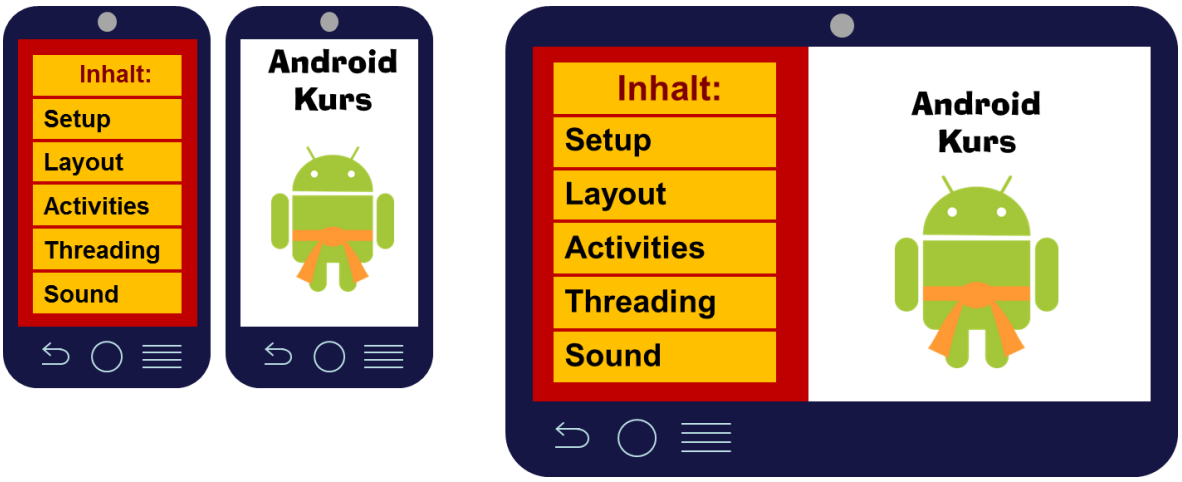
Inhaltsverzeichnis

1 Fragments

Mit dem Einzug von Tablettrechnern in die Android Welt stieß das Konzept der einfachen Activities schnell an seine Grenzen. Das Problem ist, dass der Bildschirm eines Tabletts wesentlich größer ist als der eines Handheld Devices. Applikationen, welche auf die kleineren Handhelds ausgerichtet sind wirken auf einem Tablettrechner optisch falsch designed, da der vergleichsweise üppige Platz nicht wirklich ausgenutzt wird.

Werden beim Handheld im Regelfall die verschiedenen Dialoge sequenziell abgearbeitet, so können sie bei Tabletts parallel angezeigt werden:



Die Lösung für dieses Problem heißt “Fragments”. Fragments sind Teile einer Activity, welche unabhängig voneinander und unabhängig vom Layout geladen und mit Funktionalität bestückt werden können. Sie „leben“ innerhalb von Activities und bilden somit die Grundlage für ein dynamisches Layout, welches sich auf die Besonderheiten des Anzeigegerätes anpassen kann.

2 Der Life Cycle eines Fragments

Die Kontrolle über den Life Cycle hat zwar wie bei Activities das Android System, jedoch ist der Fragment Life Cycle sehr eng mit dem Life Cycle der Activity verbunden:

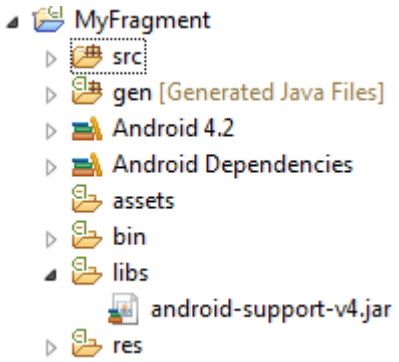
Activity Status:	Fragment Stat.:	Fragment Callbacks:
Created	Created	onAttach()
		onCreate()
		onCreateView()
		onActivityCreated()
Started	Started	onStart()
Resumed	Resumed	onResume()
Paused	Paused	onPause()
Stoped	Stoped	onStop()
Destroyed	Destroyed	onDestroyView()
		onDestroy()
		onDetach()

Wie ihr seht, stimmen die Statuswerte des Fragments mit dem der Activity überein. Alle Callback Methoden welche wir bei den Activities kennengelernt haben, gibt es auch bei den Fragments – hier sind aber noch zusätzlich ein paar Callbacks mehr aufgetaucht:

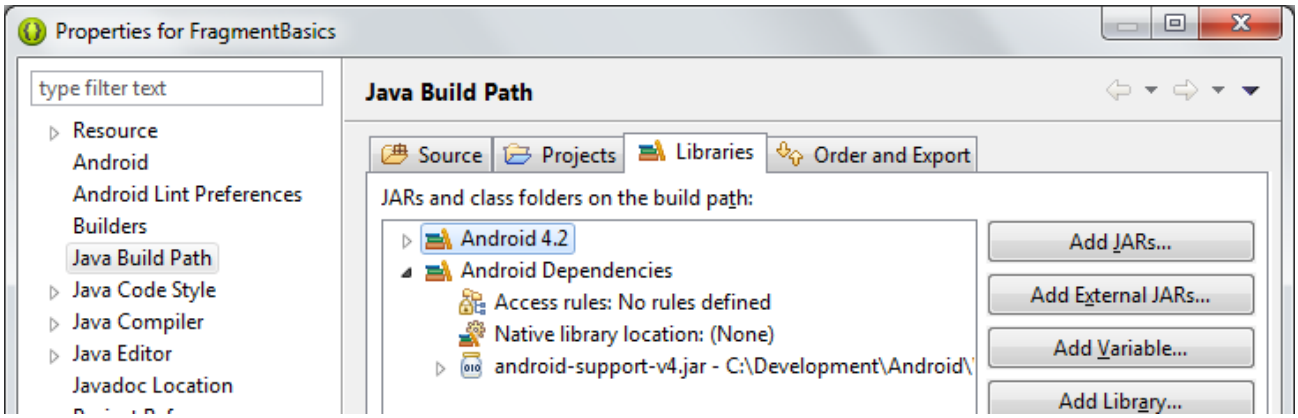
Methode:	Wird aufgerufen wenn:	Was soll passieren:
onAttach()	Das Fragment wurde an die Activity gehängt.	Wird u.A. genutzt um die aufrufende Activity als Callback Handler zu referenzieren.
onCreateView()	Die Viewstruktur des aufgebaut wurde in der das Fragment eingebettet wird.	Wird überschrieben, da wir hier das Layout des Fragments über den LayoutInflater erzeugen lassen.
onActivityCreated()	Die Activity hat die Methode onCreate() durchlaufen.	Wird in der Regel nicht überschrieben. Lediglich, wenn Rückgriffe auf die Activity notwendig sind, könnte hier Logik notwendig sein.
onDestroyView()	Wenn der View, in dem das Fragment eingebettet ist entfernt wird, kommt diese Methode zum Einsatz.	Hier können Ressourcen freigegeben werden.
onDetach()	Wird kurz vor der Trennung von der Activity aufgerufen.	Ebenfalls für die Ressourcenfreigabe.

3 Die Supportlibrary oder das Problem mit der Version

Nachdem das Konzept der Fragments erst mit dem API Level 11 eingeführt wurde, wir aber mitunter für ältere Android Versionen auch die Fragments realisieren wollen, haben sich die Android Entwickler etwas überlegt. Sie naben eine „Support Library“ geschaffen, in der alle wichtigen Fragment Funktionen eingebettet sind. Wir müssen uns somit als Entwickler entscheiden, ob wir das API Level auf >= 11 setzen und alle älteren Android Versionen ausklammern, oder ob wir mit der Support Library arbeiten. Diese funktioniert bei allen Android Versionen ab Level 4. Um die Library zu nutzen, muss sie zuerst installiert werden. Hierzu öffnen wir den Android SDK Manager und suchen den Eintrag „Android Support Library“. Der wird angeklickt und dann auf „Install Packages“ geklickt (eventuelle Lizenzinformationen müssen vorher akzeptiert werden).



In einem eventuellen Projekt muss die Library in dem Projekt referenziert sein und einem Ordner „libs“ enthalten sein und in. Wenn nicht, muss dies manuell erfolgen, indem wir mit einem Rechtsklick auf dem Projekt die Properties aufzeigen und im Tab „Libraries“ das android-support-v4.jar einfügen (mit dem Button „Add External JARs…“:

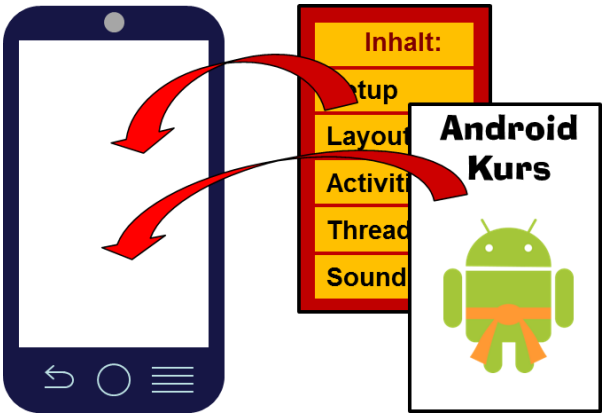


Das Jar File findet ihr in eurem Android SDK Ordner und dort unter /extras/android/support/v4/android

Grundsätzlich bleibt es euch überlassen, ob ihr euer Projekt mit einer minimalen API von 11 aufsetzt, oder darunter liegt und die SupportLibrary nutzt. Die API ist für beide Optionen fast identisch. Die Unterschiede für das später besprochene Beispiel sind, dass die Imports von bspw. android.support.v4.app.Fragment auf android.app.Fragment geändert werden müssen, die Methode getSupportFragmentManager() auf getFragmentManager() geändert und anstatt FragmentActivity nun die Activity direkt genommen werden kann.

4 Aufbau der Layouts

Das Layout mit Hilfe von Fragments wird zwar vom Aufrufkonzept komplett anders realisiert wie Apps, welche die unterschiedlichen Inhalte mit Hilfe von Activities aufbaut – die gute Nachricht ist jedoch, dass das wesentliche Verhalten der Views und Layouts, so wie wir sie im Gelbgurt kennen gelernt haben, sich nicht ändert. Ein paar Änderungen müssen wir zwar in Kauf nehmen, aber das ist marginal.



Der grundsätzliche Unterschied liegt erst mal auf der Hand – wenn wir einen anderen Bildschirminhalt anzeigen lassen wollen, dann rufen wir keine neue Activity auf, sondern wir tauschen das Fragment aus. Das hat den Vorteil, dass die Activity als zentrale Anlaufstelle für Logik fungieren kann, was den Datenaustausch zwischen den Fragments erheblich erleichtert. Im Kapitel „Die Kommunikation innerhalb der Activity“ werden wir näher darauf eingehen.

Fragments benötigen immer einen Container , was nichts anderes ist als eine View, in der das Fragment residiert. Grundsätzlich unterscheiden wir zwischen zwei Arten, Fragments in ein Layout einer Activity einzufügen – statisch und dynamisch. Beginnen wir mit der statischen Einbindung. Hierbei wird in einem beliebigen Layout (in unserem Beispiel verwenden wir ein FrameLayout) ein „fragment“ eingeführt (Achtung – es wird tatsächlich klein geschrieben), welches wie ein normales Viewelement konfiguriert wird. Lediglich das „name“ Attribut ist neu. Hier wird die Klasse eingetragen, welche beim Laden des Layouts instanziiert wird – in unserem Fall ControlFragment:

```
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
 android:id="@+id/my_fragment_container"
 android:layout_width="match_parent"
 android:layout_height="match_parent">
  <fragment android:name="com.example.myfragment.ControlFragment"
   android:id="@+id/control_fragment"
   android:layout_width="match_parent"
   android:layout_height="match_parent" />
</FrameLayout>
```

Sehen wir uns den wesentlichen Code der Klasse, welche eine Ableitung von Fragment ist, genauer an:

```
public class ControlFragment extends Fragment {
  @Override
  public View onCreateView(LayoutInflater inflater, ViewGroup container,
      Bundle savedInstanceState) {
    if (savedInstanceState != null) {
      // restore internal values if necessary
    }
    View myView = inflater.inflate(R.layout.control_view,
        container, false);
    return myView;
  }
  @Override
  public void onSaveInstanceState(Bundle outState) {
    super.onSaveInstanceState(outState);
    // Save internal values if necessary
  }
}
```

Die Struktur unterscheidet sich nicht wesentlich von der einer Activity. Lediglich die onCreateView Methode ist hier wichtig – sie wird ja aufgerufen, wenn die Activity sich instanziiert hat. Hier schreiben wir den Code rein, welcher das Layout realisiert, mit Hilfe eines Layout Inflaters, der praktischerweise bereits als Parameter geliefert wird. Die erzeugte View wird dann an den Aufrufer zurückgegeben, der die View dann in die Activity einbauen kann.

Das Layout des Fragments, welcher durch den Layout Inflater instanziiert werden soll, liegt in einem Ressourcenfile „control_view.xml“. Dies ist wiederum ein ganz normales Layoutfile, welches die notwendigen Viewelemente beinhaltet:

```
<?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="vertical" >
  <EditText android:id="@+id/editControlText"
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:layout_gravity="center_horizontal"
   android:inputType="textNoSuggestions"/>
  <Button android:id="@+id/sendControlText"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center_horizontal"
    android:text="@string/showSendControlButtonText"/>
</LinearLayout>
```

Wenn wir nun dieses Fragment in ein anderes Layout einhängen wollen, dann können wir dies ohne großen Aufwand umsetzen, indem wir ein neues Layoutfile erzeugen, welches im Rahmen einer Activity instanziiert wird. Wie ihr seht habe ich im untenstehenden File das „fragment“ Element in ein Linear Layout eingefügt:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
 android:orientation="horizontal"
 android:layout_width="match_parent"
 android:layout_height="match_parent">
  <fragment android:name="com.example.myfragment.ControlFragment"
   android:id="@+id/control_fragment"
   android:layout_weight="1"
   android:layout_width="0dp"
   android:layout_height="match_parent" />
  <FrameLayout
   android:id="@+id/my_content_fragment_container"
   android:layout_weight="2"
   android:layout_width="0dp"
   android:layout_height="match_parent" />
</LinearLayout>
```

Wichtig ist nun noch, dass wir für den Fall, dass wir die Support Library nutzen eine Activity, welche Fragments einbinden muss nicht eine Ableitung der Klasse „Activity“ ist, sondern eine Ableitung der Klasse „FragmentActivty“ sein muss:

```
public class MainActivity extends FragmentActivity
```

Dies ist deshalb notwendig, weil die Activity nun mit diversen internen Funktionalitäten mit den Fragments kommunizieren muss, was bei einer klassischen Acitivty bis API Level 10 noch nicht vorgesehen war. Wenn wir das minimale API Level auf 11 gestellt haben und die „normalen“ API Funktionen nutzen, dann nutzen wir wie gehabt die Klasse „Acitvity“, da hier die Fragmentfunktionen nun eingebaut wurden.

Soweit ist der Aufwand für die statische Einbindung von Fragments überschaubar. Der zentrale Nachteil dieser Vorgehensweise ist, dass wir hier nicht in der Lage sind, das Fragment auszutauschen. Vor allem in unserem obigen Beispiel, in dem wir das Fragment in ein FrameLayout eingebunden haben wird sich nachteilig erweisen, da wir keine Chance haben das Fragment zu tauschen. Hier müssen wir einen anderen Ansatz wählen – das dynamische Einbinden von Fragments. Sehen wir uns hierzu wieder das Layout File der Activity an:´

```
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
 android:id="@+id/my_fragment_container"
 android:layout_width="match_parent"
 android:layout_height="match_parent"/>
```

Was hier fehlt ist das Fragment – wir wollen es ja auch dynamisch (soll heißen im Java Code) einbinden. Wie ihr seht, habe ich dem FrameLayout eine ID verpasst, damit wir im Java Code das Layout eindeutig identifizieren können. Unten seht ihr den Code der onCreate Methode unserer MainActivity:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.activity_main);
  if (savedInstanceState != null) {
    return;
  }
  ControlFragment myFragment = new ControlFragment();
  getSupportFragmentManager().beginTransaction().add(
    R.id.my_fragment_container, myFragment).commit();
}
```

Zwei Sachen an diesem Code bedürfen wohl einer Erklärung. Zuerst wird mit dem savedInstanceState Objekt geprüft, ob die Activity bereits instanziiert war. Wenn ja, dann darf das Fragment nicht neu hinzugefügt werden, da wir sonst zwei Fragments übereinander zu liegen haben:

```
  if (savedInstanceState != null) {
    return;
  }
```

Anschließend wird das Fragment instanziiert und mittels einer Transaktion in die Activity eingebaut. Hierbei wird der Methode „add“ mitgeteilt, welche ID das Layout hat, in der das Fragment eingehängt werden soll (R.id.my_fragment_container) und welches Fragment einzuhängen ist (myFragment). Warum es über Transaktionen eingehängt wird werden wir im nächsten Kapitel näher beleuchten.

Wenn wir nun das Fragment austauschen wollen, dann erfolgt dies wie folgt. Zuerst erzeugen wir ein neues Fragment – hier der Kasse ContentFragmentA (wir werden später das ganze Beispiel im Code nochmal durchgehen):

```
ContentFragmentA newFragment = new ContentFragmentA();
```

Als nächstes erzeugen wir eine Transaktion, mit deren Hilfe wir das Fragment austauschen wollen:

```
FragmentTransaction transaction =
 getSupportFragmentManager().beginTransaction();
```

Nun tauschen wir das Fragment im Layout mit der gegebenen ID durch unser neues Fragment aus und schließen die Transaktion mit commit ab.

```
transaction.replace(R.id.my_fragment_container, newFragment);
transaction.commit();
```

Dieser Code hängt genau an der Stelle, an der die Aktion für den Austausch (bspw. ein Button wurde angeklickt) getriggert wurde.

5 OnClickListener

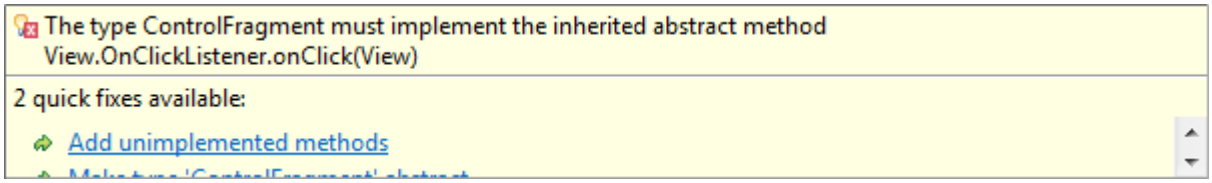
Es gilt noch eine Besonderheit von Fragments anzusprechen. Im Gelbgurt haben wir uns den Button angesehen und festgestellt, dass wir lediglich im Layout die Methode hinterlegen müssen, welche beim Anklicken des Buttons aufgerufen werden soll (bspw. android:onClick="myButtonClicked"). Die Methode haben wir in dem Code der Activity eingetragen. Nun haben wir aber das Problem, dass das Layout, wo der Button eingefügt wird im Zweifelsfall ein Fragment ist. Der Aufruf von „myButtonClicked" wird aber in der Activity aufschlagen und nicht in der Instanz des Fragments. In der Activity hat aber streng genommen die Methode nichts zu suchen, da sie ja ein Teil des Fragments ist. Wir können also in diesem Zusammenhang das „onClick" Attribut des Buttons nicht mehr ohne weiteres nutzen.

Eine Alternative bietet der OnClickListener. Die Listener werden wir in der nächsten Gurtfarbe näher ansehen, aber vorab sprechen wir hier die Funktionalitäten an, welche wir in diesem Rahmen benötigen. Ein Listener ist eine Anlaufstelle für Ereignisse, welche im System passieren. Ein Klick auf einen Button, eine ImageView etc. löst ein Ereignis im System aus, welches durch sogenannte Listener erkannt werden können. Programmierer, welche sich bereits mit anderen GUI APIs wie bspw. SWING auseinander gesetzt haben kennen das Konzept.

In Android müssen wir nun im Fragment einen solchen Listener einbauen, damit das Fragmentobjekt mitkriegt, wann auf den Button geklickt wird. Dies geschieht über die Implementierung des Interfaces OnClickListener in die Fragmentklasse:

```
public class ControlFragment extends Fragment implements OnClickListener
```

Wenn ihr nun den Listener implementiert, dann meckert Eclipse sofort, dass nicht alle notwendigen Methoden in der Klasse ControlFragment implementiert wurden:



Wir klicken dann einfach auf „Add unimplemented methods" und Eclipse fügt somit die „onClick" Methode ein. Diese Methode wird immer dann aufgerufen, wenn auf irgendwas in dem Fragment geklickt wurde. Wir können somit anhand der ID der Quelle des Ereignisses (also der ID des Views auf den geklickt wurde) beliebige Aktionen durchführen:

```
@Override
public void onClick(View myView) {
  if (myView.getId() == R.id.sendControlText) {
    // Do something meaningful
  }
}
```

Wie wir sehen, liefert uns der Parameter der onClick Methode die View, welche das Ereignis ausgelöst hat. Wenn wir nur einen oder zwei Elemente haben, auf die wir „hören" müssen, so können wir die IDs mit einer if – Abfrage prüfen, ansonsten empfehle ich eine switch Anweisung. Was jetzt noch fehlt ist, dem Button den Listener zuzuordnen.

Dazu ergänzen wir die onCreateView Methode wie folgt:

```
Button btnSendText = (Button) myView.findViewById(R.id.sendControlText);
btnSendText.setOnClickListener(this);
```

Nachdem wir unsere gesamte ControlFragment Klasse nun zu einem Listener gemacht haben, können wir diese nun unserem Button als OnClickListener zuordnen.

6 Die FragmentTransaction

In den Codezeilen zur Erzeugung bzw. zum Austausch von Fragments haben wir gesehen, dass wir dies in einer Transatktion durchführen. Ein Grund hierfür ist der „Back Button" des Android Gerätes. Wenn wir eine Transaktion starten, dann können wir die einzelnen Aktionen in den „BackStack" schreiben, womit die Betätigung des Back Buttons diese Transaktion wieder rückgängig macht – das ersetzte Fragment also wieder zurückgerufen wird. Sehen wir uns das im Code nochmal an, indem wir den Tausch nochmal leicht abändern:

```
ContentFragmentA newFragment = new ContentFragmentA();
FragmentTransaction transaction =
 getSupportFragmentManager().beginTransaction();
transaction.replace(R.id.my_fragment_container, newFragment);
transaction.addToBackStack(null);
transaction.commit();
```

Mit der folgenden Zeile wird die gesamte Transaktion in den BackStack geschrieben:

```
transaction.addToBackStack(null);
```

Die Aktion wird in der Regel kurz vor dem commit durchgeführt, wodurch wir mehrere Aktionen in einer Transaktion zusammenfassen können und als Gesamtes mittels dem Back Button wieder rückgängig machen können. Der Parameterwert „null" ist für uns erst mal ausreichend. Wenn wir uns später in unserer Android Karriere mal näher mit den Möglichkeiten der Transaktion auseinander setzten, dann werden wir diesen Wert anderweitig nutzen.

Wenn wir diesen call nicht durchführen, so wird der Back Button auf der Activity Ebene zurückgehen, also im Falle der MainActivity wird diese dann beendet.

Ein Wort noch zum FragmentManager. Wenn wir mit der SupportLibrary arbeiten, so müssen wir die Methode „getSupportFragmentManager()" aufrufen, nutzen wir die API Level 11 oder höher, so rufen wir getFragmentManager() auf.
Der Fragment Manager implementiert die Kontrolle der Fragments und somit auch die FragmentTransaction.

7 Die Kommunikation innerhalb der Activity

Wie bereits erwähnt wurde, ist ein wesentlicher Unterschied zwischen einem Design welches sich darauf beschränkt verschiedene Bildschirmaufbauten mit verschiedenen Activities zu realisieren und einem Design basierend auf Fragments, dass die Fragments immer im Rahmen einer Activity laufen. Diese also immer als zentraler Ansprechpunkt für Datenaustausch fungieren kann. Dies machen wir für unsere interne Activitykommunikation zunutze. Sehen wir uns das im Detail an.

7.1 Datenübermittlung mittels Bundle

Beginnen wir mit der Instanziierung eines Fragments. Hier haben wir, wie bei den Intents auch, die Möglichkeit Werte in ein Bundle zu schreiben und dieses dem Fragment zuzuordnen:

```
ContentFragmentA newFragment = new ContentFragmentA();
Bundle args = new Bundle();
args.putString(ContentFragmentA.FRAG_MESSAGE_DEF_A, messageString);
```

```
newFragment.setArguments(args);
FragmentTransaction transaction =
 getSupportFragmentManager().beginTransaction();
transaction.replace(R.id.my_content_fragment_container, newFragment);
transaction.addToBackStack(null);
transaction.commit();
```

Auch hier ist ein eindeutiger Name im Bundle vorzusehen, welcher später zur Identifikation des Wertes herangezogen werden kann. In der Fragment Klasse wird der Wert wie folgt herausgelesen:

```
@Override
public void onStart() {
  super.onStart();
  String textValue = "";
  Bundle args = getArguments();
  if (args != null) {
    textValue =  args.getString(FRAG_MESSAGE_DEF_A);
  }
  setTextValue(textValue);
}
```

Die "getArguments" Methode liefert das Bundle zurück, welches bei der Erzeugung des Fragments hinterlegt wurde. Das Auslesen erfolgt genauso wie beim Intent. Diese Methodik ist vor allem sinnvoll, wenn wir gleich bei der Instanziierung des Fragments Daten bereitstellen möchten.

7.2 Aufruf einer public Methode

Die Activity kann jederzeit eine Referenz auf das Fragment Objekt ermitteln. Wird diese in eine entsprechende Klassenvariable geschrieben, so können eventuelle öffentliche Methoden aufgerufen werden. Gehen wir davon aus, dass in der Klasse ContentFragmentA folgende Methode existiert:

```
public void setTextValue(String newValue){
  Activity myActivity = getActivity();
  if (myActivity != null)
    ((TextView) myActivity.findViewById(
          R.id.fragASentText)).setText(newValue);
}
```

Die Methode bekommt also einen String als Parameter, welcher in einen TextView übernommen wird. Bevor wir uns den passenden Code in der Activity ansehen, noch ein paar Hinweise zum oben dargestellten Code. Jedes Fragment kann mit der Methode „getActivity()" eine Referenz auf die Activity erlangen, in der das Fragment residiert. Es gibt allerdings Situationen, in denen diese Methode „null" liefert (bspw. wenn die Activity gerade aufgelöst wird). Um in diesen Situationen nicht auf eine NullPointerException zu laufen, sollte vorher geprüft werden, ob wirklich eine saubere Referenz zurückgegeben wurde – sie wird somit auf ungleich null geprüft. Erst dann kann auf die üblichen Activity Methoden, wie bspw. das Finden eines View Elements zurückgegriffen werden.

Doch nun zum Code in der Activity:

```
Fragment contentFragment = getSupportFragmentManager().findFragmentById(
    R.id.my_content_fragment_container);
if ((contentFragment != null)
  && (findViewById(R.id.linearLayoutA) != null)) {
    ((ContentFragmentA)contentFragment).setTextValue(optionalData);
}
```

Hier wird zuerst eine Referenz auf das Fragment abgerufen. Dies erfolgt über den getSupportFragmentManager, bzw. ab API Level 11 ohne Support Library über die Methode getFragmentManager. Da jedes Fragment in einem Layoutcontainer eingebettet sein muss, kann das Fragment mit der entsprechenden ID des Layouts (in unserem Beispiel „R.id.my_content_fragment_container“) identifiziert werden. Auch hier empfiehlt es sich zu prüfen, ob wir eine saubere Referenz erhalten haben (dies wird umso wichtiger, wenn wir mit einfachen (one pane) und doppelten (two pane) Bildschirmaufteilungen arbeiten. Darauf werden wir im nächsten Kapitel tiefer eingehen.
Nun können wir nach einem Typecast direkt auf die public Klassenmethoden unseres Fragments zugreifen und die entsprechenden Werte setzen.
Diese Methodik ist sinnvoll, wenn wir im two pane Modus sind und Daten von einem sichtbaren Fragment zu einem anderen sichtbaren Fragment senden möchten.

7.3 Kommunikation mittels Callback Interface

Wie wir oben gesehen haben, können wir im Fragment eine Referenz auf die zugrundeliegende Activity abrufen. Theoretisch könnten wir an dieser Stelle auch einen Typecast auf die tatsächlich von der Activity (bzw. FragmentActivity) abgeleitete, von uns erzeugte Klasse durchführen und wie im vorausgegangenen Kapitel eine public Methode in unserer Activity realisieren und diese für den Datenaustausch vom Fragment zur Acitivty nutzen. Nachdem wir dadurch aber die beiden Klassen sehr eng verbinden und eine Abhängigkeit des Fragments zur Activity erzeugen, ist das ein nicht zu empfehlender Weg. Viel besser ist es, sich eines Interfaces zu bedienen. Folgender Code im Fragment soll dies verdeutlichen:

```
private OnContentAButtonClickedListener myCallback;
public interface OnContentAButtonClickedListener {
  public void onContentAButtonClicked(String optionalData);
}
```

Es wird dem Fragment ein Interface zugeordnet, welches von der Activity implementiert werden kann. Somit können wir von dem Fragment aus auf eine beliebige Activity zugreifen, sofern sie das Interface implementiert. Weiterhin wird eine Variable deklariert (mCallback), welche eine Referenz auf die zugrundeliegende Activity erhalten soll. Dies erfolgt idealerweise in der onAttach() Methode, da hier ja die Activity das Fragment an sich bindet. Zugleich bekommen wir hier auch gleich die Activity als Parameter geliefert.

```
@Override
public void onAttach(Activity activity) {
  super.onAttach(activity);
  try {
    myCallback = (OnContentAButtonClickedListener) activity;
  } catch (ClassCastException e) {
    throw new ClassCastException(activity.toString()
      + " must implement OnContentAButtonClickedListener");
  }
}
```

Wir versuchen hier also einen Classcast auf unser Interface, welches ja von der Activity implementiert werden soll. Wenn dieser Cast fehlschlägt, dann hat die Activity das Interface nicht implementiert und wir informieren den Entwickler über eine Exception darüber. Wenn der Classcast funktioniert, haben wir in der Variable mCallback eine Referenz auf die zugrundeliegende Activity und können die Methoden des Interfaces (in unserem Beispiel „nur“ onContentAButtonClicked()) nutzen – bspw. in unserer onClick Methode:

```
myCallback.onContentAButtonClicked(messageContent);
```

Doch bevor dies funktioniert, müssen wir in unserer Activity das Interface erst mal implementieren:

```
public class MainActivity extends FragmentActivity implements
  ContentFragmentA.OnContentAButtonClickedListener
{
```

```
  @Override
  public void onContentAButtonClicked(String optionalData)      {
      // Verarbeitung des Klicks
  }
  // weitere Implementierung unserer Activity
```

Nachdem man mehrere Interfaces implementieren kann, ist es möglich einen Kommunikationskanal für jedes Fragment, welches durch die Acitivty instanziiert werden kann, zu schaffen.

8 One pane vs. two pane Layout

Der hauptsächliche Sinn der Fragments war ja die Flexibilisierung unseres Layout Designs vor dem Hintergrund der unterschiedlicher Bildschirmgrößen von Handheld und Tablett Devices. An dieser Stelle werden wir uns überlegen, wie wir diese beiden verschiedenen Situationen in unserer Activity und den Fragments handeln können.

Gehen wir von folgenden beiden Layouts für unsere MainActivity aus:

Im Layout Ordner platzieren wir ein Layoutfile (Namens activity_main.xml) für den Standardfall – was das normale Handheld Device, also „Handy“ sein soll:

```
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/my_fragment_container"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />
```

Für den Fall, dass wir ein Tablett finden, dann könnten wir einen weiteren Layout Ordner erstellen, namens layout-large-land, um für große Bildschirme (640dp x 480dp oder größer) in horizontaler Ausrichtung ein eigenes Layout zur Verfügung zu stellen. Alternativ kann auch ein Ordner layout-w640dp erzeugt werden, um Bildschirme zu unterstützen, welche mindestens 640dp in der Breite (egal ob horizontal oder vertikal gehalten) aufweisen. Da dies erst ab Android 3.2 (API Level 13) funktioniert, sollten eventuell noch Alternativen vorgehalten werden. Wir werden für unser Beispiel erst mal auf layout-large-land zurückgreifen. Dort schreiben wir auch ein activity_main.xml File hinein, mit folgendem Inhalt:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
 android:orientation="horizontal"
 android:layout_width="match_parent"
 android:layout_height="match_parent">
  <fragment android:name="com.example.myfragment.ControlFragment"
   android:id="@+id/control_fragment"
   android:layout_weight="1"
   android:layout_width="0dp"
    android:layout_height="match_parent" />
  <FrameLayout
   android:id="@+id/my_content_fragment_container"
   android:layout_weight="2"
   android:layout_width="0dp"
   android:layout_height="match_parent" />
</LinearLayout>
```

Wie ihr seht, habe ich hier zwei Bereiche definiert – einmal ein statisches Fragment, welches eine Art Kontollelement darstellt und ein dynamisch ladbares Fragment, welches spatter ausgetauscht werden kann. Dieses packe ich einfach in ein FrameLayout rein.

Wenn wir jetzt in der MainActivity die einzelnen Fragments instanziieren, dann müssen wir wissen, ob wir gerade in einem Modus sind, in dem wir das Layout mit zwei, oder einem Fragment anzeigen. Leider kann das manchmal recht schwierig sein. Gehen wir zuerst mal vom einfachen Fall aus – wir wollen unterschei-

den, ob wir die App in einem Handheld oder in einem Tablett laufen lassen. Dies können wir mit folgender einfachen Abfrage prüfen:

```
if (getSupportFragmentManager().findFragmentById(
  R.id.my_content_fragment_container) == null) {
  // one pane mode
} else {
  // two pane mode
}
```

Der Gedanke hinter diesem Code ist, dass bei einem Handheld das Gerät immer das erste (Standard) Layout instanziieren wird, dort der my_content_fragment_container aber nicht vorhanden ist. Starten wir die App mit einem Tablett so wird das zweite Layout instanziiert, der FragmentManager kann somit diesen Fragment Container nun finden. Man kann auch auf ein Fragment prüfen, welches im Standard Layout abgelegt wurde (also my_fragment_container) – wobei man hier auf ungleich null prüfen müsste.

Nun zum schwierigen Teil. Wir haben unser zweites Layout nun nicht in den Ordner layout-large, sondern in den Ordner layour-large-land eingetragen. Das bedeutet nun dass wir in eine Situation kommen könnten, in der wir die App in horizontaler Haltung starten, sie dann aber in die Vertikale ausrichten, wodurch wir nun im one pane Modus landen würden. Das Problem dabei ist, dass es vorkommen kann, dass der FragmentManager das Element mit der ID my_content_fragment_container noch finden kann – wir haben somit keine Möglichkeit mehr den Modus zu verifizieren. Wir müssten somit auch prüfen, ob wir das Gerät derzeit vertikal halten. Ein funktionierender Code würde somit wie folgt aussehen:

```
private boolean inTwoPaneMode()
{
  if (getResources().getConfiguration().orientation ==
    Configuration.ORIENTATION_PORTRAIT) {
    return false;
  }
  return (getSupportFragmentManager().findFragmentById(
    R.id.my_content_fragment_container) != null);
}
```

Wenn das Gerät vertikal gehalten wird (also Portait), dann können wir grundsätzlich vom one pane Modus ausgehen. Wenn nicht, dann prüfen wir, ob wir den my_content_fragment_container finden können – der wird nur dann existieren, wenn wir das Gerät horizontal halten und der Bildschirm als „large“ klassifiziert wurde.

Noch schwieriger wird es, wenn wir das Layout nicht auf Basis der horizontalen bzw. vertikalen Ausrichtung bestimmen, sondern auf Basis der minimal zur Verfügung stehenden Breite. In solchen Fällen müssen wir auf Werte wie getConfiguration().screenWidthDp oder getConfiguration().screenLayout abprüfen. Für’s Erste soll aber der oben genannte Code vorerst ausreichend sein.

9 Ein konkretes Beispiel

Um nun alles oben genannte sinnvoll unter einen Hut zu bringen, habe ich mir folgende App überlegt (zugegeben, sie ist nicht wirklich sinnvoll einsetzbar, aber sie zeigt fast alles, was wir im Zuge der Fragments beachten müssen. Wir haben drei verschiedene Fragments, welche alle in einer Activity residieren. Das erste Fragment wird das ControlFragment sein, welches Daten zu den beiden anderen Fragments senden kann. Es beinhaltet drei Buttons, „Show A“, „Show B“ und „Update“. Weiterhin existiert ein EditText View.

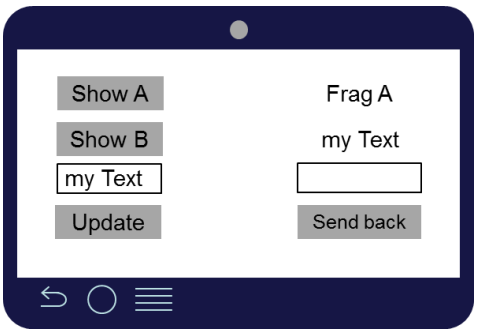
Die beiden anderen Fragments sind identisch aufgebaut, mit zwei TextViews, einem Edit-

Text View und einem Button "Send back". Diese sind in den Fragments ContentFragment A und Content-FragmentB untergebracht.

Folgende Funktion soll im one pane Modus realisert werden:

Beim Start der App wird das ControlFragment sichtbar sein. Ein Klick auf „Show A" sendet den Text aus dem EditText View an das ContentFragmentA, wobei dieses nun anstatt dem ControlFragment platziert werden muss. Gleiches gilt für „Show B" – aber eben auf das ContentFragment B. Ein Klick auf „Update" wird im one pane Modus keinerlei Änderungen mit sich bringen. Der Text wird mit dem Präfix „new" in dem mittleren TextView angezeigt.

Ist das ContentFragmentA bzw. ContentFragmentB sichtbar, dann wird der Text in dem jeweiligen EditText View wieder zurück an das ControlFragment gesendet (wobei dieser nun wieder den Platz in der Activity einnimmt) und dieser Text dann im EditText Feld erscheint.



Im two pane Modus wird die Funktionalität etwas abgeändert. Beim Start der App sind zwei Fragments sichtbar – das ControlFragment und das ContentFragmentA. Wenn nun im ControlFragment der Button „Show A" gedrückt wird, so wird der EditText Inhalt in das ContentFragmentA geschrieben, jetzt allerdings mit dem Präfix „change". Sollte jedoch ContentFragmentB rechts sichtbar sein, so wird vorher das Fragment ausgetauscht, so dass ContentFragmentA erscheint. Umgekehrt gilt gleiches für ContentFragmentB. Wenn in einer der beiden ContentFragments wiederum auf den „Send back" Button geklickt wird, so wird der Text direkt in das EditText Element des ControlFragments geschrieben. Wird im ControlFragment der „Update" Button geklickt, so wird im aktuell sichtbaren ContentFragment auf der rechten Seite der Text einfach ausgetauscht.

Ich habe das Beispiel desshalb so gewählt, weil wir hier folgendes Verhalten sehen können:

- Statisches instanziieren eines Fragments
- Dynamisches instanziieren eines Fragments
- Dynamischer Austausch von Fragmenten
- Datenübertragung beim instanziieren von Fragmenten
- Datenaustausch zwischen den Fragmenten
- One pane und two pane Modus
- Wechsel vom one pane Modus in den two pane Modus innerhalb einer Activity und umgekehrt

Beginnen wir mit den Layouts.

9.1 Die Layout Files

9.1.1 activity_main.xml

Da das Layout der MainActivity je nach genutzten Device unterschiedlich sein muss, ist es notwendig zwei verschiedene Realisierungen von **activity_main.xml** anzufertigen. Beginnen wir mit dem Standardfall – dies ist der one pane Modus und wird dementsprechend auch im Ordner **layout** abgelegt:

```
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/my_fragment_container"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />
```

Für den two pane Modus benötigen wir im Ordner **layout-large-land** die zweite Version unseres Files:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
 android:orientation="horizontal"
 android:layout_width="match_parent"
```

```
 android:layout_height="match_parent">
    <fragment android:name="com.example.myfragment.ControlFragment"
     android:id="@+id/control_fragment"
     android:layout_weight="1"
     android:layout_width="0dp"
     android:layout_height="match_parent" />
    <FrameLayout
     android:id="@+id/my_content_fragment_container"
     android:layout_weight="2"
     android:layout_width="0dp"
     android:layout_height="match_parent" />
</LinearLayout>
```

Es wurde dergestalt angelegt, dass wir das ControlFragment statisch einbinden und die ContentFragments dynamisch realisieren können.

9.1.2 control_view.xml

Als nächstes nehmen wir uns die Layoutdefinitionen der Fragments vor. Beginnen wir mit dem ControlFragment:

```
<?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="vertical" >
    <Button android:id="@+id/showFragA"
     android:layout_width="wrap_content"
     android:layout_height="wrap_content"
     android:layout_gravity="center_horizontal"
     android:text="@string/showFragAButtonText"/>
    <Button android:id="@+id/showFragB"
     android:layout_width="wrap_content"
     android:layout_height="wrap_content"
     android:layout_gravity="center_horizontal"
     android:text="@string/showFragBButtonText"/>
    <EditText android:id="@+id/editControlText"
     android:layout_width="wrap_content"
     android:layout_height="wrap_content"
     android:layout_gravity="center_horizontal"
     android:inputType="textNoSuggestions"/>
    <Button android:id="@+id/sendControlText"
     android:layout_width="wrap_content"
     android:layout_height="wrap_content"
     android:layout_gravity="center_horizontal"
     android:text="@string/showSendControlButtonText"/>
</LinearLayout>
```

Die Struktur ist relative einfach gehalten. Es wurde lediglich ein vertikales LinearLayout mit den einzelnen Views darin designed. Wichtig ist – wie immer – dass alle Views eine eindeutige ID aufweisen.

9.1.3 content_view_a.xml und content_view_b.xml

Die beiden Layouts sind absolut identisch aufgebaut – lediglich das Kürzel A des Files **content_view_a.xml** wurde jeweils als Kürzel B im **content_view_b.xml** File ersetzt. Deshalb wird an dieser Stelle nur das File **content_view_a.xml** dargestellt:

```
<?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="vertical"
 android:id="@+id/linearLayoutA">
    <TextView android:id="@+id/fragAText"
     android:layout_width="wrap_content"
     android:layout_height="wrap_content"
     android:layout_gravity="center_horizontal"
     android:text="@string/fragAText"/>
    <TextView android:id="@+id/fragASentText"
     android:layout_width="wrap_content"
     android:layout_height="wrap_content"
     android:layout_gravity="center_horizontal"
     android:text="@string/emptyText"/>
    <EditText android:id="@+id/editContentAText"
     android:layout_width="wrap_content"
     android:layout_height="wrap_content"
     android:layout_gravity="center_horizontal"
     android:inputType="textNoSuggestions"/>
    <Button android:id="@+id/sendContentAText"
     android:layout_width="wrap_content"
     android:layout_height="wrap_content"
     android:layout_gravity="center_horizontal"
     android:text="@string/showSendContentButtonText"/>
</LinearLayout>
```

Auch dieses Layout wurde recht minimalistisch aufgesetzt. Wenn ihr hier aufregendere Sachen machen wollt, könnt ihr das gerne tun – lediglich die ID Werte sollten gleich bleiben, damit meine Codebeispiele auch nach wie vor funktionieren.

9.1.4 strings.xml

Lediglich der Vollständigkeit halber füge ich hier noch das **strings.xml** File mit an.

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="app_name">MyFragment</string>
    <string name="showFragAButtonText">Show A</string>
    <string name="showFragBButtonText">Show B</string>
    <string name="showSendControlButtonText">Update</string>
    <string name="showSendContentButtonText">Send back</string>
    <string name="fragBText">Frag B</string>
    <string name="fragAText">Frag A</string>
    <string name="emptyText">-</string>
    <string name="menu_settings">Settings</string>
    <string name="prefix_one_pane">new</string>
    <string name="prefix_two_pane">change</string>
</resources>
```

9.2 Die Java Klassen

Die hier gemachtgen Beispielklassen wurden mit Hilfe der Support Library erstellt. Wie weiter Oben schon erwähnt, kann der Code ohne große Probleme in die standard API ab Level 11 umgewandelt werden, indem lediglich die Imports von android.support.v4.app.* auf android.app.* umgewandelt werden und darüber hinaus noch die Klassen FragmentActivity in Activity und die Methode getSupportFragmentManager() auf getFragmentManager() geändert werden.

9.2.1 ControlFragment.java

Auch wenn die MainActivity von der Hierarchie her ganz Oben steht, fangen wir trotzdem mit den Fragments an, da sie im Entwicklungsprozess vor der Activity zu designen sind. Zuerst werfen wir einen Blick auf das Control Fragment – hier erstmal die Package und Import Statements:

```
package com.example.myfragment;

import android.app.Activity;
import android.os.Bundle;
import android.support.v4.app.Fragment;
import android.view.LayoutInflater;
import android.view.View;
import android.view.View.OnClickListener;
import android.view.ViewGroup;
import android.widget.Button;
import android.widget.EditText;
```

Als nächstes kommt die Klassendefinition mitsamt des OnClickListeners – der ja Notwendig war, da die onClick Eigenschaft im Layout File nicht auf das Fragment, sondern auf die Activity abzielt:

```
public class ControlFragment extends Fragment implements OnClickListener
{
```

Weiterhin deklarieren wir eine Konstante, welche die eindeutige Nachrichtenkennung für unseren Bundle Wert darstellt.

```
public static String FRAG_MESSAGE_DEF_C =
 "com.example.myfragment.CALL_FRAGMENT_C";
```

Nun wird das Interface eingebaut, welches die MainActivity implementieren muss um Nachrichten an das Fragment senden zu können. Die Details dieser Methode werde ich im Kapitel MainActivity.java beschreiben.

```
private OnControlButtonClickedListener mCallback;
    public interface OnControlButtonClickedListener {
         public void onControlButtonClicked(int buttonNo,
          String optionalData);
     }
```

Nun wird die CreateView Methode überschrieben. Da wir keine internen Variablen haben, welche im InstanceState gesichert werden müssen, müssen wir aus dem Bundle auch nichts lesen. Ich habe den Code trotzdem drin gelassen, um zu indizieren, wo man die Werte auslesen solltef:

```
    @Override
    public View onCreateView(LayoutInflater inflater,
          ViewGroup container, Bundle savedInstanceState) {
         if (savedInstanceState != null) {
             // restore internal values
         }
```

Als nächstes wird die View mittels Layout Inflater erzeugt. Diese muss am Ende der Methode als Rückgabewert geliefert werden – vorher müssen wir aber noch die drei Buttons finden und den OnClickListener registrieren, damit wir Klicks auf die Buttons in unserer onClick Methode verarbeiten können:

```
View myView = inflater.inflate(R.layout.control_view, container, false);
Button btnShowFragA = (Button) myView.findViewById(R.id.showFragA);
```

```
btnShowFragA.setOnClickListener(this);
Button btnShowFragB = (Button) myView.findViewById(R.id.showFragB);
btnShowFragB.setOnClickListener(this);
Button btnSendText = (Button) myView.findViewById(R.id.sendControlText);
btnSendText.setOnClickListener(this);
return myView;
}
```

Die nächste überschriebene Methode ist onStart. Hier prüfen wir erst, ob bei der Instanziierung des Fragments irgendwelche Informationen ins Bundle geschrieben wurden. Wenn ja, dann prüfen wir zuerst, ob wir eine Activity finden und schreiben den Bundle Wert in den EditText View mit der ID „editControlText“ hinein:

```
@Override
public void onStart() {
    super.onStart();
    Bundle args = getArguments();
    if (args != null) {
        Activity myActivity = getActivity();
        if (myActivity != null)
          ((EditText) myActivity.findViewById(
          R.id.editControlText)).setText(
          args.getString(FRAG_MESSAGE_DEF_C));
    }
}
```

Die Methode “onSaveInstanceState” habe ich einfach drin gelassen um zu zeigen, wo eventuelle Wertpersistierungen durchgeführt werden sollen:

```
@Override
public void onSaveInstanceState(Bundle outState) {
    super.onSaveInstanceState(outState);
    // Save internal values if necessary
}
```

Wie in den oberen Kapiteln beschrieben, sollten wir in der onAttach Methode prüfen, ob die zugrundeliegende Activity auch tatsächlich das Interface zur Datenübermittlung implementiert hat:

```
@Override
public void onAttach(Activity activity) {
    super.onAttach(activity);
    try {
        mCallback = (OnControlButtonClickedListener) activity;
    } catch (ClassCastException e) {
        throw new ClassCastException(activity.toString()
          + " must implement OnControlButtonClickedListener");
    }
}
```

In der onClick Methode wird nun lediglich auf Basis der ID des aufrufenden Buttons die passende Methode aufgerufen. Da die zentrale Verarbeitung in der Activity läuft, wird hier lediglich die Methode der Activity aufgerufen, welche ja über das am Anfang definierte Interface verfügen muss.

```
@Override
public void onClick(View myView) {
    Activity myActivity = getActivity();
    if (myActivity != null) {
```

```
        String myMessage = ((EditText) myActivity.findViewById(
                R.id.editControlText)).getText().toString();
        if ((myView.getId() == R.id.showFragA) ||
             (myView.getId() == R.id.showFragB) ||
             (myView.getId() == R.id. sendControlText))
            mCallback.onControlButtonClicked(myView.getId(), myMessage);
    }
}
```

9.2.2 ContentFragmentA.java und ContentFragmentB.java

Wei bei den Layouts werden wir bei den Java Klassen von ContentFragmentA und ContentFragmentB nur ContentFragmentA beschreiben – ihr müsst bei der eigenen Implementierung dann lediglich für Content-FragmentB alle A Indizes mit B ersetzen. Beginnen wir mit den Imports – auch hier wieder die Nutzung der Support Library:

```
package com.example.myfragment;

import android.app.Activity;
import android.os.Bundle;
import android.support.v4.app.Fragment;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.view.View.OnClickListener;
import android.widget.Button;
import android.widget.EditText;
import android.widget.TextView;
```

Wie bei dem ControlFragment deklarieren wir einen eindeutigen Key für die Bundleinformation.

```
public class ContentFragmentA extends Fragment implements OnClickListener
{
    public static String FRAG_MESSAGE_DEF_A =
     "com.example.myfragment.CALL_FRAGMENT_A";
```

Auch sehen wir hier ein Interface für die Spätere Kommunikation mit der Acitivity vor. Die Details dieser Methode werde ich im Kapitel MainActivity.java beschreiben.

```
private OnContentAButtonClickedListener mCallback;
public interface OnContentAButtonClickedListener {
    public void onContentAButtonClicked(int buttonID,
     String optionalData);
}
```

Nun benötigt der ContentFragmentA auch ein Layout. Gegebenenfalls müssen hier Instanzvariablen nach dem Wiederherstellen aus dem Bundle neu beschrieben werden – da wir in unserem Beispiel aber keine Instanzvariablen haben, benötigen wir dies hier nicht.

```
@Override
public View onCreateView(LayoutInflater inflater,
    ViewGroup container, Bundle savedInstanceState) {
    View myView = inflater.inflate(R.layout.content_view_a,
     container, false);
    Button btnShowFragA =
     (Button) myView.findViewById(R.id.sendContentAText);
    btnShowFragA.setOnClickListener(this);
```

```
    if (savedInstanceState != null) {
            // restore internal values - here not needed
    }
    return myView;
}
```

Die onStart Methode wird von uns genutzt, um in den TextView die Informationen aus dem ControlFragment zu schreiben. Grundsätzlich gibt es zwei Situationen, in denen onStart aufgerufen werden kann – entweder im one pane oder im two pane Mode, wobei hier nur, wenn vorher ContentPaneB auf der rechten Seite sichtbar war. Wenn ContentPaneA sichtbar ist, dann müssen wir nur den Text austauschen – wir brauchen das Fragment nicht neu zu erzeugen. Hier werden wir nun die Forderung umsetzen, dass im one pane Mode der Präfix „new" ergänzen und im two pane Mode den Präfix „change".

```
@Override
public void onStart() {
    super.onStart();
    String textValue = "";
    // check if in two pane view mode
    if (getFragmentManager().findFragmentById(R.id.control_fragment)
     != null) {
        textValue = getResources().getString(R.string.prefix_two_pane);
    } else {
        textValue = getResources().getString(R.string.prefix_one_pane);
    }
```

Jetzt, da wir den Präfix festgelegt haben, müssen wir nur noch aus dem Bundle die Information aus dem ControlPane lesen und im String ergänzen. Die Methode setTextValue wird weiter unten erklärt – hier habe ich nur die Identifikation und beschreiben des TextViews eingebaut:

```
    Bundle args = getArguments();
    if (args != null) {
        textValue += " " + args.getString(FRAG_MESSAGE_DEF_A);
    }
    setTextValue(textValue);
}
```

Auch hier lasse ich die onSaveInstanceState Methode drin – wenn ihr Klassenvariablen habt, sollten die hier in das Bundle reingeschrieben werden.

```
@Override
public void onSaveInstanceState(Bundle outState) {
    super.onSaveInstanceState(outState);
    // Save internal values if necessary
}
```

Der nächste Schritt ist wieder identisch mit dem ControlFragment – wir holen uns eine Referenz auf die zugrundeliegende Activity und versuchen den Typecast – schlägt der fehl, dann müssen wir eine Exception werfen, da wir darauf angewiesen sind, dass die Klasse der Activity unser Interface implementiert.

```
@Override
public void onAttach(Activity activity) {
    super.onAttach(activity);
    try {
        mCallback = (OnContentAButtonClickedListener) activity;
    } catch (ClassCastException e) {
        throw new ClassCastException(activity.toString()
          + " must implement OnContentAButtonClickedListener");
```

```
    }
}
```

Diese Methode ist dafür da, in en TextView mit der ID "fragASentText" einen neuen Wert einzutragen. Wir verwenden diese Methode aus der eigenen Klasse heraus (siehe Methode „onStart"), aber wir werden diese Methode auch aus der Activity im two pane Modus nutzen. Aus diesem Grunde habe ich sie public gesetzt. Wie wir weiter Oben gelernt haben, können wir uns nicht darauf verlassen, dass mit der Methode „getActivity()" immer die Activity gefunden wird, da sie gerade in einem falschen Zustand ist. Insofern prüfen wir zuerst, ob die Activity wirklich nicht gleich null ist. Der Vollständigkeit halber machen wir die gleiche Prüfung, wenn der TextView nicht gefunden werden sollte:

```
public void setTextValue(String newValue)
{
    Activity myActivity = getActivity();
    if (myActivity != null) {
        TextView myTextView =
         (TextView) myActivity.findViewById(R.id.fragASentText);
        if (myTextView != null)
            myTextView.setText(newValue);
    }
}
```

Die onClick Methode, welche ja beim Betätigen unseres „Send back" Buttons aufgerufen wird, prüfen wir zuerst, ob das Event tatsächlich von unserem Button kommt. Wenn ja, dann ermitteln wir den Text aus unserem EditText View und schreiben ihn in den „messageContent" String hinein. Nun können wir in der Activity die Methode „onContentAButtonClicked" aufrufen – da wir ja in onAttach sichergestellt haben, dass die Activity unser Interface implementiert hat.

```
@Override
public void onClick(View myView) {
    if (myView.getId() == R.id.sendContentAText) {
        Activity myActivity = getActivity();
        if (myActivity != null) {
            String messageContent = ((EditText)
            myActivity.findViewById(
            R.id.editContentAText)).getText().toString();
            mCallback.onContentAButtonClicked(R.id.sendContentAText,
             messageContent);
        }
    }
}
```

9.2.3 MainActivity.java

Der größte Brocken ist die MainActivity. Hier laufen alle Fäden zusammen – insofern gilt es hier auch, den meisten Code zu implementieren. Beginne wir mit den obligatorischen Imports (wieder mit der Support Library):

```
package com.example.myfragment;

import android.content.res.Configuration;
import android.os.Bundle;
import android.support.v4.app.Fragment;
import android.support.v4.app.FragmentActivity;
import android.support.v4.app.FragmentTransaction;
import android.view.Menu;
import android.widget.EditText;
```

Anschließend wird die Klasse definiert. Wir vererben aus der FragmentAcitivity (da wir ja mit den Support Library Klassen arbeiten – wenn wir mit der nativen API ab Level 11 arbeiten würden, dann würde hier nur Activity stehen). Weiterhin implementieren wir die Interfaces, welche wir in unseren drei Fragmentklassen eingebaut haben (also in der ControlFragment, ContentFragmentA und ContentFragmentB Klasse):

```
public class MainActivity extends FragmentActivity implements
      ControlFragment.OnControlButtonClickedListener,
      ContentFragmentA.OnContentAButtonClickedListener,
      ContentFragmentB.OnContentBButtonClickedListener
{
```

Die zu implementierenden Klassen werden von Eclipse automatisch eingefügt. Wir wollen bei den beiden ContentFragment Klassen (A und B) ja Daten von dem ContentFragment in Richtung ControlFragment senden (wir erinnern uns – wir klicken in dem ContenFragment auf den „Send back“ Button und der Textinhalt wird in das EditText Element vom ControlFragment geschickt). Da die Codeausführung bei beiden ContentFragments identisch ist, packen wir die eigentliche Funktionalität in eine extra Methode, welche wir von beiden überschriebenen Methoden aufrufen. Ich habe sie mit einer buttonID ausgestattet, die wir hier zwar nicht brauchen, wenn wir aber mehrere Buttons im Fragment hätten, würde dieser Parameter recht nützlich sein (wir werden dies bei der Methode onControlButtonClicked() sehen).

```
@Override
public void onContentAButtonClicked(int buttonID, String optionalData) {
     sendDataToControl(optionalData);
}
@Override
public void onContentBButtonClicked(int buttonID, String optionalData) {
     sendDataToControl(optionalData);
}
```

Die eigentliche Arbeit wird von der “sendDataToControl” Methode übernommen. Zuerst prüfen wir, ob wir im one oder two pane Modus sind hierfür habe ich eine eigene Methode geschrieben, welche weiter Unten erklärt wird. Sind wir im one pane Modus, so ist davon auszugehen, dass wir gerade ContentFragmentA oder ContentFragmentB sehen (sonst würde diese Methode nicht aufgerufen werden – sie wird ja durch den Button „Send back“ aktiviert). Wenn dies also der Fall ist, dann müssen wir das ContentFragment durch das ControlFragment austauschen. Weiterhin müssen wir in das Bundle beim Aufruf den Datenwert eintragen, der in das EditText Feld geschrieben wird (in der Methode ControlFragment.onStart() lesen wir ja das Bundle aus und tragen den Wert in das EditText Feld ein). Am Schluss wird die Trasaktion zum Austausch des Fragments gestartet und committed:

```
private void sendDataToControl(String optionalData) {
     if (!inTwoPaneMode()) {
          ControlFragment newFragment = new ControlFragment();
          Bundle args = new Bundle();
          args.putString(ControlFragment.FRAG_MESSAGE_DEF_C,
            optionalData);
          newFragment.setArguments(args);
          FragmentTransaction transaction =
            getSupportFragmentManager().beginTransaction();
          transaction.replace(R.id.my_fragment_container, newFragment);
          transaction.addToBackStack(null);
          transaction.commit();
```

Sollten wir im two pane Modus sein, so können wir sicher sein, dass das ControlFragment sichtbar ist. Insofern können wir einfach den EditText View suchen und den Wert eintragen:

```
     } else {
```

```
          ((EditText) findViewById(
            R.id.editControlText)).setText(optionalData);
     }
}
```

Schwieriger wird es, wenn ein Button im ControlFragment gedrückt wird. Hier haben wir zum einen mehrere Buttons, zum anderen reagieren sie unterschiedlich – je nach Modus. Insofern prüfe ich in dieser Methode lediglich, welcher Button für den Aufruf dieser Methode verantwortlich war und rufe dementsprechend die notwendige Methode auf. Vorher hole ich jedoch noch aus dem EditText View den eingegebenen Text raus, damit ich ihn gegebenenfalls weiterleiten kann:

```
@Override
     public void onControlButtonClicked(int buttonID,
          String optionalData) {
     String messageValue = ((EditText)
     findViewById(R.id.editControlText)).getText().toString();
     switch(buttonID){
          case R.id.showFragA:
              sendDataToContentA(messageValue);
               break;
          case R.id.showFragB:
               sendDataToContentB(messageValue);
               break;
          case R.id.sendControlText:
               sendControlText(messageValue);
          break;
     }
}
```

Nun können wir einen Blick auf die aufgerufenen Methoden werfen. Fangen wir mit den Methoden „sendDataToContentA“ und „sendDataToContentB“ an – wobei ich hier nur eine der beiden Methoden beschreibe, die zweite sieht wieder identisch aus, lediglich die „A“ Indexwerte werden durch ein „B“ ersetzt. Ich beginne mit der Prüfung, ob wir im two pane Modus sind. Ist dies der Fall, müssen wir wiederum prüfen, ob gerade der ContentFragmentA gezeigt wird. (Ich mache das, indem ich sehe, ob ich das linearLayoutA finde. Alternativ kann man ach prüfen, ob contentFragment.getClass().getName() die Klasse „com.example.myfragment.ContentFragmentA„ zurückgibt.). Wenn ContentFragmentA gezeigt wird, dann kann der Wert direkt eingetragen werden. Hierfür habe ich die public Methode „setTextValue“ vorgesehen (nur um eine Alternative zum direkten Reinschreiben zu zeigen, so wie ich es in sendDataToControl() gemacht habe.

```
private void sendDataToContentA(String optionalData) {
     if (inTwoPaneMode()) {
          if (findViewById(R.id.linearLayoutA) != null){
               ContentFragmentA contentFragment = (ContentFragmentA)
                 getSupportFragmentManager().findFragmentById(
                 R.id.my_content_fragment_container);
               contentFragment.setTextValue(optionalData);
```

Sollte ContentFragmentA nicht gezeigt werden, so muss ein neues ContentFragmentA erzeugt werden und das existierende Fragment im „my_content_fragment_container“ dadurch ersetzt werden. Um die Daten nun zu übertragen, setzen wir den Textwert in das Bundle rein.

```
          } else {
              ContentFragmentA newFragment = new ContentFragmentA();
               Bundle args = new Bundle();
               args.putString(ContentFragmentA.FRAG_MESSAGE_DEF_A,
                 optionalData);
```

```
            newFragment.setArguments(args);
            FragmentTransaction transaction =
              getSupportFragmentManager().beginTransaction();
            transaction.replace(R.id.my_content_fragment_container,
              newFragment);
            transaction.addToBackStack(null);
            transaction.commit();
        }
```

Für den Fall, dass wir im one pane Modus sind, so können wir immer davon ausgehen, dass wir das Fragment austauschen müssen – schließlich kam der Befehl für diese Methode von einem Button im ControlFragment. Der Einzige Unterschied zum Code im oberen Bereich ist nun, dass wir das neu erzeugte ContentFragmentA in den „my_fragment_container“ platzieren,

```
    } else {
        ContentFragmentA newFragment = new ContentFragmentA();
        Bundle args = new Bundle();
        args.putString(ContentFragmentA.FRAG_MESSAGE_DEF_A,
          optionalData);
        newFragment.setArguments(args);
        FragmentTransaction transaction =
          getSupportFragmentManager().beginTransaction();
        transaction.replace(R.id.my_fragment_container, newFragment);
        transaction.addToBackStack(null);
           transaction.commit();
    }
}
```

Wie vorher schon erwähnt, müsst ihr die Methode sendDataToContentB identisch aufbauen. Kommen wir nun zu der Methode, welche durch den Button „Update“ aufgerufen wird. Dieser soll nur dann eine Funktion aufweisen, wenn wir im two pane Modus sind. Der Einfachheit halber prüfen wir hier lediglich, ob in der Activity das linearLayoutA oder linearLayoutB gefunden wird. Dementsprechend wissen wir, ob wir das contentFragment auf ContentFragmentA oder ContentFragmentB casten müssen (auch hier könnten wir alternativ über getClass() gehen). Nun können wir ohne Probleme auf die setTextValue Methode zugreifen und den Wert übermitteln.

```
private void sendControlText(String optionalData) {
    Fragment contentFragment =
    getSupportFragmentManager().findFragmentById(
        R.id.my_content_fragment_container);
    if ((contentFragment != null) &&
     (findViewById(R.id.linearLayoutA) != null)) {
        ((ContentFragmentA)contentFragment).setTextValue(optionalData);
    } else  if ((contentFragment != null) &&
     (findViewById(R.id.linearLayoutB) != null)) {
        ((ContentFragmentB)contentFragment).setTextValue(optionalData);
    }
}
```

Blicken wir noch kurz auf die Mehtode, welche feststellt, ob wir im one pane oder two pane Modus sind. Wir wissen aufgrund unserer Layoutvorgaben, dass die App in vertikaler Ausrichtung immer im one pane Modus ist. Dies prüfen wir somit zuerst. Wenn wir in horizontaler Ausrichtung sind, müssen wir lediglich feststellen, ob my_content_fragment_container existiert, welcher nur dann vorkommen kann, wenn wir ein Tablett haben.

```
private boolean inTwoPaneMode() {
    if (getResources().getConfiguration().orientation ==
```

```
      Configuration.ORIENTATION_PORTRAIT) {
         return false;
    }
    return (getSupportFragmentManager().findFragmentById(
      R.id.my_content_fragment_container) != null);
}
```

Kommen wir nun zum schwierigsten Teil unserer App, die onCreate Methode. Hier müssen wir sicherstellen, dass die richtigen Fragments erzeugt werden, sie aber nicht doppelt erzeugen. Gehen wir den Code im Einzelnen durch. Zuerst rufen wir die onCreate Methode in der Superklasse auf und setzen das Layout ein (wir erinnern uns – Android kümmert sich darum, ob das Standardlayout oder das Layout aus dem layout-large-land Ordner verwendet wird):

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

Nun prüfen wir, ob der my_fragment_container existiert. Wenn dem so ist, dann wissen wir, dass wir im one pane Modus sind. Jetzt prüfen wir, ob savedInstanceState existiert – die Activity also nicht komplett neu gestartet wurde und die einzelnen internen Viewwerte im Speicher bereits existieren (durch den Aufruf der onCreate Methode der Superklasse). Jetzt müssen wir nur noch prüfen, ob wir den my_fragment_container finden. Wenn dieser noch nicht existiert, dann müssen wir ihn neu erzeugen. Existiert er bereits, dürfen wir ihn nicht nochmal erzeugen, da er sonst doppelt existiert – wir brechen also mit „return“ ab:

```
    if (findViewById(R.id.my_fragment_container) != null) {
        if (savedInstanceState != null) {
            if (getSupportFragmentManager().findFragmentById(
              R.id.my_fragment_container) != null)
                return;
        }
        ControlFragment firstFragment = new ControlFragment();
        firstFragment.setArguments(getIntent().getExtras());
            getSupportFragmentManager().beginTransaction()
          .add(R.id.my_fragment_container, firstFragment).commit();
```

Für den Fall, dass wir den my_fragment_container nicht finden, müssen wir im two pane Modus sein. Wir prüfen dann auch hier, ob wir in diese Methode nach dem Nachladen der Views aus dem savedInstanceState gelangt sind, oder nicht. Wenn savedInstanceState null ist, dann wurde die Activity zum ersten mal aufgerufen. Wir können somit das ContentFragmentA laden. Das ControlFragment müssen wir in dieser Situation nicht laden, da es ja statisch im **activity_main.xml** File hinterlegt wurde.

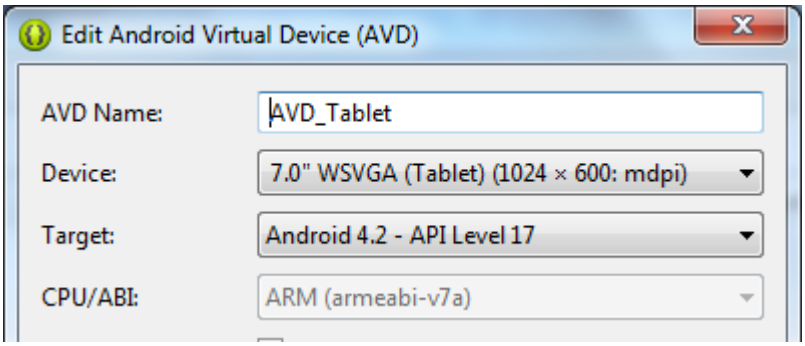
```
    } else {
        if (savedInstanceState == null) {
            ContentFragmentA secondFragment =
              new ContentFragmentA();
            secondFragment.setArguments(getIntent().getExtras());
            getSupportFragmentManager().beginTransaction()
              .add(R.id.my_content_fragment_container,
              secondFragment).commit();
        }
    }
}
```

Zum Schluss kommt noch die Methode hinzu, welche das Options Menü erzeugt. Wir belassen Sie wie sie von Eclipse eingetragen wurde. Mit diesem Thema werden wir uns später näher beschäftigen.

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    getMenuInflater().inflate(R.menu.activity_main, menu);
    return true;
}
```

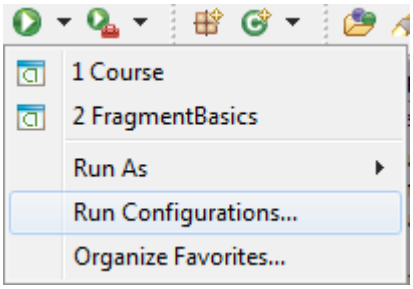
9.3 Testen der App

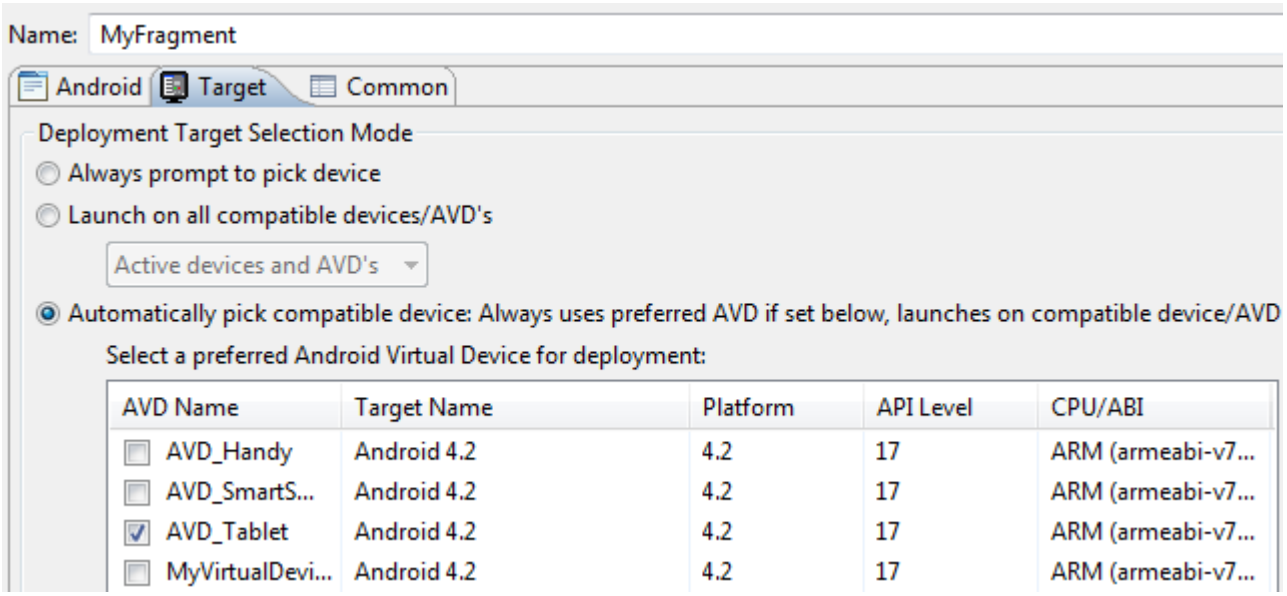
Um die App testen zu können, müssen wir zwei verschiedene virtuelle Devices konfigurieren. Zuerst das „normal“ Handheld, welches wir für die normalen Tests ohnehin schon verwendet haben und ein Tablett. Hierfür erzeugen wir in unserem „Android Virtual Device Manager“ ein Tablett (ich habe das Device „7.0‘‘ WSVGA (Tablet) verwendet (siehe links).



Wenn wir die App hierauf starten, dann müssen wir auch die Funktionalität testen das Gerät während des Betriebes zu drehen. Dies erreichen wir durch das Drücken der „Strg“ + „F11“ bzw. der „Strg“ + „F12“ Tasten.

Wenn ihr Probleme habt, die App einem bestimmten Device zuzuordnen, dann könnt ihr das über „Run Configurations“ einstellen. Ihr klickt bei dem Button auf das schwarze Dreieck und wählt „Run Configurations aus. Im folgenden Menü müsst ihr auf das Tab „Target“ gehen und dort das Device auswählen, auf dem die App laufen soll.





Ihr solltet alle Möglichkeiten (Handheld, Tablett, vertikal starten, horizontal starten, während dem Lauf die Position wechseln) mal durchprobieren, ob auch wirklich alle Optionen berücksichtigt wurden.

10 Lizenz

11 Haftung

Ich übernehme keinerlei Haftung für die Richtigkeit der hier gemachten Angaben. Sollten Fehler in dem Dokument enthalten sein, würde ich mich über eine kurze Info unter maik.aicher@gmx.net freuen.