Kapitel 3

Layout-Container in Android-Anwendungen

Wir haben in unserem ersten Beispiel bereits ein einfaches, lineares Layout gestaltet. In diesem Abschnitt gehen wir nun genauer auf die Layout-Gestaltung unter Android ein. Im SWT haben wir bereits Layout-Manager kennen gelernt. Diese hatten wir jedoch direkt im Java-Quelltext erzeugt und konfiguriert. Android stellt zwei Wege bereit, das Layout einer App zu konfigurieren:

- Konfiguration im Java-Quelltext der Activity – ähnlich wie wir unsere Layout-Manager im SWT benutzt haben.
- Nutzung der Layout-Ressourcen in XML.

Android stellt uns sehr komfortable ***Layout-Ressourcen*** in XML-Form bereit. Diese Dateien liegen im einfachsten Fall im Verzeichnis `res/layouts` innerhalb des Projektverzeichnisses.

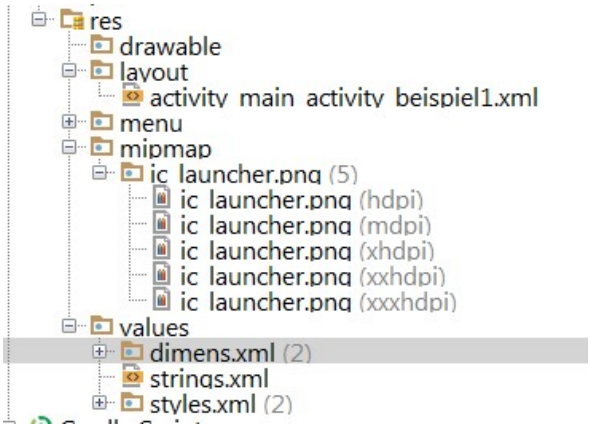

Abbildung 3.1: Pfad für Layout-Dateien

Interessant sind unter anderem folgende Layout-Container:

Layout-Container des Android-SDK	
Layout	**Erläuterung**
`LinearLayout`	Ordnet seine Kind-Widgets vertikal oder horizontal an. Ob die Kind-Widgets bis zur Parent-Grenze „glatt gezogen“ werden, wird für jedes Kind-Widget separat in seinen eigenen Attributen festgelegt.
`GridLayout`	Gitterförmiges Layout. Erst ab API-Level 14 verfügbar.
`TableLayout`	Ähnlich wie `GridLayout`, aber für Tabellenausgaben gedacht. `GridLayout` wird eher für die Anordnung fest verankerter Widgets (z. B. Taschenrechner) verwendet.
`FrameLayout`	Enthält immer nur ***ein*** Widget. Dieses kann jedoch beliebige Kind-Widgets enthalten.
`RelativeLayout`	In ihm können Widgets relativ zueinander (Bezug zum Nachbarwidget) positioniert werden.
`ScrollView`	Kann horizontal oder vertikal orientiert sein. Wenn die Anzahl der Controls in der View die Höhe / Breite des Displays übersteigt, können wir weiterscrollen. Dieses Layout kann nur Kind-Widgets, aber keine weiteren inneren Layouts enthalten.
`ListView`	kann zur Laufzeit mit Inhalts-Views (zB TextView) bestückt werden – arbeitet nach MVC-Pattern.
`RadioGroup`	Fasst RadioButtons in einer Single-Choice-Auswahl zusammen.

Tabelle 3.1: Layout-Container des Android-SDK

Die folgende Tabelle zeigt einige häufig verwendete Layout-Attribute:

Attribute der Layout-Container	
Attribut	**Erläuterung**
`android:layout_width`	Kann meist die folgenden Werte annehmen: • `fill_parent` : füllt die umrahmende Eltern-Instanz in voller Breite aus. Ist veraltet und sollte durch `match_parent` ersetzt werden. • `match_parent` : das neuere Äquivalent zu `fill_parent` • `wrap_content` : das Layout wird nur so breit, wie sein breitester Inhalt.
`android:layout_height`	Kann meist die folgenden Werte annehmen: • `fill_parent` : füllt die umrahmende Eltern-Instanz in voller Höhe aus. Ist veraltet und sollte durch `match_parent` ersetzt werden. • `match_parent` : das neuere Äquivalent zu `fill_parent` • `wrap_content` : das Layout wird nur so hoch wie sein Inhalt.

Fortsetzung auf nächster Seite

Attribut	Erläuterung
`android:orientation`	Kann die folgenden Werte annehmen: • `vertical` : Vertikal orientiertes Layout. Die Kind-Widgets werden untereinander platziert. • `horizontal` : Horizontal orientiertes Layout. Die Kind-Widgets werden nebeneinander platziert.
`android:id`	Hier kann dem Layout / dem Widget eine eigene ID gegeben werden, unter der es im Java-Quelltext der Activity ansprechbar ist.

Tabelle 3.2: Bekannteste Layout-Attribute

Die unterschiedlichen Layouts besitzen noch sehr viel mehr Attribute. Sie sollten bei Bedarf in der API nachgeschlagen werden oder über die Code-Completion innerhalb von Eclipse ausprobiert werden.

Häufig werden in den Attributen der Layouts oder der Kind-Widgets auch Größen definiert, wie z. B. Fontgröße oder Längen- und Breitenangaben von Kind-Widgets. Hierzu stehen folgende Maßeinheiten zur Verfügung:

Von Android unterstützte Maßeinheiten	
Maßeinheit	**Erläuterung**
`dp` bzw. `dip`	***D***ensity ***I***ndependent ***P***ixels. Eine abstrakte Maßeinheit, die von der Display-Auflösung abhängig ist. Die Maßeinheit wird relativ zu einem 160dpi-Display skaliert – hier entspricht ein dip etwa einem Pixel. Bei höher aufgelösten Displays wird heraufskaliert, bei niedriger aufgelösten Displays wird herunterskaliert. ***Wird für die Größendefinition von Widgets, Icons, etc. verwendet.***
`sp`	***S***cale Independent ***P***ixels. Ist wie `dip` eine abstrakte Größe. ***Wird für die Größendefinition von Fonts verwendet.***
`pt`	***P***oin***t***s . $\frac{1}{72}$ inch. Wird nicht zur Verwendung empfohlen.
`px`	***P***i***x***els. Größe eines Pixels auf dem jeweiligen Display. Wird nicht zur Verwendung empfohlen.
`mm`	Millimeter. Wird nicht zur Verwendung empfohlen.

Tabelle 3.3: Von Android unterstützte Maßeinheiten (Quelle: [16])

3.1 Resource-Datei von Layouts am Beispiel LinearLayout – *Beispiel3Android*

Das folgende Beispiel-Projekt *Beispiel3Android* zeigt eine einfache, noch nicht reaktionsfähige Activity mit einem linearen Layout. Dieses hält 6 nicht editierbare Textfelder, sowie einen Button für die „Einladung“ von Gästen zu einem „'Dinner for one“.

Abbildung 3.2: Screenshot der Hauptactivity von Projekt *Beispiel3Android*

Es folgt die Layout-Definition der Haupt-Activity. Zunächst der Kopf der Layout-Datei:

```
1 <LinearLayout
2     xmlns:android="http://schemas.android.com/
        apk/res/android"
3     android:orientation="vertical"
4     android:layout_width="match_parent"
5     android:layout_height="wrap_content">
```

In den äußeren Tags wird vermerkt, welche Layout-Typ verwendet wird. Das Start-Tag (hier: `<LinearLayout ...>` enthält die Attribute des Layouts. Zeile 1

`xmlns:android=....`

bindet hierbei den Namensraum `android` ein, so dass wir aus diesem später Attribute wie `android:id` benutzen können.

Nach dem Start-Tag des Layouts folgen die zu platzierenden Kind-Widgets:

```
6     <TextView
7         android:id="@+id/textview1"
8         android:layout_width="match_parent"
9         android:layout_height="wrap_content"
10        android:background="#FF0000"
11        android:textSize="70.0sp"
12        android:textColor="#FFFFFF"
13        android:text="@string/textfield1"/>
14    <TextView android:id="@+id/textview2"
15        android:layout_width="match_parent"
16        android:layout_height="wrap_content"
17        android:background="#0000FF"
18        android:textSize="70.0sp"
19        android:textColor="#FFFFFF"
20        android:text="@string/textfield2"/>
21    <!-- und so weiter fuer die anderen 4
        TextViews ...-->
```

```
22  <!-- nach den TextViews folgt noch ein
      Button ... -->
23  <Button android:id="@+id/button1"
24    android:textSize="70.0sp"
25    android:layout_width="match_parent"
26    android:layout_height="wrap_content"
27    android:text="@string/button1"
28    />
29 </LinearLayout>
```

Auf die ersten beiden Textfelder folgen noch 4 weitere, die bis auf die Betitelung mit `android:text` gleich definiert sind.

Erklärungsbedürftig sind die folgenden Attribute:

- **android:background**: Hier wird die Hintergrundfarbe des Textfeldes als RGB-Wert in hexadezimaler Form angegeben. Die einleitende Raute **#** bedeutet: Hier kommt ein RGB-Wert. Die drei Hex-Bytes FF0000 sind als rotgrünblau zu interpretieren.
- **android:textSize**: Hier wird die Fontgröße der Beschriftung angegeben. Sie sollte immer in **sp** angegeben werden.
- **android:textColor**: Hier wird die Farbe der Beschriftung (Vordergrundfarbe) ebenfalls als RGB-Wert in hexadezimaler Form angegeben.

Die String-Betitelungen für Textfelder und Buttons liegen im Projektverzeichnis in der Datei `res\values\strings.xml`:

```
<resources>
    <string name="app_name">Beispiel3Android</string>
    <string name="hello_world">Hello world!</string>
    <string name="menu_settings">Settings</string>
    <string name="textfield1">Pommeroy</string>
    <string name="textfield2">Winterbottom</string>
    <string name="textfield3">v. Schneider</string>
    <string name="textfield4">Toby</string>
    <string name="textfield5">James</string>
    <string name="textfield6">Sophie</string>
    <string name="button1">Invite</string>
</resources>
```

Der Java-Quellcode für die Haupt-Activity enthält wieder nur eine überschriebene *onCreate()*-Methode:

```
1  package guidev.inf.hsas.de.beispiel3neu;
2
3  import android.app.Activity;
4  import android.os.Bundle;
5
6  public class Beispiel3Activity
7          extends Activity {
8
9      @Override
10     protected void onCreate(
11             Bundle savedInstanceState) {
12         // rufe ererbte onCreate()-Methode auf
13         super.onCreate(
14                 savedInstanceState);
15
16         // XML-Layout der Haupt-Activity
17         // ist in Klasse R anhand des
18         // Dateinamens der XML-Datei
19         // verankert. Er wird hier als
20         // View aufgerufen.
21         setContentView(
22                 R.layout.activity_beispiel3);
23     } // end method onCreate()
24 } // end class
```

3.2 ScrollView / HorizontalScrollView – *Beispiel5*

Das oben gezeigte Layout ist für viele Anwendungsbeispiele vollkommen ausreichend. Lineare Laoyuts können auch geschachtelt werden. Für kompliziertere Layouts können wir auch ein *GridLayout* oder ein *RelativeLayout* verwenden. Diese Layout-Typen setzen jedoch alle voraus, dass der Inhalt der Activity auf einen Bildschirm paßt. Wenn dies nicht der Fall ist, benötigen wir ein ***scrollbares*** Layout. Hier bietet sich die *ScrollView* an. Bei ihr müssen jedoch einige Besonderheiten beachtet werden:

- Eine *ScrollView* kann nur ***ein einziges*** Child-Item verwalten.
- Dieses eine Child-Item ist dann meist ein weiterer Layout-Manager, wie beispielsweise *LinearLayout* oder *RelativeLayout*. In ihm sind die darzustellenden Kind-Widgets platziert.
- Eine *ScrollView* ist immer vertikal orientiert →daher muss auch der ihr zugeordnete Layout-Manager vertikal orientiert sein.
- Eine *HorizontalScrollView* ist immer horizontal orientiert →daher muss auch der ihr zugeordnete Layout-Manager horizontal orientiert sein.

Wir zeigen ein Beispiel:

Abbildung 3.3: Screenshot der Hauptactivity von Projekt *Beispiel5Android*

Das Layout mit den *TextView*-Objekten ist vertikal scrollbar. Es ist allerdings nicht der aus Standard-GUIs gewohnte ScrollBar sichtbar – wir können das Layout aber mit der von Tablets und Smartphones gewohnten Wisch- (Swipe-)Bewegung hinauf und hinunter navigieren.

Wir zeigen zunächst wieder die Layout-Definition der Haupt-Activity:

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <ScrollView xmlns:android="http://schemas.
     android.com/apk/res/android"
3      android:layout_width="match_parent"
4      android:layout_height="match_parent"
5      >
6      <TextView
7          android:layout_width="wrap_content"
8          android:layout_height="wrap_content"
9          android:textSize="50sp"
10         android:text="@string/langer_text"
11          />
12
13 </ScrollView>
```

In diesem Beispiel fällt auf:

- Ab Zeile 6 sehen wir, dass die *ScrollView* tatsächlich nur ein einziges direktes Kind-Objekt enthält: Ein Textfeld
- Dieses enthält einen sehr langen Text.
- Dieser Text ist in den String-Ressourcen verankert.

Die String-Ressourcen zur Betitelung der *TextView*-Objekte sehen hier folgendermaßen aus:

```
<resources>
    <string name="app_name">Beispiel5</string>
    <string name="langer_text">
        Es klapperten die Klapperschlangen,
        bis ihre Klappern schlapper klangen.
          Es klapperten die Klapperschlangen,
        bis ihre Klappern schlapper klangen.
          Es klapperten die Klapperschlangen,
        bis ihre Klappern schlapper klangen.
          Es klapperten die Klapperschlangen,
        bis ihre Klappern schlapper klangen.
          Es klapperten die Klapperschlangen,
        bis ihre Klappern schlapper klangen.
    </string>
</resources>
```

Da die Activity derzeit noch kein Dialogverhalten enthält, besteht der Java-Quellcode für die Haupt-Activity enthält wieder nur aus einer Kind-Klasse von *Activity* mit einer überschriebenen *onCreate()*-Methode:

```
1  package de.hsas.inf.guidev.beispiel5;
2  import android.app.Activity;
3  import android.os.Bundle;
4
5  public class Beispiel5Activity
6    extends Activity {
7
8      @Override
9      protected void onCreate(
10       Bundle savedInstanceState) {
11
12         super.onCreate(savedInstanceState);
13
14         setContentView(
15           R.layout.activity_beispiel5);
16
17     } // end method onCreate()
18 } // end class Beispiel5Activity
```

Merke

- Eine *ScrollView* oder *HorizontalScrollView* wird verwendet, wenn das Layout einer Activity nicht mehr in ein Fenster hineinpasst.
- Eine *ScrollView* / *HorizontalScrollView* kann nur ***ein einziges*** Kind-Objekt verwalten. Dieses kann ein einzelnes Widget sein – oder aber ein weiterer Layout-Manager.
- Die Bestückung eines Layouts innerhalb einer *ScrollView* / *HorizontalScrollView* kann – wie bei den anderen bisher behandelten Layouts – statisch innerhalb der XML-Layoutdefinition erfolgen.

3.3 MVC mit AdapterViews am Beispiel ListView – *Beispiel6*

Die bislang verwendeten Layouts beinhalteten fest verankerte Kind-Widgets. Es gibt jedoch auch Apps, bei denen sich erst zur Laufzeit entscheidet, wie viele z.B. Listenelemente in der Activity erscheinen sollen. Auch können sich zur Laufzeit die Listenelemente verändern – sie können z. B. entfernt werden oder es können welche hinzukommen. Für diese Widgets gibt es die sog. ***AdapterViews***. Hierbei handelt es sich um Views, die zur Laufzeit ihre Daten aus Objekten einer ***Adapter-Klasse*** (z. B. Listen, ähnlich C++-Templates) beziehen. Der Inhalt dieser Objekte kann aus beliebigen Datenquellen kommen:

- Nutzereingaben,
- Datenbankabfragen,
- Netzwerk-Kommunikation (Sockets),
- Dateien,
- ...

Diese Adapter fungieren als Schnittstelle zwischen darstellender View und der echten Datenhaltung (vgl. MVC-Architektur):

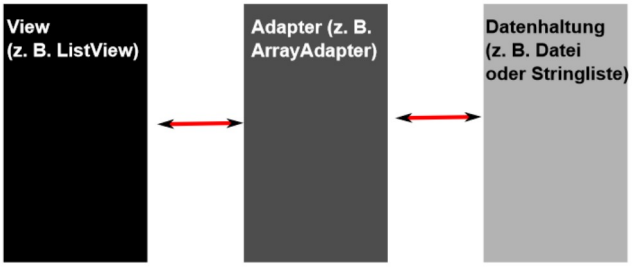

Abbildung 3.4: Zusammenspiel AdapterView und Adapter

Ein Adapter-Objekt bekommt hierbei z.B. ein Array mit Elementen aus der Datenhaltung übergeben. Dieses Adapter-Objekt wird dann mit seinem Inhalt bei der View registriert. Das folgende Klassendiagramm zeigt zunächst einen Auszug aus der Android-Adapter-Klassenhierarchie:

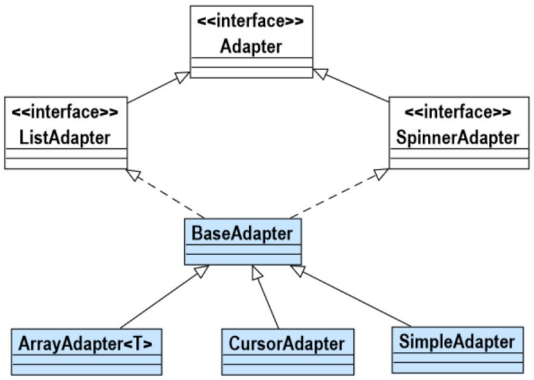

Abbildung 3.5: Auszug aus der Android-*Adapter*-Klassenhierarchie

Die gebräuchlichsten Adapterklassen implementieren hierbei die Interfaces *ListAdapter* und *SpinnerAdapter*.

Das folgende Klassendiagramm zeigt einen Auszug aus der Android-AdapterView-Klassenhierarchie:

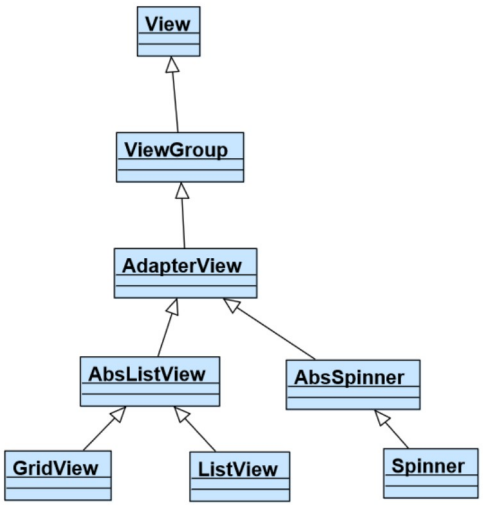

Abbildung 3.6: Auszug aus der Android-*AdapterView*-Klassenhierarchie

3.3.1 Einfache ListView-Anwendung ohne Dynamik

Unser nächstes Beispiel zeigt eine zur Laufzeit gefüllte *ListView*, welche ihre Strings über einen *ArrayAdapter* geliefert bekommt:

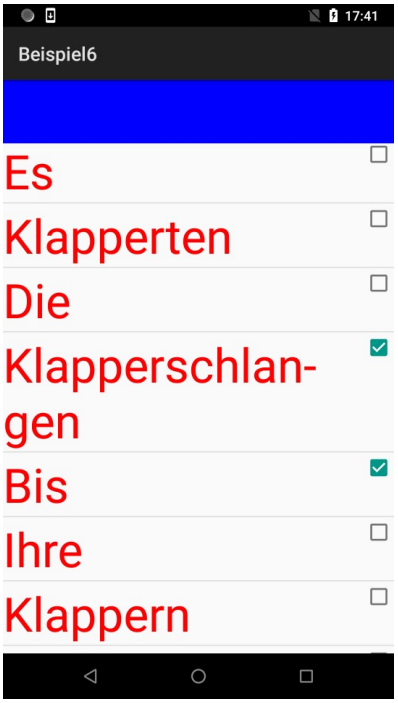

Abbildung 3.7: *Beispiel6Android* mit *ListView* mit *CheckedTextView*-Elementen

Wir benötigen hierzu zwei Layoutdefinitionen:

- `activity_beispiel6.xml` : Sie definiert das Layout der Haupt-Activity.
- `simple_list_item_multiple_choice.xml` : Sie definiert das Layout eines Listenelementes.

Beide Layout-Definitionen sind in der generierten Klasse *R* als Ressource in der Inner Class *R.layout* zugreifbar.

Zunächst das Layout der Haupt-Activity:
activity_beispiel6.xml:

```
1  <LinearLayout xmlns:android="http://schemas.
     android.com/apk/res/android"
2      android:layout_width="match_parent"
3      android:layout_height="match_parent"
4      android:orientation="vertical"
5      >
6      <TextView
7          android:id="@+id/choice"
8          android:layout_width="match_parent"
9          android:layout_height="wrap_content"
10         android:background="#0000FF"
11         android:textColor="#FFFFFF"
12         android:textSize="50sp" />
13     <ListView
14         android:id="@android:id/list"
15         android:layout_width="match_parent"
16         android:layout_height="match_parent"
17         android:choiceMode="multipleChoice"/>
18 </LinearLayout>
```

Wir sehen, dass hier – im Gegensatz zu unserem *ScrollView*-Beispiel – keine einzelnen Listenelemente definiert sind. Sie werden später im Java-Code zur Laufzeit generiert werden.

Das Layout der einzelnen *ListView*-Elemente wird in einer eigenen Layout-Datei definiert: **simple_list_item_multiple_choice.xml:**

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <CheckedTextView
3      xmlns:android="http://schemas.android.com/apk/res/android"
4      android:id="@android:id/text1"
5      android:layout_width="match_parent"
6      android:layout_height="match_parent"
7      android:textSize="50sp"
8      android:textColor="#FF0000"
9      android:checkMark="?android:attr/listChoiceIndicatorMultiple">
10 </CheckedTextView>
```

Wir haben hier als Listenelement ein Textfeld mit einer CheckBox eingeführt: Das Widget *CheckedTextView*. Zeile 9 zeigt hier eine Attribut-Zuweisung, die wir bislang noch nicht kennen:

`android:checkMark="?android:attr/listChoiceIndicatorMultiple"`

Sie weist dem Attribut *android:checkMark* einen Wert aus dem Namensraum des Standard-Android-Themes (vgl. Kapitel 3.4) zu.

Die Befüllung der *ListView* geschieht im Java-Code der Haupt-Activity:

```
1  package de.hsas.inf.guidev.beispiel6;
2  import android.os.Bundle;
3  import android.app.ListActivity;
4  import android.widget.ArrayAdapter;
5  import android.widget.ListView;
6
7  public class Beispiel6Activity extends
     ListActivity {
8
9      // Ein Array mit Worten als Datenquelle
10     private String[] items=
11       {"Es", "Klapperten", "Die",
12         "Klapperschlangen", "Bis", "Ihre",
13         "Klappern", "Schlapper",
14         "Klangen", "Ene", "Mene", "Mu"};
```

Neu ist an diesem Beispiel:

- Die Klasse der Haupt-Activity ist von der Klasse *ListActivity* abgeleitet.
- Die Klasse enthält als „Mock“ für eine externe Datenquelle ein Array aus Strings.

In den nächsten Zeilen sehen wir die Implementierung der Methode *onCreate()*:

```
16      private ListView listView;
17
18      @Override
19      protected void onCreate(
20        Bundle savedInstanceState) {
21          // Superklassen-Methode aufrufen
22          super.onCreate(savedInstanceState);
23          // Layout der Haupt-Activity holen
24          setContentView(R.layout.activity_beispiel6);
25
26          // Wird jetzt noch nicht nicht gebraucht
27          listView = findViewById(android.R.id.list);
28
29          // Listenadapter mit Daten bestuecken
30          ArrayAdapter<String> myList =
31            new ArrayAdapter<String>(this,
32              R.layout.simple_list_item_multiple_choice,
33              items);
34
35          // Listenadapter (Model)  mit ListView verknuepfen
36          this.setListAdapter(myList);
37      } // end method onCreate()
38  } // end class
```

Hier sind zwei Statements erklärungsbedürftig:

- Zeile 29-33: Hier wird ein *ArrayAdapter* (Model) verknüpft mit:
  - seiner Datenquelle – dem String-Array *items* von oben.
  - seiner View – dem CheckedTextField aus dem Layout `simple_list_item_multiple_choice.xml` .
- Zeile 36: Hier wird das Model aus dem *ArrayAdapter* mit der Activity (dem Controller) verknüpft. Die Methode *setListAdapter()* erbt die Activity von *ListActivity*.

Merke

- Eine *AdapterView* wie z.B. eine *ListView* wird verwendet, wenn
  - das Layout einer Activity nicht mehr in ein Fenster hineinpasst ***und***
  - die Listenelemente erst ***zur Laufzeit*** z. B. über eine Eingabe oder über eine Datei- oder Datenbankabfrage entstehen.
  - In unserem Beispiel wurde dieses „Entstehen“ über eine feste String-Liste simuliert.
- Die Formatierung der einzelnen Listenelemente geschieht meist über eine separate Layout-Vereinbarung. Bei uns war das das Layout `simple_list_item_multiple_choice.xml` .
- Die in Java geschriebene Haupt-Activity-Klasse ist nun abgeleitet von *ListActivity*.
- In ihrer *onCreate()*-Methode finden nun zwei wichtige Verknüpfungen statt:

- Die Verknüpfung der Haupt-Activity mit dem Haupt-Activity-Layout. Dies geschieht – wie gewohnt – mit dem Aufruf der ererbten Methode *setContentView()*.
- Die Erzeugung des Daten-Zugriffs-Objektes – hier ein *ArrayAdapter*-Objekt. Dieses wird direkt bei der Erzeugung mit dem Layout für einzelne Datenelemente verknüpft.

3.3.2 Umfangreichere ListView-Anwendung mit Dynamik

Bisher haben wir nur ein sehr einfaches MVC-Modell mit unserer *ListView* gezeigt: Der *ArrayAdapter* (Model) wurde mit dem Inhalt eines Arrays befüllt – und danach einfach nicht mehr geändert.

Neu: Ein MVC mit dynamisch vergrößerbarem / verkleinerbarem Inhalt über eine *ListView* mit *ArrayAdapter*.
Die folgenden Screenshots zeigen das Verhalten der Anwendung:

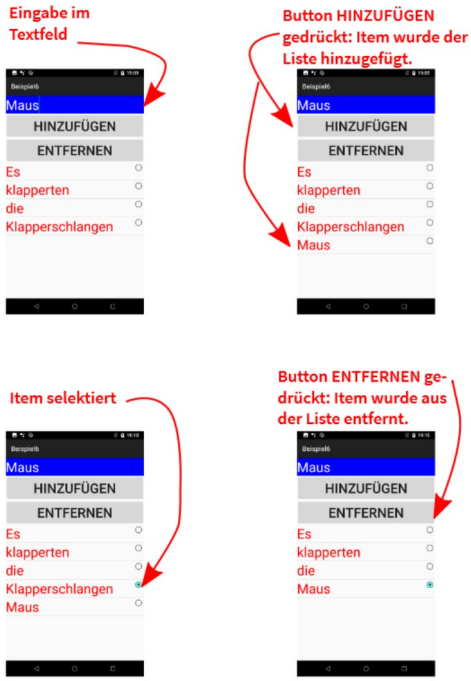

Abbildung 3.8: *Beispiel6b* mit dynamischer Elemente-Verwaltung

Zuerst wird wieder das Layout der Anwendung gezeigt – zunächst das Activity-Layout: activity_beispiel6.xml:

```
<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/input"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:background="#0000FF"
        android:textColor="#FFFFFF"
        android:textSize="40sp"
        android:maxLines="1"
        android:inputType="text"
        android:imeOptions="actionDone"
        />
```

Im oberen Bereich ist das blau eingefärbte Eingabefeld definiert.

```
17      <Button
18          android:layout_width="match_parent"
19          android:layout_height="wrap_content"
20          android:textSize="40sp"
21          android:text="@string/add"
22          android:onClick="add"
23          />
24      <Button
25          android:layout_width="match_parent"
26          android:layout_height="wrap_content"
27          android:textSize="40sp"
28          android:text="@string/sub"
29          android:onClick="sub"
30          />
31      <ListView
32          android:id="@android:id/list"
33          android:layout_width="match_parent"
34          android:layout_height="match_parent"
35          android:choiceMode="singleChoice"/>
36  </LinearLayout>
```

Danach folgen die beiden Buttons. Sie sind jeweils mit der Activity-Methode *add()* bzw. *sub()* verknüpft. Im unteren Bereich sehen wir die ListView – diesmal mit *singleChoice*-Elementen.
Mit `android:choiceMode="singleChoice"` wird angegeben, dass immer nur eines der Felder in der ListView selektiert werden kann.

Das Layout der einzelnen Listenelemente ist wieder recht kurz:

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <CheckedTextView
3     xmlns:android="http://schemas.android.com/
        apk/res/android"
4     android:id="@android:id/text1"
5     android:layout_width="match_parent"
6     android:layout_height="match_parent"
7     android:textSize="40sp"
8     android:textColor="#FF0000"
9     android:checkMark="?android:attr/
        listChoiceIndicatorSingle">
10 </CheckedTextView>
```

Mit
`android:checkMark="?android:attr/listChoiceIndicatorSingle"` wird die Ausgestaltung der Felder mit RadioButtons angegeben.

Die Activity-Klasse ist diesmal nicht mehr von *ListActivity* abgeleitet:
Beispiel6Activity.java:

```
1  package de.hsas.inf.guidev.beispiel6;
2  import android.app.Activity;
3  import android.os.Bundle;
4  import android.util.Log;
5  import android.view.View;
6  import android.widget.AdapterView;
7  import android.widget.ArrayAdapter;
8  import android.widget.CheckedTextView;
9  import android.widget.EditText;
10 import android.widget.ListView;
11
12 import java.util.ArrayList;
13 import java.util.Arrays;
14
15 // Diesmal nicht mehr von ListActivity ab-
16 // geleitet!
17 public class Beispiel6Activity
18      extends Activity {
19   private ArrayAdapter<String> model;
20   private ArrayList<String> elements;
21
22   private ListView listView;
23   private EditText input;
24   private int pos=-1;
```

In der *onCreate()*-Methode werden wieder die wichtigsten Elemente der Activity über ihre ID abgefragt:

```
25  @Override
26  protected void onCreate(
27    Bundle savedInstanceState) {
28    // Superklassen-Methode aufrufen
29    super.onCreate(savedInstanceState);
30    // Layout der Haupt-Activity holen
31    setContentView(
32      R.layout.activity_beispiel6);
33
34    // ListView mit ID hineinholen
35    listView =
36      findViewById(android.R.id.list);
37
38    // Eingabefeld mit ID hineinholen
39    input = findViewById(R.id.input);
```

Im nächsten Abschnitt der *onCreate()*-Methode wird zunächst eine *ArrayList* mit einigen Strings bestückt. Diese wird dann mit unserem Model – dem *ArrayAdapter* verknüpft.

```
40
41      // ArrayList mit Worten bestuecken.
42      // Achtung: Fuer erweiterbare/verkuerzbare
43      // Adapter muss eine Listenkonstruktion
44      // verwendet werden, kein Array!
45      elements = new ArrayList<String>(
46        Arrays.asList("Es","klapperten","die",
47          "Klapperschlangen"));
48
49      // Listenadapter fuer ListView
50      // mit Worten bestuecken
51      model = new ArrayAdapter<String>(this,
52        R.layout.simple_list_item_single_choice,
53        elements);
54
55      // ListenAdapter in die ListView der
56      // ListActivity einhaengen
57      listView.setAdapter(model);
```

Unterschied zu vorher: Wenn ein *ArrayAdapter* mit einer *ArrayList* anstatt mit einem einfachen Array initialisiert wird, kann er später wachsen oder schrumpfen.
Im letzten Schritt wird in diesem Abschnitt das Model mit der View verknüpft – diesmal über die Methode *setAdapter()* der *ListView*.

Im letzten Abschnitt der *onCreate()*-Methode bekommt die *ListView* einen Listener, der reagiert, sobald eines ihrer Items selektiert wird:

```
58      // Listener merkt sich die Position des
59      // angewaehlten Items
60      listView.setOnItemClickListener(
61        new AdapterView.OnItemClickListener() {
62          @Override
63          public void onItemClick(
64            AdapterView<?> parent,
65            View view, int position, long id) {
66
67            CheckedTextView v =
68              (CheckedTextView)view;
69
70            // wenn selektiert: Position merken
71            if (v.isChecked()) {
72              pos = position;
73            } // end if
74          } // end method onItemClick
75        } // end  anonymous class
76      ); // end call setOnItemClickListener
77    } // end method onCreate()
```

Er merkt sich die Position des zuletzt selektierten Items. Bedeutung der Übergabeparameter der Listener-Methode *onItemClick()*:

- *parent*: das Parent der angeklickten View – hier also unsere *ListView*.
- *view*: das angeklickte Item – hier also eines der *CheckedTextView*-Elemente.
- *position*: Position des angeklickten Items – kann auch zum Zugriff innerhalb des *ArrayAdapters* benutzt werden (z.B. Löschen).
- *id*: ID des angeklickten Items.

Es folgen noch die Methoden

- *add()* zum Hinzufügen eines Items zur Liste, und
- *sub()* zum Entfernen eines Items aus der Liste.

Zunächst folgt die *sub()*-Methode:

```
78  // Element entfernen
79  public void sub(View view) {
80
81    // wenn eine gueltige Position gemerkt ...
82    if (pos != -1) {
83      Log.e("SUB","pos != -1");
84      // item abfragen und ...
85      String item = model.getItem(pos);
86      // aus ArrayAdapter entfernen.
87      model.remove(item);
88
89      // Danach Aenderung an View senden
90      model.notifyDataSetChanged();
91    } // end if
92    // Falls pos an einer nicht verfuegbaren
93    // Position: umsetzen!
94    if (pos >= model.getCount()){
95      if (model.getCount() > 0){
96        pos =0;
97      }
98      else {
99        pos = -1;
100     } // end else
101   } // end if
102 } // end method
```

Wichtig: Nach dem Entfernen des Elements an der gewünschten Position aus dem *ArrayAdapter* mit dessen Methode *remove()* muss noch die *ListView* benachrichtigt werden. Dies geschieht mit der Methode *notifyDataSetChanged()* des *ArrayAdapters*.

Zuletzt folgt die *add()*-Methode:

```
103   // Hinzufuegen eines Items
104   public void add(View view) {
105     // ArrayAdapter uebernimmt Text
106     // aus Eingabefeld ...
107     model.add(input.getText().toString());
108
109     // .. und meldet Aenderung an View.
110     model.notifyDataSetChanged();
111   } // end method
112 } // end class
```

Sie entnimmt den String aus dem Eingabefeld und fügt ihn dem *ArrayAdapter* mit dessen Methode *add()* hinzu. Danach wird wieder die *ListView* mit der *ArrayAdapter*-Methode *notifyDataSetChanged()* benachrichtigt.

Merke

- Ein reaktionsfähiger MVC mit *ListView* und *ArrayAdapter* benötigt in der Regel einen Mechanismus zum Hinzufügen und Entfernen von Elementen.
- Im obigen Beispiel wurde dieser in der Activity über die Methoden *add()* und *sub()* implementiert. Jede der beiden Methoden war mit einem Button der Activity verknüpft.
- Die Elemente werden dabei im Model – hier dem *ArrayAdapter* – hinzugefügt oder entfernt.
- Das Model – hier der *ArrayAdapter* benachrichtigt danach die zugehörige View mit seiner Methode*notifyDataSetChanged()* über die Änderung.

3.4 Styles und Themes

Wir haben in den bisherigen Beispielen gesehen, dass sich viele Layout-Angaben für einzelne Kind-Widgets wiederholt haben. Hier liegt es nahe, solche Angaben global zu vereinbaren. Hierzu bietet Android zwei Techniken an:

- Styles: Hierbei handelt es sich um Ressourcen, welche in XML-Form im Projektverzeichnis unter

  `./res/styles/mystylename.xml` abgelegt werden. Sie werden dann in den Layout- Dateien ***für jedes Widget***, das sie anwenden soll, mit dem Attribut

  `style=@style/mystylename` verankert.

- Themes: Hierbei handelt es sich um wirklich globale Style-Vereinbarungen, die für die gesamte App gelten. Sie werden im gleichen Verzeichnis abgelegt wie Styles. Nur die Verlinkung erfolgt anders. Sie erfolgt in der Manifest-Datei etwa in der Form

  ```
  activity android:theme="@style:mythemename"
  ```

3.4.1 Vereinbarung von Styles – *Beispiel7Android*

Das folgende Beispiel zeigt eine *ScrollView* ähnlich wie in *Beispiel5Android*:

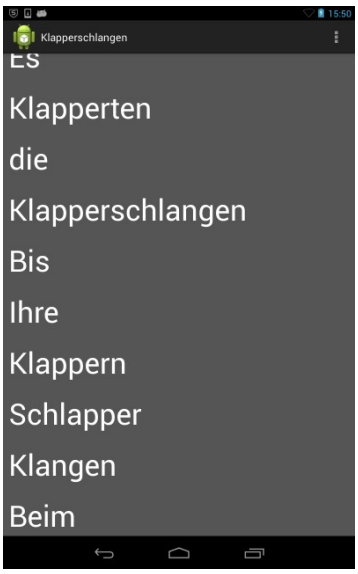

Abbildung 3.9: Screenshot der Hauptactivity von Projekt *Beispiel7Android*

Die gemeinsamen Eigenschaften der *ScrollView*-Elemente werden jedoch in einer eigenen *Style-Ressource* vereinbart. Sie kann für jedes Widget, welches nach diesem Style gestaltet werden soll, eingebunden werden:

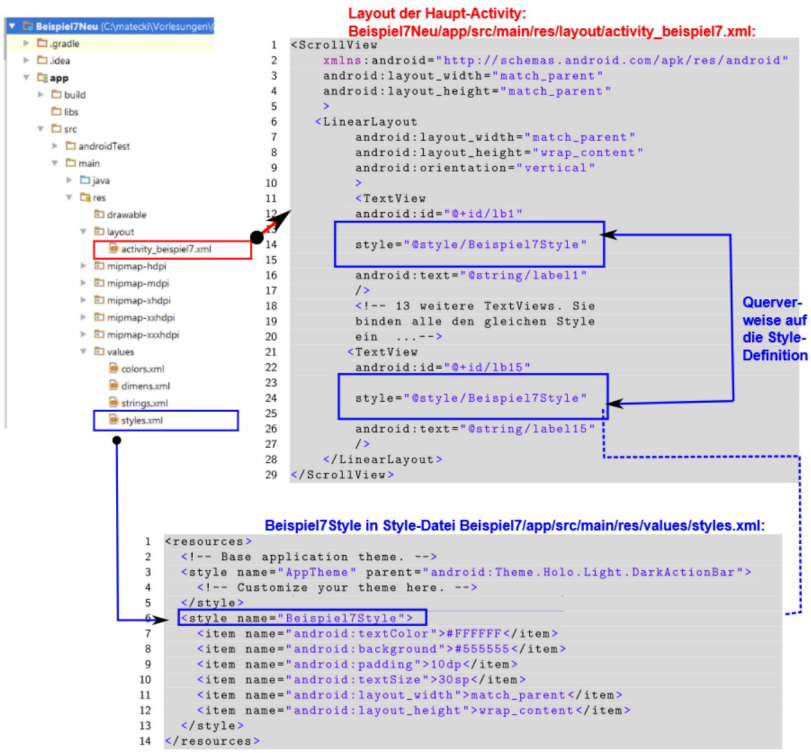

Abbildung 3.10: Zusammenspiel Layoutdefinition der Activity – Style-Definition

Die restliche Implementierung von *Beispiel7Android* ist gleich wie *Beispiel5Android* aus Kapitel 3.2.

3.4.2 Vereinbarung von Themes und Shapes – *Beispiel8Android*

Im vorherigen Beispiel haben wir gesehen, dass in allen 15 Textfeldern der gleiche Style verankert wurde. Dies kann noch weiter vereinfacht werden, indem ein Style global im Manifest vereinbart wird. Solche globalen Styles werden als *Themes* bezeichnet.
Das folgende *Beispiel8Android* zeigt die Verankerung eines solchen Themes. Hier wurde wiederum *Beispiel5Android* leicht abgewandelt:

Abbildung 3.11: Screenshot der Hauptactivity von Projekt *Beispiel8Android*

Die folgende Zeichnung zeigt das Zusammenspiel Style-Definition des Themes mit der Manifestdatei der Applikation:

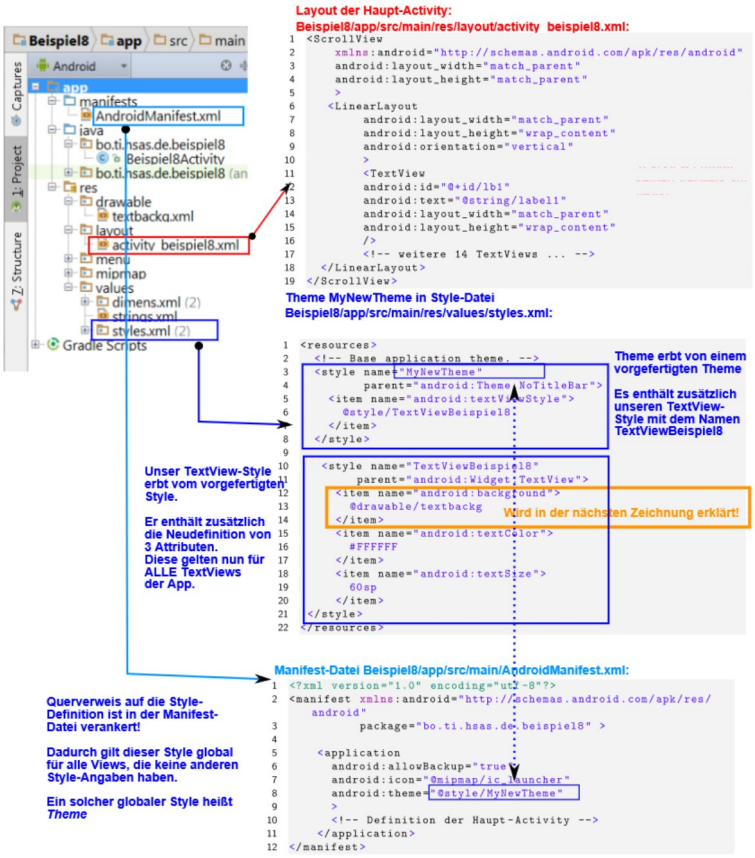

Abbildung 3.12: Zusammenspiel Style-Definition des Themes – Manifestdatei

In der Style-Definition des Themes in `Beispiel8Android/res/values/beispiel8style.xml` ist statt einer Hintergrundfarbe für die zu stylenden Views ein ***Drawable*** hinter-

legt:

```
<item name= "android:background">@drawable/textback</item>
```

Dieser Querverweis führt zur Datei **`Beispiel8Android/res/drawable/textback.xml`**. Sie enthält eine eigene ***Shape*** (Form), welche den Hintergrund künftiger Oberflächenelemente genauer spezifiziert. Derartige Shapes können definiert werden als:

- **`android:shape="rectangle"`** : Rechteck-Form mit ggf. abgerundeten Ecken
- **`android:shape="oval"`** : Ovale Form
- **`android:shape="line"`** : Linie
- **`android:shape="ring"`** : Ring

Das folgende Bild zeigt das Zusammenspiel zwischen Style-Definition und verwendeter Shape innerhalb des Styles:

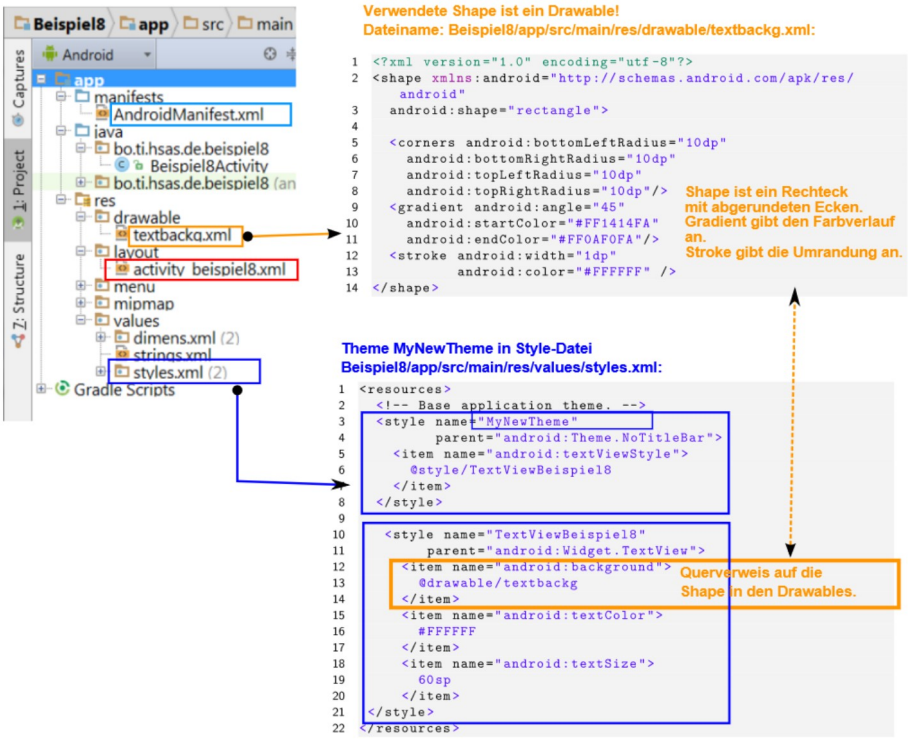

Abbildung 3.13: Zusammenspiel Style-Definition des Themes – Shape

Die Einstellungen des Themes werden für alle Views der App, für die nichts anderes angegeben wurde, übernommen. In der Shape-Definition, welche als Hintergrund für alle Views bestimmt wurde, sind zwei wesentliche Einstellungen festgelegt:

- `<gradient ... />` : legt den Farbverlauf fest
  - `android:startColor` : legt die erste von zwei Verlaufsfarben fest.

  - `android:endColor` : legt die zweite von zwei Verlaufsfarben fest.
  - `android:angle` : legt die Richtung des Farbverlaufes fest. Wird in Vielfachen von $45\,\mathrm{deg}$ festgelegt.

- `<corners .../>` : legt die Rundung der Ecken des Rechtecks fest.