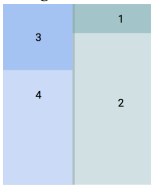
Damit die Toolbar als ActionBar funktioniert braucht es zusätzliche Konfiguration im Java Code

```
public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState)
    ↪ {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Toolbar toolbar = (Toolbar)findViewById(R.id.
      ↪ toolbar);
        setSupportActionBar(toolbar);
    }
}
```

Navigation Drawer Dies ist eine Art Hauptmenü. Es wechselt zwischen Activities oder Fragments. Es können auch gewisse Einstellungen verändert werden (Toggle).



Der Drawer benötigt eine Layoutdefinition und eine Menudeklaration. Das Layout der Activity ist rechts vom Navigation Drawer und braucht ein FrameLayout.



1. Toolbar, 2. Content, 3. Header, 4. Menu-Items

```
<android.support.v4.widget.DrawerLayout xmlns:android=
    ↪ "..." xmlns:app="..." xmlns:tools="..."
    android:id="@+id/drawer_layout"
    android:fitsSystemWindows="true"
    tools:context=".MainActivity">
    <FrameLayout
        android:id="@+id/content"
        android:orientation="vertical">
        <android.support.v7.widget.Toolbar
            android:id="@+id/toolbar"/>
        <TextView ... />
    </FrameLayout>
    <android.support.design.widget.NavigationView
        android:id="@+id/navigation_view"
        android:layout_gravity="start"
        app:headerLayout="@layout/drawer_header"
        app:menu="@menu/drawer" />
</android.support.v4.widget.DrawerLayout>
```

In der Activity kann ein Listener registriert werden, welcher auf Menüpunkte des Drawers reagiert.

```
NavigationView view = (NavigationView)findViewById(R.
    ↪ id.navigation_view);
view.setNavigationItemSelectedListener(
        new NavigationView.
    ↪ OnNavigationItemSelectedListener() {
    @Override
    public boolean onNavigationItemSelected(MenuItem
    ↪ item) {
        item.setChecked(true);
        drawerLayout.closeDrawers();
        return true;
    }
});
```

Toast und Snackbar Toast sind kleine Feedbacknachrichten. Sie werden mit `Toast.makeText(...).show()` angezeigt.

Design Patterns

Listen

Listen sind ein häufig verwendetes UI-ELement. `ListView` und `ExpandableListView` sind die implementierenden Klassen. Die Einträge können komplexe Layouts sein und es könne beliebige Java-Objekte dargestellt werden.

```
<ListView
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:id="@+id/listView"/>
```

Da die ListView mit allen Java Klassen umgehen können muss, verwendet es das Adapter Design Pattern. Das Datenmodell soll unabhängig von der ListView bleiben, und der Adapter vermittelt zwischen Daten-Klassen und ListView. ListView und GridViews sind AdapterViews. Für das zuweisen eines Adapters wird `setAdapter()` verwendet.

```
ArrayAdapter<String> adapter = new ArrayAdapter<>(this
    ↪ , R.layout.rowlayout, R.id.label);
listView.setAdapter(adapter);
```

Möchte man einen eigenen ArrayAdapter implementieren muss man in diesem die `getView` Methode überschreiben um zu kontrollieren, wie genau ein Eintrag aussieht.

```
View getView(int position, View convertView, ViewGroup
    ↪ parent);
```

Wobei `position` der Index ist, der dargestellt werden soll, `convertView` die alte View (oder null) ist, und `parent` den zukünftigen Parent. Die alte View wird aus Performancegründen mitgegeben, damit nicht immer eine neue Instanz des Layouts erstellt sondern nur die Daten aktualisiert werden. Eine ListView mit Checkboxen:

```
public View getView(int position, View convertView,
    ↪ ViewGroup parent) {
    final Module module = moduleList.get(position);
    if(convertView == null) {
        LayoutInflater lf = (...)getSystemService(
      ↪ Context.LAYOUT_INFLATER_SERVICE);
        convertView = lf.inflate(R.layout.rowlayout,
      ↪ null);
    }
    TextView tv = (...)convertView.findViewById(R.di.
      ↪ textView);
    CheckBox cb = (...)convertView.findViewById(R.id.
      ↪ checkbox);
    tv.setText("(" + module.getCode() + ")");
    cb.setText(module.getName());
    cb.setChecked(module.isSelected());
    cb.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            CheckBox b = (...)v;
            module.setSelected(!b.isChecked());
        }
    });
    return convertView;
}
```

Die Performance der ListView ist nur bedingt gut. `findViewById` sind teure Operationen und diese werden mit komplexeren Layouts mehr.

View Tags Views können getagged werden (`setTag(Object tag)`). Man kann das mit einem Key-Value Paar machen oder einfach mit einem Object. So kann man beliebige Daten an eine View anhängen.

Recycler View

Die RecylcerView hat mehrere Verbesserungen gegenüber den anderen beiden Views. Es können mehrere Layoutmanager definiert werden, es können Animationen für Hinzufügen und Entfernen von Einträgen gesetzt werden, und das Recyling von Elementen wurde fest eingebaut. Leider ist es mit der RecyclerView mühsamer das Click-Handling zu implementieren. Die Komponenten der RecylcerView sind:

- **Adapter:** Analog zu eigenem ArrayAdapter (leicht anderes API)
- **ViewHolder:** Klasse welche die Views zwischenspeichert
- **LayoutManager:** Defaults für übliche Layouts (optional)
- **ItemDecorator:** Trennlinien oder andere Dekoratoren um die Elemente (optional)
- **ItemAnimator:** Animiert hinzufügen, entfernen und scrolling von Elementen (optional)

```
public class MyAdapter extends RecyclerView.Adapter<
    ↪ ViewHolder> {
    private ArrayList<Module> dataset;
    public MyAdapter (ArrayList<Module> modules) {
      ↪ dataset = modules;}
    public ViewHolder onCreateViewHolder(ViewGroup
      ↪ parent, int viewType){
        /* ~~ */
    }
    public onBindViewHolder(ViewHolder holder, int
      ↪ position) {
        /* ... */
    }
}
```

Listing 1: "RecyclerView Schnittstelle"

Design und Architektur Patterns

Um viele Klassen zu ordnen kann man beispielsweise die Multitier Architecture verwenden. Darin sind die Abhängigkeiten klar gerichtet und es existieren keine Zyklen. Um nun die Domain überwachbar zu machen, kann man das Observer Pattern implementieren.

Swipe Refresh Anstelle, dass unsere App sich selbst aktualisiert kann man das mit einem Swipe Refresh machen. Dafür braucht man eine Support Library.

```
<android.support.v4.widget.SwipeRefreshLayout>
    <android.support.v7.widget.RecyclerView/>
</android.support.v4.widget.SwipeRefreshLayout>
```

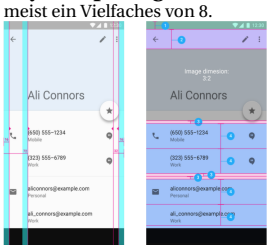
```
final SwipeRefreshLayout swipe =
    (SwipeRefreshLayout) findViewById(R.id.
      ↪ swipeRefreshLayout);
swipe.setOnRefreshListener(new SwipeRefreshLayout.
    ↪ OnRefreshListener() {
    public void onRefresh() {
        /*...*/
        swipe.setRefreshing(false);
    }
});
```

Material Design

Material Design ist eine Design Sprache (Design Language). Eine Design Sprache ist eine Hilfestellung für den Designprozess. Es beschreibt, wie die Teile einer Applikation aussehen und sich verhalten sollen. Material ist die Metapher und arbeitet im 3D-Raum mit Licht und Schatten. Es ist angelehnt an physikalische Begebenheiten. Man verwendet ein Grid für die Ausrichtung und versucht, wenn angemessen eine Reaktion auf User-Input zu geben. Materials sind geometrische Formen (Schnipsel aus Papier) mit einer Dicke von exakt 1dp. Duch die unterschiedliche Anordnung auf der Z-Achse entsteht Schatten. Dieses Konzept hat Einfluss auf verschiedene Aspekte. Material Design Styleguides umfasset deshalb auch: Layout, Style, Animation, Components, Patterns und Usability.

Layout Grundelement ist das Papier, das an- und übereinander gelegt werden kann. Ein Floating Action Button beispielsweise, bietet eine Aktionsmöglichkeit für das Papier. Alle anderen Materialien werden an einem 8dp Grid ausgerichtet.

Layout Spacing Der Abstand zwischen Elementen ist meist ein Vielfaches von 8.



1. Status bar: 24dp
2. Toolbar: 56dp
3. Space: 8dp
4. List item: 72dp

Style Für Material Design wird empfohlen, dass man 3 Farbtöne der Primärpalette und eine Akzentfarbe aus einer zweiten Palette wählt.

Animation helfen die Illusion aufrecht zu halten, dass wir die Materialien auf dem Screen direkt manipulieren (wie durch eine Glasscheibe). Man sollte visuelles Feedback auf Input geben und den Übergang zwischen Zuständen animieren.

Usability Wichtige Punkte in Usability sind:

- Touch-Targes nicht zu klein machen
- Navigation soll nachvollziehbar und nicht komplex sein
- Fontgrössen, Farben
- RTL-Layouts werden (wenn richtig gemacht automatisch) gespiegelt
- Texte in Grafiken sind problematisch

Umsetzung

View Styling Views können direkt im XML-Layout gestyled werden.

```
<Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="New Button"
    android:id="@+id/button2"
    android:layout_gravity="center"
    android:background="#ff85e1"
    android:height="36dp"
    android:minWidth="64dp"
    android:padding="8dp" />
```

Meist ist es am schönsten wenn man so etwas in ein Style auslagert.

```
<style name="MyButtonStyle">
    <item name="android:background">#ff85e1</item>
    <item name="android:height">36dp</item>
    <item name="android:minWidth">64dp</item>
    <item name="android:padding">8dp</item>
</style>
```

Dies wird dann mit `style="style/MyButtonStyle"` referenziert. Auch diese Ressourcen können für unterschiedliche Geräte, Versionen etc. spezifiziert werden.

Themes sind Styledefinitionen die für eine Activity oder App gelten

```
<application
    ...
    android:theme="@style/AppTheme">
    <activity
        android:name".MainActivity"
        android:theme="@style/AppTheme" >
```

Das AppTheme ist das Standarttheme, und müsste nicht spezifiziert werden. Im `styles.xml` kann man die entsprechenden Styles definieren.

```
<style name="AppTheme" parents="Theme.AppCompat.Light.
    ↪ NoActionBar">
<!-- Ueberschreiben von Theme einstellungen -->
</style>
```

Im Theme definierte Styles können in anderen Ressourcen referenziert werden, anstelle eines wird ein ? verwendet.

```
<style name="AccentedButton">
    <item name="android:background">?colorAccent</item
      ↪ >
</style>
<!-- ... -->
<Button
    android:background="?attr/colorAccent"
    ...
```

Die Aktzenfraben können im Style.xml definiert werden

```
<item name="primary" type="color">#3F51B5</item>
<item name="primary_dark" type="color">#303F9F</item>
<item name="accent" type="color">#FF4081</item>
```

Theme Overlays Die Kombination von Light-Theme und dunkler Toolbar führt dazu, dass die Schrift in der Toolbar schwarz ist. View können keine eigenen Themes haben, aber Theme Overlays, die einen Teil der Theme-Attribute überschreiben.

```
<android.support.v7.widget.Toolbar
    ...
    app:theme="@style/ThemeOverlay.AppCompat.Dark.
      ↪ ActionBar"
    app.popupTheme="@style/ThemeOverlay.AppCompat.
      ↪ Light" />
```

Floating Action Button Der FAB bietet eine primäre Aktion für die darunterliegende View an.

```
<android.support.design.widget.FloatingActionButton
    android:src="@drawable/plus"
    android:onClick="onPlusClicked" />
```

TextInputLayout Dies ist ein EditText, bei dem der Hinweistext beim editieren nicht verschwindet.

```
<android.support.design.widget.TextInputLayout
    android:id="@+id/username_text_input_layout"
    android:layout_width="match_parent"
    android:layout_height="wrap_content
    android:layout_gravity="center" >
    <EditText
        android:id="@+id/username_edit_text"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="Your username"
        android:layout_gravity="center" />
</android.support.design.widget.TextInputLayout>
```

Scrollende Toolbar Die Toolbar kann auch mit dem Inhalt weggescrollt werden.

```
<android.support.design.widget.CoordinatorLayout
      ↪ xmlns:android="..." xmlns:app="..."
    android:layout_width="match_parent"
      ↪ android:layout_height="match_parent">
    <android.support.v7.widget.RecyclerView
        android:id="@+id/recyclerView"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        app:layout_behavior="@string/
      ↪ appbar_scrolling_view_behavior" />
    <android.support.design.widget.AppBarLayout
        android:id="@+id/appBarLayout"
        android:layout_width="match_parent"
        android:layout_height="wrap_content">
    <android.support.v7.widget.Toolbar
        android:id="@+id/toolbar"
        android:layout_width="match_parent"
        android:layout_height="?attr/actionBarSize"
        app:layout_scrollFlags="scroll|enterAlways"/>
</..>
```

Zusammenklappbare Toolbars Toolbars die Verschwinden können.

```
<android.support.design.widget.AppBarLayout
    ...
    android:layout_height="128dp">
    <android.support.design.widget.
      ↪ CollapsingToolbarLayout
        android:id="@+id/collapsing_toolbar"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        app:contentScrim="?attr/colorPrimary"
        app:expandedTitleMarginEnd="64dp"
        app:expandedTitleMarginStart="48dp"
        app:layout_scrollFlags="scroll|
      ↪ exitUntilCollapsed">
    <android.support.v7.widget.Toolbar
        android:id="@+id/toolbar"
        android:layout_width="match_parent"
        android:layout_height="?attr/actionBarSize"
        app:layout_collapseMode="pin" />
```

Tab Navigation Dies ist kein neues Feature, aber mit dem CoordinatorLayout und AppBarLayout integriert worden. Tabs sind unterschiedliche Screens (Fragments), und Swipe oder Touch wechselt zwischen Tabs.

```
<CoordinatorLayout>
    <AppBarLayout>
        <Toolbar />
        <TabLayout />
    </AppBarLayout>
    <ViewPager />
</CoordinatorLayout>
```

Das TabLayout zeigt die Liste der Tabs an, der ViewPager den Inhalt der Tabs (wechselt zwischen Fragments).

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    ...
    ViewPager vp = (...)findViewById(R.di.viewpager);
    ViewPagerAdapger adapter = new ViewPagerAdapger(
      ↪ getSupportFragmentManager());
    adapter.addFragment(new ListFragment(), "CALLS");
    adapter.addFragment(new ListFragment(), "CHATS");
    adapter.addFragment(new ListFragment(); "CONTACTS"
      ↪ );
    vp.setAdapter(adapter);
    TabLayout tl = (...)findViewById(R.id.tabs);
    tl.setupWithViewPager(vp);
```

Diese Toolbar lässt sich beim Scrollen auch verstecken.

```
<android.support.design.widget.CoordinatorLayout ...>
    <android.support.design.widget.AppBarLayout ...>
        <android.support.v7.widget.Toolbar ...
            app:layout_scrollFlags="scroll|enterAlways
      ↪ />
        <android.support.design.widget.TabLayout .../>
    </android.support.design.widget.AppBarLayout>
    <android.support.v4.view.ViewPager ..
        app:layout_behavior="@string/
      ↪ appbar_scrolling_view_behavior" />
</android.support.design.widget.CoordinatorLayout>
```

Dies klappt jedoch nur, wenn die Fragments RecylcerView oder NestedScrollViews sind (nicht mit ListViews oder ScrollViews).

Persistenz

Die Apps werden vom System beendet, ohne dass dies dem Benutzer bewusst ist. Das Sichern der Daten ist also Aufgabe der App. Es gibt zwei unterschiedliche Arten von Daten: Zustandsdaten der Views und Anwendungsdaten der Domain-Klassen.

View-Daten Persistieren

`onCreate` und `onSaveInstanceState` erhalten je ein Bundle Objekt, dies ist wie eine Map/Property List mit assoziierten String-Key Values.

```
public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState)
      ↪ {
        super.onCreate(savedInstanceState);
    }
    @Override
    protected void onSaveInstanceState(Bundle outState
      ↪ ) {
        super.onSaveInstanceState(outState);
    }
}
```

Wichtig: Der `super` Aufruf speichert alle Views die eine ID haben.
Die AppDaten sollten in `onPause` gesichert werden, da `onSaveInstanceState` nicht immer ausgeführt wird. Es gibt verschiedene Möglichkeiten um Daten zu sichern:

- **Shared Preferences:** Key-Value Paare für wenige Daten
- **Files:** Für private Daten die mit anderen Programmen geteilt werden
- **SQLite:** Für strukturierte Daten, die in einer relationalen Datenbank abgelegt werden können
- **Cloud:** Nicht immer verfügbar, lokaler Zwischenspeicher nötig

Shared Preferences

Erlaubt sind Key-Value Paare mit Value vom Typ: `boolean | float | int | long | String | Set<String>`.

```
SharedPreferences settings = getSharedPreferences(
      ↪ PREFS_NAME, MODE_PRIVATE);
SharedPreferences.Editor editor = settings.edit();
editor.putBoolean("disabled", false);
boolean isDisabled = settings.getBoolean("disabled",
      ↪ false);
editor.commit();
```

Eine Art Observer für diese Settings ist der `settings.registerOnSharedPreferenceChangeListener`.

File Storage

Es gibt zwei Speicher-Typen in Android. Den internen (privaten) und den externen Speicher (extern muss nicht zwingend auf einem externen Medium liegen). In den internen Speicher schreibt man fast ganz normal, wie das in Java üblich ist.

```
FileOutputStream fos = openFileOutput(FILENAME,
      ↪ Context.MODE_PRIVATE);
fos.write("File content".getBytes());
fos.close();
```

Möchte man in den externen Speicher schreiben, um beispielsweise Daten anderen Apps zur verfügung zu stellen, braucht man zuerst die Permissions im Manifest zu setzten.

```
File path = Environment.
      ↪ getExternalStoragePublicDirectory(Environment
      ↪ .DIRECTORY_PICTURES);
File file = new File(path, "HSR_Cat.png");
```

Es gibt Konstanten für verschiedene vordefinierte Verzeichnisse.

SQLite Storage

Der SQLite Storage eignet sich für relationale Daten wie Domain Objekte.

```
public class DBHelpter extends SQLiteOpenHelper {
    private static final int DATABASE_VERSION = 2;
    DBHelper(Context context) {
        super(context, DATABASE_NAME, null,
      ↪ DATABASE_VERSION);
    }
    @Override
    public void onCreate(SQLiteDatabase db) {
        db.execSQL("CREATE TABLE ....;");
    }
    @Override
    public void onUpgrade(SQLiteDatabase db, int
      ↪ oldVersion, int newVersion {}
}
// Irgendwo anders
DBHelper helper = new DBHelper(this);
SQLiteDatabase db = helper.getReadableDatabase();
db.execSQL("SELECT * FROM ...");
```

Hintergrunddienste

Da der Main-Thread für das GUI zuständig ist, und die Ereignisse in einer Queue abgearbeitet werden, sollte man keine lange andauernden Aufträge in diesem laufen lassen (Siehe Abschnitt EventHandling).
Eine simple und schnelle Lösung ist die Verwendung von Java Threads. Dies ist in Ordnung für kleinere Aufgaben, welche das GUI nicht manipulieren. Möchte man das UI manipulieren muss man mit Hilfsmethoden wie `post()` arbeiten.

```
public void onClick(View v) {
    Runnable runnable = new Runnable() {
        @Override
        public void run() {
            final Bitmap bitmap = download("something"
      ↪ );
            Runnable command = new Runnable() {
                @Override
                public void run() {
                    imageView.setImageBitmap(bitmap);
                }
            };
            imageView.post(command);
        }
    };
    Thread thread = new Thread(runnable);
    thread.start();
}
```

Mit `post()` übergibt man die `Runnable` Instanz dem UI-Thread und führt diesen aus. Das `command` wird also im UI-Thread ausgeführt.

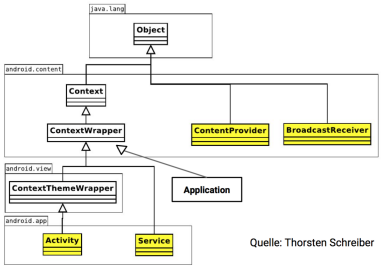
AsyncTask

Typischerweise startet man eine Aufgabe die in einem eigenen Thread ablaufen soll, um am Ende wieder etwas auf den Main-Thread zu tun. Der AsyncTask nimmt Parameter als generische Typen entgegen, für die Aufgabe die man Durchführen will. Der **erste Parameter** ist der Typ des Inputs, der **zweite** der Typ des Feedbacks über den Fortschritt und der **dritte** der Typ des Outputs. Pre = Vor, Post = Nach

```
public class DownloadBitmapTask extends AsyncTask<
      ↪ String, Integer, Bitmap> {
    @Override
    protected void onPreExecute() {
        // UI-Thread
        super.onPreExecute();
    }
    @Override
    protected Bitmap doInBackground(String ... params)
      ↪ {
        // Eigener Thread
        publishProgress(10);
        return download(params[0]);
    }
    @Override
    protected void onPostExecute(Bitmap bitmap){
        //UI-Thread
        imageView.setImageBitmap(bitmap);
    }
    @Override
    protected void onProgressUpdate(Integer... values)
      ↪ {
        //UI-Thread
        progressBar.setProgress(values[0]);
    }
}
```

Möchte man keine Information über den Fortschritt, setzt man den zweiten generischen Parameter auf `Void`.

Komponenten einer Android App



Quelle: Thorsten Schreiber

Context Klasse

Mit einer Context-Instanz kann man:

- Neue View erstellen (Kontext muss als Parameter mitgegeben werden)
- Auf System-Service zugreifen (`context.getSystemService(LAYOUT_INFLATER_SERVICE)`)
- Die Applikationsinstanz erhalten
- Neue Acativities starten (mittels Intent)
- Preferences lesen und schreiben
- Services starten

Services

Für Aufgaben die im Hintergrund ablaufen sollen oder deren Abarbeitung das UI zu lange blockieren würde, verwendet man Services. Diese haben folgende Eigenschaften:

- Kein UI
- Können gestarted oder gebunden werden
- Arbeiten im UI-Thread der Applikation, sind keine eigenen Threads
- Müssen in der Manifest Datei deklariert werden

Deklaration

Wie Activities müssen auch Service-Komponenten im Manifest deklariert werden

```
<application>
    <service
        android:name=".ExampleService"
        android:exported="false" />
</application>
```

Es gibt zwei verschiedene Arten von Services. Den **gebundenen** Service (Client-Server ähnliche Kommunikation über eine längere Zeitdauer) und den **gestarteten** Service (erledigen einmaliger Aufgaben).

Started Services

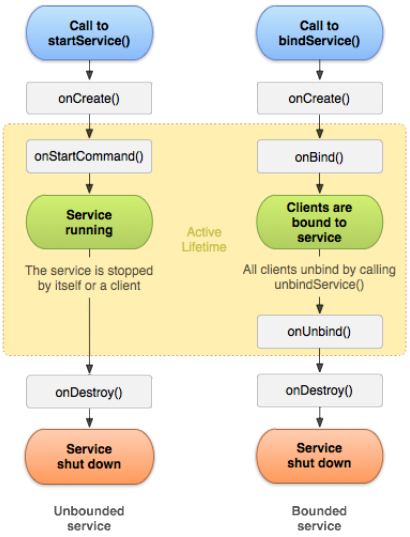
Ein Service wird mit `context.startService(intent)` gestartet. Der Service läuft im Hintergrund und wird nicht gestoppt, auch wenn der Anwender die App wechselt oder der startende Context zerstört wird (im Hintergrund meint nicht in einem eigenen Thread). Der Service soll sich selbst beenden, wenn der User die App wechselt. Grundsätzlich kann man sagen der Service entkoppelt Aufgaben vom Context, und der AsyncTask entkoppelt Aufgaben vom Main-Thread.

Bound Services

Ein Service wird mit `context.bindService(intent, ...)` gebunden. Der Service liefert ein Interface, das dem Client erlaubt, Methoden des Services aufzurufen. Es ist möglich, dass mehrere Clients gebunden sind. Wenn alle Clients `context.unbindService(...)` aufgerufen haben, wird der Service zerstört.

Lifecycle

Die Callbacks eines Services sind ähnlich wie die der Activitiy



Started Service Starten

Der Service wird aus einer Activity gestartet.

```
Intent intent = new Intent(this, SimpleService.class);
this.startService(intent);
```

Dies ruft ggf `onCreate()` und anschliessen `onStartCommand()` im Service auf. Ohne Einstellung, findet keine Kommunikation zwischen Service und Acitivity statt. Der Service kann mit `stopSelf()` gestoppt werden und sollte auch getan werden. Man kann den Service auch in der Acitivity stoppen, mittels `stopService()`.

Started Service IntentService

Der IntentService stellt einen Worker-Thread zur Verfügung. Er verwendet eine Queue, um Intents zwischenzuspeichern, welche sequentiell in `onHandleIntent()` abgearbeitet werden. Der Service wird gestoppt, nachdem alle Intents abgearbeitet wurden.

```
public class HelloIntentService extends IntentService
      ↪ {
    public HelloIntentService() {
        super("HelloIntentService");
    }
    @Override
    protected void onHandleIntent(Intent intent) { }
}
```

Der Service enthält keine Referenz an die aufrufende Activity. Um nun ein Resultat vom Service zu bekommen, kann man entweder einen Intent mit Resultat als Broadcast versenden, wo dann wiederum die Activity einen Broadcast-Receiver zur Verfügung stellen muss um den Intent zu empfangen. Oder man verwendet einen. `PendingIntent`.

Pending Intent ist ein Objekt das einen Intent umhüllt (wrapped). Genauer ist es ein Token das man einer "Fremden App" übergiebt (e.g. `NotificationManager`, `AlarmManager` Home Screen `AppWidgetManager` oder 3rd Party Apps), welch die "Fremde App"berechtigt bestimmten Code auszuführen.
Wenn man der "fremden App" einen Intent übergibt, und diese App diesen Intent sendet/broadcastet, dann wird dieser Intent mit seinen eigenen Berechtigungen ausgeführt. Wenn man aber der "fremden Appëinen PendingIntent übergibt werden die Berechtigungen der übergebenden App verwendet. Wird die Klasse, welche den PendingIntent erzeugt terminiert, bleibt der PendingIntent bestehen und kann von anderen Apps weiterverwendet werden. Intents behandelen spezifische Komponenten der App, genauso PendingIntents und nutzen dafür z.B.
`(PendingIntent.getActivity()`
`PendingIntent.getBroadcast()`
`PendingIntent.getService()`
Bsp. 1: Intent erzeugen und wrappen, ausführen der Operation zugehörend zum PendingIntent (`send()`)

```
// Explicit intent to wrap
Intent intent = new Intent(this, LoginActivity.class);

// Create pending intent and wrap our intent
PendingIntent pendingIntent = PendingIntent.
    ↪ getActivity(this, 1, intent, PendingIntent.
    ↪ FLAG_CANCEL_CURRENT);
try {
    // Perform the operation associated with our
      ↪ pendingIntent
    pendingIntent.send();
} catch (PendingIntent.CanceledException e) {
    e.printStackTrace();
}
```

Besseres Bsp. 2: Erzeugen, wrappen, starten mit AlarmManager nach 3 sec.

```
int seconds = 2;
// Create an intent that will be wrapped in
    ↪ PendingIntent
Intent intent = new Intent(this, MyReceiver.class);

// Create the pending intent and wrap our intent
PendingIntent pendingIntent = PendingIntent.
    ↪ getBroadcast(this, 1, intent, 0);

// Get the alarm manager service and schedule it to go
    ↪  off after 3s
AlarmManager alarmManager = (AlarmManager)
    ↪ getSystemService(ALARM_SERVICE);
alarmManager.set(AlarmManager.RTC_WAKEUP, System.
    ↪ currentTimeMillis() + (seconds * 1000),
    ↪ pendingIntent);

Toast.makeText(this, "Alarm set in " + seconds + "
    ↪ seconds", Toast.LENGTH_LONG).show();
```

Wenn man noch eine BroadcastReceiver onReceive() Methode hinzufügt, vibriert das Gerät für 2 sec. sobald ein Broadcast Event gesendet wurde.

```
@Override
public void onReceive(Context context, Intent intent)
    ↪ {
    // Vibrate for 2 seconds
    Vibrator vibrator = (Vibrator) context.
      ↪ getSystemService(Context.VIBRATOR_SERVICE);
    vibrator.vibrate(2000);
}
```

Bound Service Der Bound Service funktioniert nach einem Client-Server Modell. Clients können Methoden eines Service aufrufen und die App kann die Funktionalität des Services andern Apps zur verfügugn stellen. Beim Aufruf von `BindService()` erhält der Client ein Interface um mit dem Service zu kommunizieren. Es ist möglich, dass mehrere Clients gleichzeitig gebunden sind. Nachdem alle Clients `unbindService()` aufgerufen haben, wird der Service beendet.

Bound Service: Server Der Server muss von der Service Klasse und ein Binder implementieren.

```
public class LocalService extends Service {
    private final IBinder binder = new LocalBinder();
    public class LocalBinder extends Binder {
        LocalService getService() {
            return LoaclService.this;
        }
    }
    @Override
    public IBinder onBind(Intent intent) {
        return binder;
    }
    public int getRandomNumber() {
        return new Random.nextInt(100);
    }
}
```

Bound Service: Client Ein Bound-Service liefert bei `onBind()` ein Interface vom Typ `IBinder`. Wenn der Service nur innerhalb der App und im gleichen Prozess arbeitet, sollte ein Binder als Interface geliefert werden. Clients rufen so die Methoden des Binders oder des Services auf.

```
Intent intent = new Intent(this, LocalService.class);
bindService(intent, connection, Context.
    ↪ BIND_AUTO_CREATE);
```

Der Aufruf von `bindService()` liefert nicht den `onBind()` Binder, sondern der Callback `onServiceConnected()` wird aufgerufen.

```
private ServiceConnection c = new ServiceConnection()
    ↪ {
    @Override
    public void onServiceConnected(ComponentName
      ↪ className, IBinder binder) {
        LocalService.LocalBinder myBinder = (...)
      ↪ binder;
        LocalService myService = myBinder.getService()
      ↪ ;
        int random = myService.getRandomNumer();
    }
    @Override
    public void onServiceDisconeccted(ComponentName
      ↪ name) {}
};
```

Bound Service: Messenger Ein Messenger kann eingesetzt werden um prozessübergreifend Informationen auszutauschen. Der Service stellt dabei einen Handler zur Verfügung um Meldungen zu erhalten, welche er sequentiell abarbeitet. Für Rückgabewerte des Servers muss der Client seinerseits einen Messenger zur Verfügung stellen.

Broadcast Receiver
Broadcasts sind Meldungen die als Intent verschickt werden. Die Action bestimmt den Ereignistyp. Eine Meldung enthält Informationen über ein bestimmtes Ereignis (SMS empfangen, System booted, Akku schwach etc.). Ein Broadcast Receiver kann registriert werden, um bestimmte Meldungen zu erhalten. Standartmeldungen des Systems umfassen unter anderem: `ACTION_BOOT_COMPLETED` | `ACTION_POWER_CONNECTED` | `ACTION_POWER_DISCONNECTED` | `ACTION_BATTERY_LOW` | `ACTION_BATTERY_OKAY`.

Broadcast Receiver: Registrieren Man kann den Receiver entweder statisch in der Manifest Datei, dann ist der Receiver global in der App verfügbar (an keine Activity gebunden) und wird bei jeder Meldung neu instanziert. Dynamisch macht man das mit `context.registerReceiver()`. Dann wird der Receiver an die Activity gebunden und muss abgemeldet werden, wenn er nicht mehr benötigt wird.

```
<receiver android:name=".TimeChangedReceiver" >
    <intent-filter>
        <action android:name="android.intent.action.
      ↪ TIME_SET" />
    </intent-filter>
</receiver>
```

Listing 2: "Statische Registration"

```
LocalBroadcastManager lbm = LocalBroadcastManager.
    ↪ getInstance(getApplicationContext());
IntentFilter filter = new IntentFilter(Intent.
    ↪ ACTION_BOOT_COMPLETED);
MyBroadcastReceiver receiver = new MyBroadcastReceiver
    ↪ (this);
lbm.registerReceiver(receiver, filter);
```

Listing 3: "Dynamische Registration"

```
LocalBroadcastManager lbm = LocalBroadcastManager.
    ↪ getInstance(getApplicationContext());
lbm.unregisterReceiver(receiver);
```

Listing 4: "Receiver abmelden"

Broadcast Receiver: Implementation Die Implementation eines Receivers muss von der Klasse `BroadcastReceiver` ableiten. Die Methode `onReceive()` muss überschrieben werden um Meldungen zu erhalten und verarbeiten.

```
private class MyBroadcastReceiver extends
    ↪ BroadcastReceiver {
    public MyBroadcastReceiver(MainActivity activity)
      ↪ { }
    @Override
    public void onReceive(Context context, Intent
      ↪ intent){ }
}
```

Broadcast Receiver: Versenden Der Broadcast Receiver kann auch Meldungen versenden (semantik?). Dies kann man mit der Methode `context.sendBroadcast(intent)`. Die Action im angegeben Intent definiert den Ereignistyp (Namensgebend und in der Action ist global, bei eigenen Actions, Name mit eigenem Namespace verwenden). Beim Intent können unter Extras zusätzliche Daten hinterlegt werden. Wenn man Meldungen nur im eigenen selben Prozess versenden möchte, sollte man den `LocalBroadcastManger verwenden`. So verlassen die Meldungen den Prozess nicht und andere Prozesse können unserer App keine Meldungen senden.

Content Provider

Der Content Provider stellt Daten prozessübergreifend zur Verfügung. Es ist ein Client-Server Modell. Der Provider Client und Provider kommunizieren über eine standartisierte Schnittstelle um Daten auszutauschen. Android stellt beispielsweise Kalender oder Kontakte über einen Content-Provider zur Verfügung.

Schnittstelle eines Content Provider stellt in der Regel Daten zur Verfügung die in einer Tabelle in einer Datenbank abgelegt sind. Deshalb ähnelt die Schnittstelle stark der SQL Syntax.

Security Der Content Provider kann Permissions setzte um den Zugriff auf die Daten einzuschränken. Die Clients müssen die erforderlichen Permissions im Manifest angeben.

```
<uses-permission android:name="android.permission.
    ↪ READ_USER_DICTIONARY" />
```

Content Provider: Client Der ContentResolver stellt die Schnittstelle zur Verfügung. Die Tabelle des Content-Provider wird als URI angegeben: `content://user_dictionary/words`.

```
Cursor cursor = getContentResolver().query(
    UserDictionary.Words.CONTENT_URI,
    mProjection,
    mSelectionClause,
    mSelectionArgs,
    mSortOrder);
int index = cursor.getColumnIndex(UserDictionary.Words
    ↪ .WORD);
while(cursor.moveToNext()){
    String word = cursor.getString(index);
}
```

Content Provider: Server Der Einsatz eines eigenen Servers ist erforderlich wenn andere Apps strukturierte Daten oder Dateien zur Verfügung stehen sollen, oder wenn andere Apps die Möglichkeit haben sollen, Daten zu durchsuchen. Ein Provider kann Daten als Dateien oder strukturiert anbieten. Dateien werden im Internal Storage abgelegt, der Client erhält einen Handle um auf eine Datei zuzugreifen. Strukturierte Daten werden in der Regel in einer Datenbank abgelegt. Der ContentResolver hat gezeigt, dass das Prinzip von Tabellen mit Spalten und Reihen verwendet werden sollte. **Ein Content-Provider muss Thread-Save programmiert werden**.
Bei der Implementation muss man sich folgendes überlegen:

- Wie sollen Daten persistent gespeichert werden
- Die Content-URI muss festgelegt werden
- Die Klasse `ContentProvider` muss implementiert werden.
- Eine Contract Klasse muss implementiert werden. Die Klasse enthält alle Bezeichner (Konstanten) die ein Client benötigt. Die Klasse muss zudem in den Applikationscode des Clients kopiert werden.
- Permissions in der Manifest Datei müssen festgelegt werden

Weiterführende Themen

Sensoren

Bewegungs-, Umgebuns- und Lagesensoren.

Sensor	Android 4.0 (API Level 14)	Android 2.3 (API Level 9)
TYPE_ACCELEROMETER	Yes	Yes
TYPE_AMBIENT_TEMPERATURE	Yes	n/a
TYPE_GRAVITY	Yes	Yes
TYPE_GYROSCOPE	Yes	Yes
TYPE_LIGHT	Yes	Yes
TYPE_LINEAR_ACCELERATION	Yes	Yes
TYPE_MAGNETIC_FIELD	Yes	Yes
TYPE_ORIENTATION	Yes[2]	Yes[2]
TYPE_PRESSURE	Yes	Yes
TYPE_PROXIMITY	Yes	Yes
TYPE_RELATIVE_HUMIDITY	Yes	n/a
TYPE_ROTATION_VECTOR	Yes	Yes
TYPE_TEMPERATURE	Yes[2]	Yes

Für die Arbeit mit einem Sensor brauchen wir:

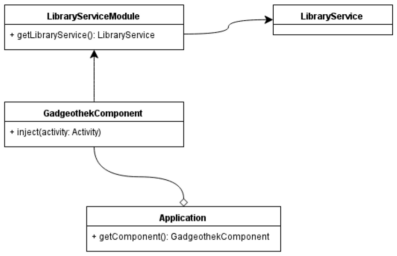
- `SensorManager` als Einstiegspunkt für die Sensoren
- `Sensor` als Repräsentant für einen konkreten Sensor
- `SensorEventListener` um Updates von Sensoren zu registrieren
- `SensorEvent` um die Sensordaten auszulesen

Ein Sensor liefert uns nur Rohdaten! Je nach Sensor unterschiedlich zu interpretieren. Sensor Bsp.:

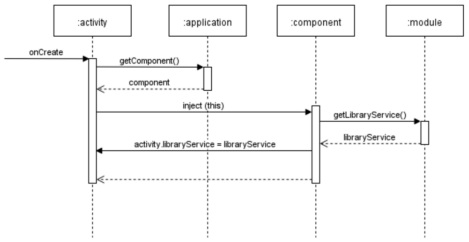
```
public class MainActivity extends AppCompatActivity
      ↪ implements SensorEventListener {
   private TextView textView;
   private SensorManager sensorManager;
   private Sensor lightSensor;
   @Override
   protected void onCreate(Bundle savedInstanceState)
      ↪ {
       ...
       textView = (TextView) findViewById(R.id.
      ↪ textView);
       sensorManager = (SensorManager)
      ↪ getSystemService(Context.SENSOR_SERVICE);
       lightSensor = sensorManager.getSensorList(
      ↪ Sensor.TYPE_LIGHT).get(0);
   }
   @Override
   protected void onResume() {
       super.onResume();
       sensorManager.registerListener(this,
      ↪ lightSensor, SensorManager.
      ↪ SENSOR_DELAY_NORMAL);
   }
   @Override
   protected void onPause() {
       super.onPause();
       sensorManager.unregisterListener(this);
   }
   @Override
   public void onSensorChanged(SensorEvent event) {
       textView.setText(String.format("Helligkeit: %.0
      ↪ f", event.values[0]));
   }
}
```

View Injection

Flexible Lösung wäre Service ausserhalb der Activity zu erstellen und dieser übergeben. Noch besser: Interface von LibraryService extrahieren und im Test durch einen Fake-Server ersetzen. Die Klasse erstellt seine Dependencies also nicht selbst, stattdessen werden diese injiziert (injected).
Einfache Lösung: Konstruktor mit Parametern und final-Attributen. Stellt sicher das Klasse vollständig initialisiert wurde.
Weniger Schön: Setter Methoden. Benutzer muss aufpassen das alle aufruft.
Dagger Tool verwenden: **Modul** instanziert unsere Klasse(z.B. LibraryService) die wir injecten wollen. **Komponente** fasst Module zusammen und ist zuständig für die Injection. Eine Klasse lässt sich über die Komponenten ihre Abhängigkeiten



injecten.




```
@Module
public class LibraryServiceModule {
   @Provides
   @Singleton
   public LibraryService getLibraryService() {
       return new LibraryService();
   }
}


@Singleton
@Component(modules = {LibraryServiceModule.class})
public interface GadgeothekComponent {
   void inject(GadgeothekActivity activity);
}
```

```
public class Application extends android.app.
      ↪ Application {
   GadgeothekComponent component;

   public void onCreate() {
       component = DaggerGadgeothekComponent.builder()
      ↪ .build();
   }

   public GadgeothekComponent getComponent() {  return
      ↪  component;  }
}


public class GadgeothekActivity extends
      ↪ AppCompatActivity {
   @Inject
   LibraryService libraryService;

   @Override
   protected void onCreate(Bundle savedInstanceState)
      ↪ {
   ...
   ((Application) getApplication()).getComponent().
      ↪ inject(this);
```

Lohnt sich Dependency Injection?
Vorteile:

- keine statischen Methoden mehr
- zentrale Konfiguration in Modul und Komponente
- Einfache Testbarkeit: Modul oder Applikation austauschen

Nachteile:

- Nicht unbedingt weniger Schreibaufwand bei kleinen Projekten
- Braucht Tests: bei einer Fehlkonfiguration drohen NullPointerException
- Spaghetticode?

Data Binding

- erspart Tipparbeit
- in XML direkt auf Objekte zugreifen (Bsp. `onClick`)
- Optimal: GUI aktualisiert sich selbst, sobald Objekt sich ändert
  - XML-Layout als Observer
  - Einfache Logik (x ? y : z) direkt im XML
- noch Beta

```
public class User {
   public String firstName;
   public String lastName;

   public User(String firstName, String lastName) {
       this.firstName = firstName;
       this.lastName = lastName;
   }
}
```

```
<layout xmlns:android="http://schemas.android.com/apk/
      ↪ res/android">
   <data >
       <variable name="user" type="ch.hsr.mge.
      ↪ databindingdemo.User"/>
   </data>
   <RelativeLayout
       ... >

       <TextView android:text="@{user.firstName}" ...
      ↪ />

       <TextView android:text="@{user.lastName}"  ...
      ↪ />
   </RelativeLayout>
</layout>
```

Expression Language unterstützt fast alle Java Expressions (Keine Statements Deklarationen, Loops, kein new, this und super)

```
android:text="@{String.valueOf(index + 1)}"
android:visibility="@{age < 13 ? View.GONE : View.
      ↪ VISIBLE}"
android:transitionName='@{"image_" + id}'
```

```
public class MainActivity extends AppCompatActivity {

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

       ActivityMainBinding binding =
           DataBindingUtil.setContentView(this, R.
      ↪ layout.activity_main);

       User user = new User("Mirko", "Stocker");
       binding.setUser(user);
   }
}
```

Neben Properties können auch Events gebunden werden

```
<Button
   android:text="Save"
   android:onClick="@{controller.onButtonSaveClicked}"
      ↪ />
   <EditText
   android:text="@{user.lastName}"
   android:addTextChangedListener="@{user.
      ↪ lastNameWatcher}" />
```

Was wenn gebundene Objekt sich ändert?

```
public class User {
   public String lastName;
   public boolean isDirty;

   public TextWatcher lastNameWatcher = new
      ↪ TextWatcher() {
       @Override
       public void beforeTextChanged(CharSequence s,
      ↪ int start, int count, int after) { }

       @Override
       public void onTextChanged(CharSequence s, int
      ↪ start, int before, int count) {
           lastName = s.toString();
       }

       @Override public void afterTextChanged(Editable
      ↪  s) { }
   };
}
```

Um Daten aus gebundenen Objekten in der View zu aktualisieren benötigen wir wieder das Observer-Pattern. **View** observiert das gebundene Objekt. **Objekt** ist observable. Um unser Objekt observable zu machen müssen wir `ObservableFields` verwenden.

```
public class User {
   public ObservableField<String> firstName = new
      ↪ ObservableField<>();
   public ObservableField<String> lastName  = new
      ↪ ObservableField<>();

   public TextWatcher lastNameWatcher = new
      ↪ TextWatcher() {
   ...
           if (!Objects.equals(lastName.get(), s.
      ↪ toString())) {
               lastName.set(s.toString());
           }
   }
};
```