Uwe Post

Android-Apps entwickeln

Galileo Press

Auf einen Blick

1 Einleitung 15
2 Ist Java nicht auch eine Insel? 39
3 Vorbereitungen 69
4 Die erste App 89
5 Ein Spiel entwickeln 131
6 Sound und Animation 179
7 Internet-Zugriff 211
8 Kamera und Augmented Reality 261
9 Sensoren und der Rest der Welt 281
10 Tipps und Tricks 329
11 Apps veröffentlichen 359

Inhalt

Vorwort 13

1 Einleitung 15

1.1 Für wen ist dieses Buch? 15
1.1.1 Magie? 16
1.1.2 Große Zahlen 16
1.1.3 Technologie für alle 17
1.1.4 Die Grenzen der Physik 18

1.2 Unendliche Möglichkeiten 19
1.2.1 Baukasten 20
1.2.2 Spiel ohne Grenzen 21
1.2.3 Alles geht 24

1.3 Was ist so toll an Android? 25
1.3.1 MapDroyd 25
1.3.2 Google Sky Map 27
1.3.3 Bump 28
1.3.4 c:geo 29
1.3.5 barcoo 31
1.3.6 Öffi 32
1.3.7 Wikitude World Browser 33
1.3.8 Sprachsuche 34
1.3.9 Cut the Rope 36
1.3.10 Shaky Tower 37

2 Ist Java nicht auch eine Insel? 39

2.1 Warum Java? 39

2.2 Grundlagen 42
2.2.1 Objektorientierung – Klassen und Objekte 42
2.2.2 Konstruktoren 44

2.3 Pakete 45
2.3.1 Packages deklarieren 45
2.3.2 Klassen importieren 46

2.4 Klassen implementieren 47
2.4.1 Attribute 48
2.4.2 Methoden 51
2.4.3 Zugriffsbeschränkungen 53
2.4.4 Eigene Konstruktoren 56
2.4.5 Lokale Variablen 57

2.5 Daten verwalten 59
2.5.1 Listen 59
2.5.2 Schleifen 61

2.6 Vererbung 63
2.6.1 Basisklassen 63
2.6.2 Polymorphie 66

3 Vorbereitungen 69

3.1 Was brauche ich, um zu beginnen? 69

3.2 JDK installieren 71

3.3 Eclipse installieren 73

3.4 Tour durch Eclipse 75

3.5 Android Development Tools installieren 77

3.6 Android SDK installieren 79

3.7 SDK Tools installieren 80

3.8 Ein virtuelles Gerät erzeugen 82

3.9 Eclipse mit dem Handy verbinden 85

3.10 Was tun, wenn mein Eclipse verrücktspielt? 86
3.10.1 Unerklärliche Unterstreichungen 86
3.10.2 Ein Handy namens Fragezeichen 87
3.10.3 Eclipse hängt sich auf 88
3.10.4 Eclipse findet Resource-Dateien nicht 88

4 Die erste App 89

4.1 Sag »Hallo«, Android! ... 89
- 4.1.1 Ein neues Android-Projekt erstellen ... 89
- 4.1.2 Die StartActivity ... 91
- 4.1.3 Der erste Start ... 97

4.2 Bestandteile einer Android-App ... 99
- 4.2.1 Versionsnummern ... 100
- 4.2.2 Activities anmelden ... 100
- 4.2.3 Permissions ... 102
- 4.2.4 Ressourcen ... 103
- 4.2.5 Generierte Dateien ... 105

4.3 Benutzeroberflächen bauen ... 110
- 4.3.1 Layout bearbeiten ... 110
- 4.3.2 String-Ressourcen ... 113
- 4.3.3 Layout-Komponenten ... 117
- 4.3.4 Weitere visuelle Komponenten ... 120

4.4 Buttons mit Funktion ... 121
- 4.4.1 Der OnClickListener ... 121
- 4.4.2 Den Listener implementieren ... 122

4.5 Eine App installieren ... 125
- 4.5.1 Start mit ADT ... 125
- 4.5.2 Installieren per USB ... 126
- 4.5.3 Installieren mit ADB ... 126
- 4.5.4 Drahtlos installieren ... 128

5 Ein Spiel entwickeln 131

5.1 Wie viele Stechmücken kann man in einer Minute fangen? ... 131
- 5.1.1 Der Plan ... 132
- 5.1.2 Das Projekt erzeugen ... 132
- 5.1.3 Layouts vorbereiten ... 133
- 5.1.4 Die GameActivity ... 134

5.2 Grafiken einbinden 138
5.2.1 Die Mücke und der Rest der Welt 138
5.2.2 Grafiken einbinden 139

5.3 Die Game Engine 141
5.3.1 Aufbau einer Game Engine 142
5.3.2 Ein neues Spiel starten 143
5.3.3 Eine Runde starten 144
5.3.4 Den Bildschirm aktualisieren 145
5.3.5 Die verbleibende Zeit herunterzählen 151
5.3.6 Prüfen, ob das Spiel vorbei ist 155
5.3.7 Prüfen, ob eine Runde vorbei ist 157
5.3.8 Eine Mücke anzeigen 158
5.3.9 Eine Mücke verschwinden lassen 162
5.3.10 Das Treffen einer Mücke mit dem Finger verarbeiten 166
5.3.11 »Game Over« 167
5.3.12 Der Handler 169

5.4 Der erste Mückenfang 173
5.4.1 Retrospektive 174
5.4.2 Feineinstellungen 174
5.4.3 Hintergrundbilder 176
5.4.4 Elefanten hinzufügen 177

6 Sound und Animation 179

6.1 Sounds hinzufügen 180
6.1.1 Sounds erzeugen 180
6.1.2 Sounds als Ressource 182

6.2 Sounds abspielen 184
6.2.1 Der MediaPlayer 184
6.2.2 MediaPlayer initialisieren 185
6.2.3 Zurückspulen und Abspielen 186

6.3 Einfache Animationen 187
6.3.1 Views einblenden 188
6.3.2 Wackelnde Buttons 191
6.3.3 Interpolation 193

6.4 Fliegende Mücken 198
6.4.1 Grundgedanken zur Animation von Views 198
6.4.2 Geschwindigkeit festlegen 198
6.4.3 Mücken bewegen 199
6.4.4 Bilder programmatisch laden 202
6.4.5 If-else-Abfragen 204
6.4.6 Zweidimensionale Arrays 205
6.4.7 Resource-IDs ermitteln 206
6.4.8 Retrospektive 208

7 Internet-Zugriff 211

7.1 Highscores speichern 211
7.1.1 Highscore anzeigen 211
7.1.2 Activities mit Rückgabewert 213
7.1.3 Werte permanent speichern 214
7.1.4 Rekordhalter verewigen 215

7.2 Bestenliste im Internet 220
7.2.1 Ein App Engine-Projekt 221
7.2.2 URL-Parameter entgegennehmen 224
7.2.3 Daten im High Replication Datastore speichern 225
7.2.4 Highscores aus dem Datastore auslesen 226
7.2.5 Die Internet-Erlaubnis 229
7.2.6 Der Android-HTTP-Client 229
7.2.7 Background-Threads 236
7.2.8 Die Oberfläche aktualisieren 238
7.2.9 Highscores zum Server schicken 240
7.2.10 HTML darstellen 242
7.2.11 HTML mit Bildern 245

7.3 Listen mit Adaptern 247
7.3.1 ListViews 247
7.3.2 ArrayAdapter 251
7.3.3 Eigene Adapter 255
7.3.4 Recyceln von Views 259

8 Kamera und Augmented Reality 261

8.1 Die Kamera verwenden ... 261
- 8.1.1 Der CameraView ... 262
- 8.1.2 CameraView ins Layout integrieren ... 266
- 8.1.3 Die Camera-Permission ... 269

8.2 Bilddaten verwenden ... 270
- 8.2.1 Bilddaten anfordern ... 270
- 8.2.2 Bilddaten auswerten ... 271
- 8.2.3 Tomaten gegen Mücken ... 274

9 Sensoren und der Rest der Welt 281

9.1 Himmels- und sonstige Richtungen ... 281
- 9.1.1 Der SensorManager ... 282
- 9.1.2 Rufen Sie nicht an, wir rufen Sie an ... 283
- 9.1.3 Die Kompassnadel und das Canvas-Element ... 285
- 9.1.4 View und Activity verbinden ... 288

9.2 Wo fliegen sie denn? ... 289
- 9.2.1 Sphärische Koordinaten ... 289
- 9.2.2 Die virtuelle Kamera ... 291
- 9.2.3 Mücken vor der virtuellen Kamera ... 293
- 9.2.4 Der Radarschirm ... 297

9.3 Beschleunigung und Erschütterungen ... 303
- 9.3.1 Ein Schrittzähler ... 304
- 9.3.2 Mit dem SensorEventListener kommunizieren ... 306
- 9.3.3 Schritt für Schritt ... 309

9.4 Hintergrund-Services ... 311
- 9.4.1 Eine Service-Klasse ... 312
- 9.4.2 Service steuern ... 314
- 9.4.3 Einfache Service-Kommunikation ... 316

9.5 Arbeiten mit Geokoordinaten ... 319
- 9.5.1 Der Weg ins Büro ... 319
- 9.5.2 Koordinaten ermitteln ... 321
- 9.5.3 Karten und Overlay ... 323

10 Tipps und Tricks 329

10.1 Fehlersuche ... 329

10.1.1 Einen Stacktrace lesen ... 330
10.1.2 Logging einbauen ... 334
10.1.3 Schritt für Schritt debuggen ... 336

10.2 Views mit Stil ... 337

10.2.1 Hintergrundgrafiken ... 338
10.2.2 Styles ... 339
10.2.3 Themes ... 340
10.2.4 Button-Zustände ... 342
10.2.5 9-Patches ... 343

10.3 Dialoge ... 345

10.3.1 Standarddialoge ... 345
10.3.2 Eigene Dialoge ... 351
10.3.3 Toasts ... 353

10.4 Layout-Gefummel ... 354

10.4.1 RelativeLayouts ... 355
10.4.2 Layout-Gewichte ... 356

10.5 Troubleshooting ... 358

10.5.1 Eclipse installiert die App nicht auf dem Handy ... 358
10.5.2 App vermisst existierende Ressourcen ... 358
10.5.3 LogCat bleibt stehen ... 358

11 Apps veröffentlichen 359

11.1 Vorarbeiten ... 359

11.1.1 Zertifikat erstellen ... 359
11.1.2 Das Entwicklerkonto ... 361
11.1.3 Die Entwicklerkonsole ... 362

11.2 Hausaufgaben ... 366

11.2.1 Updates ... 366
11.2.2 Statistiken ... 368
11.2.3 Fehlerberichte ... 370

11.3 In-App-Payment 371
11.3.1 In-App-Produkte 373
11.3.2 Der BillingService-Apparat 375
11.3.3 BillingReceiver und BillingResponseHandler 377
11.4 Alternative Markets 379
11.4.1 Amazon AppStore 380
11.4.2 AppsLib 380
11.4.3 AndroidPIT App Center 382
11.4.4 SlideME.org 384

Die Buch-DVD 387

Index 389

Kapitel 5
Ein Spiel entwickeln

»Eigentlich mag ich gar keine Pixel.«
(Pac Man)

Wenn Sie gelegentlich mit öffentlichen Verkehrsmitteln unterwegs sind, wird Ihnen die wachsende Anzahl Pendler aufgefallen sein, die konzentriert auf ihr Handy starren, anstatt die schöne Aussicht zu genießen. Manch einer liest seine E-Mails oder ist in irgendeinem sozialen Netzwerk unterwegs, andere mögen tatsächlich noch etwas Antikes wie eine SMS verfassen – aber eine ganze Reihe Menschen nutzt die Zeit im Zug, um zu spielen.

Smartphones machen es möglich. Nicht nur dem Anwender, sondern auch dem Entwickler machen Spiele mehr Spaß. Aus diesem Grund lernen Sie die weiteren Schritte der Android-Programmierung anhand eines Spiels. Wir beginnen mit einem recht simplen Game, aber in jedem Kapitel fügen Sie weitere Funktionen hinzu, bis das Resultat Ihre Mitmenschen in Staunen versetzt. Jedenfalls wenn sie hören, dass Sie es programmiert haben, obwohl Sie vor ein paar Tagen vielleicht noch gar kein Java konnten.

5.1 Wie viele Stechmücken kann man in einer Minute fangen?

Gehören Sie auch zu den bevorzugten Getränken der Insektenfamilie namens *Culicidae*? Meine Frau hat gut reden, wenn ich mitten in der Nacht das Licht anschalte, um auf die Jagd nach einem summenden Vampir zu gehen: *Ihr* Blut schmeckt den Mistviechern ja nicht. Es wird Zeit für eine fürchterliche Rache. Millionen Mücken sollen künftig zerquetscht werden, um der Gerechtigkeit Genüge zu tun. Und zwar auf den Bildschirmen Ihrer Handys, liebe Leser. Gut, die Mücken sind nicht echt, die App ist nur ein Spiel. Aber es gibt mir Genugtuung.

5.1.1 Der Plan

Wie soll das Spiel funktionieren?

Stellen Sie sich vor, dass Sie auf dem Handy-Bildschirm kleine Bilder von Mücken sehen. Sie tauchen auf und verschwinden, und es sind viele. Treffen Sie eine mit dem Finger, bevor sie verschwindet, erhalten Sie einen Punkt. Sobald Sie eine geforderte Anzahl Punkte erreicht haben, ist die Runde vorbei – spätestens aber nach einer Minute. In dem Fall heißt es »Game Over«: Sie haben verloren. Ansonsten geht das Spiel mit der nächsten, schwierigeren Runde weiter.

Vermutlich kommt Ihnen jetzt eine ganze Menge Ideen: Die Mücken könnten sich bewegen, man könnte einen Highscore speichern, oder jedes zerquetschte Insekt könnte ein lustiges Geräusch von sich geben. Solche Ideen sind wunderbar – machen Sie sich Notizen für später. Denn die Grundversion des Spiels zu bauen, ist als erster Schritt Herausforderung genug.

Halten Sie sich vor Augen, welche Fähigkeiten die Mücken-App haben muss:

- zwei verschiedene Layouts, ein Startbildschirm und das eigentliche Spiel
- Bilder von Mücken an zufälligen Stellen anzeigen und verschwinden lassen
- den berührungsempfindlichen Bildschirm verwenden, um festzustellen, ob der Spieler eine Mücke getroffen hat
- eine Zeitanzeige rückwärts laufen lassen bis zum »Game Over«

Das klingt überschaubar, nicht wahr? Es ist eine wichtige Regel für Projekte in der Informationstechnologie, größere Aufgaben in kleinere zu unterteilen und jene der Reihe nach anzugehen. Nehmen Sie sich am Anfang nie zu viel vor, denn kleine Schritte bringen schneller Erfolge, und Sie verlieren nicht so leicht den Überblick.

Bereit? Möge die Jagd beginnen.

5.1.2 Das Projekt erzeugen

Als ersten Schritt legen Sie in Eclipse ein neues Android-Projekt an. Schließen Sie alle anderen Projekte im Workspace, indem Sie das Kontextmenü mit der rechten Maustaste öffnen und CLOSE wählen.

Starten Sie den Wizard für ein neues Android-Projekt (`Strg`+`N`). Nennen Sie das Projekt nicht `Mückenmassaker`, denn ein Umlaut ist an dieser Stelle unerwünscht. `Mueckenmassaker` oder, falls es Ihnen aus irgendeinem Grund lieber ist, `Mueckenfang` funktionieren.

Wählen Sie einen passenden Package-Namen, und lassen Sie sich für den Anfang eine Activity namens `MueckenfangActivity` erstellen. Verwenden Sie als BUILD TARGET und MIN SDK VERSION die 8, also Android 2.2, es sei denn, Sie haben ein älteres Smartphone (Abbildung 5.1).

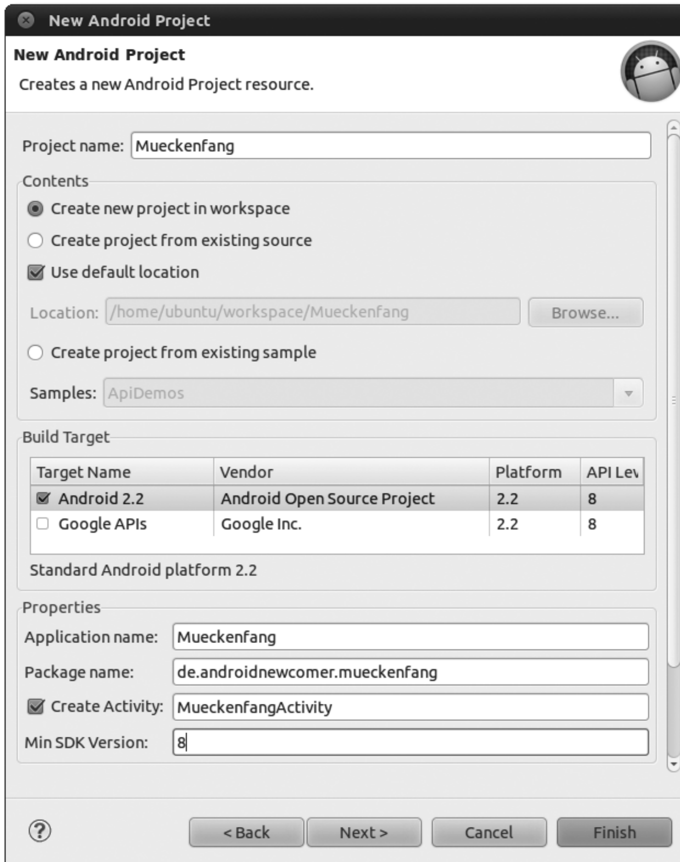

Abbildung 5.1 Der Mückenfang beginnt, bloß ohne Umlaute.

Werfen Sie einen Blick auf die Dateien, die der Wizard erzeugt hat. Wie schon bei Ihrer ersten App finden Sie den Quellcode Ihrer Activity, das Android-Manifest und ein Standard-Icon im *res*-Verzeichnis.

5.1.3 Layouts vorbereiten

Der Android-Wizard hat unter anderem eine Layout-Datei *main.xml* erzeugt. Dieses Layout wird später den Startbildschirm des Spiels definieren.

Für das eigentliche Spiel benötigen Sie ein zweites Layout. Doppelklicken Sie auf *main.xml*, und erzeugen Sie eine Kopie, indem Sie die Datei mit FILE • SAVE AS unter einem neuen Namen speichern: *game.xml*. Das geht schneller als mit dem zugehörigen Wizard, der natürlich ebenfalls zur Verfügung steht.

Löschen Sie die `TextView`, damit die Layouts auf den ersten Blick unterscheidbar sind. Das endgültige Layout werden Sie zu einem späteren Zeitpunkt erstellen.

Öffnen Sie dann erneut das Layout *main.xml*, und fügen Sie einen Button mit der Aufschrift START hinzu (siehe Abschnitt 4.3, »Benutzeroberflächen bauen«). Da Sie dabei ohnehin die *strings.xml* bearbeiten müssen, ändern Sie den »Hello«-Text in »Willkommen beim Mückenfang«.

5.1.4 Die GameActivity

Aus dem Spielkonzept geht hervor, dass Sie zwei Screens benötigen: Da ist zunächst der Hauptbildschirm, der den Spieler begrüßt, ihm vielleicht das Spiel erklärt und natürlich einen START-Button bietet. Verwenden Sie die vom Wizard erzeugte `MueckenfangActivity` als Basis dafür, denn diese Activity ist im Android-Manifest bereits als Start-Activity definiert. Das ist genau das gewünschte Verhalten: Wenn der Spieler das App-Icon antippt, gelangt er in den Hauptbildschirm.

Der zweite Screen wird das eigentliche Spiel darstellen. Dazu benötigen Sie eine zweite Activity, für die sich der Name `GameActivity` anbietet. Drücken Sie `Strg`+`N`, und wählen Sie den Wizard zum Erzeugen einer neuen Java-Klasse (Abbildung 5.2).

Abbildung 5.2 Reichhaltiges Wizard-Angebot ...

Der Wizard zeigt Ihnen einen Dialog, in dem Sie die wichtigsten Eckdaten der gewünschten Klassen eintragen können. Dazu gehört an erster Stelle der Name, in diesem Fall `GameActivity`. Achten Sie darauf, dass der richtige Package-Name vorausgefüllt ist.

Tragen Sie als Basisklasse für Activities immer `android.app.Activity` ein. An diesem Textfeld signalisiert die kleine Glühbirne, dass Eclipse bereit ist, Sie bei der Eingabe zu unterstützen. Tippen Sie »Activity«, gefolgt von Strg + Leertaste, und Sie sparen dank Content Assist Tipparbeit (Abbildung 5.3). Beenden Sie den Wizard mit FINISH.

Abbildung 5.3 Der Wizard erzeugt die »GameActivity« mit wenigen Klicks.

Das Resultat zeigt Ihnen Eclipse sofort an: Den Quellcode Ihrer neuen Activity. Sie hätten diesen Java-Code auch höchstpersönlich eintippen können, und viele Programmierer ziehen das durchaus vor – es ist Geschmackssache. Die Hauptsache ist, dass Sie die verschiedenen Wege kennen, die zu einem bestimmten Resultat führen. Welcher Ihnen am meisten liegt, können Sie selbst entscheiden.

Sorgen Sie nun dafür, dass die `GameActivity` das richtige Layout anzeigt. Am einfachsten bekommen Sie das hin, indem Sie den betreffenden Code aus der `MueckenfangActivity`

übernehmen: Öffnen Sie also diese Klasse, und kopieren Sie die `onCreate()`-Methode hinüber in die `GameActivity`. Übergeben Sie dann anstelle von `R.layout.main` an `setContentView()` `R.layout.game`. Der Code sieht dann wie folgt aus:

```
package de.androidnewcomer.mueckenfang;
import android.app.Activity;
import android.os.Bundle;
public class GameActivity extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.game);
    }
}
```

Im Layout *main.xml* haben Sie bereits einen Button eingefügt, und wie Sie den Klick behandeln, wissen Sie schon aus Abschnitt 4.4.1, »Der OnClickListener«. Fügen Sie also eine Methode `onClickListener()` in die Klasse `MueckenfangActivity` ein, und lassen Sie sie das Interface `OnClickListener` implementieren:

```
public class MueckenfangActivity extends Activity implements
OnClickListener {
   ...
   @Override
   public void onClick(View v) {
   }
}
```

Sorgen Sie nun dafür, dass diese Methode die `GameActivity` startet, indem Sie darin die Methode `startActivity()` aufrufen:

```
startActivity(new Intent(this, GameActivity.class));
```

Das `Intent`-Objekt, das Sie als Parameter übergeben müssen, definiert, welche Klasse Android für die zu startende Activity verwenden soll. Mit solchen `Intent`-Objekten lässt sich noch eine ganze Menge mehr anstellen – ich werde später darauf zurückkommen.

Beachten Sie, dass dieser Aufruf zwar die `GameActivity` startet, aber die aktuelle Activity *nicht beendet*. Sie können daher später mit der Zurück-Taste am Gerät von der `GameActivity` in die `MueckenfangActivity` zurückkehren. Es gibt durchaus die Möglichkeit, eine Activity zu beenden (mit der Methode `finish()`), allerdings ist das Standardverhalten im Sinne unseres Spiels.

Verbinden Sie als Nächstes in der `onCreate()`-Methode den Button mit der Activity (siehe ebenfalls Abschnitt 4.4.1):

```
Button button = (Button) findViewById(R.id.button1);
button.setOnClickListener(this);
```

Der Code der `MueckenfangActivity` sieht damit wie folgt aus:

```
package de.androidnewcomer.mueckenfang;
import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;

public class MueckenfangActivity extends Activity implements
OnClickListener {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        Button button = (Button) findViewById(R.id.button1);
        button.setOnClickListener(this);
    }

    @Override
    public void onClick(View v) {
           startActivity(new Intent(this,GameActivity.class));
    }
}
```

Melden Sie zum Schluss die neue `GameActivity` im Android-Manifest an (siehe Abschnitt 4.2.2, »Activities anmelden«).

Prüfen Sie, ob irgendwo Tippfehler rote Fehler-Symbole verursachen. Wenn nicht, starten Sie die App mit Rechtsklick auf das Projekt und RUN AS... ANDROID APPLICATION – entweder im Emulator oder auf einem angeschlossenen Smartphone.

Bislang ist von einem Spiel nichts zu sehen: Sie können mit dem START-Button zum leeren Game-Screen wechseln und mit der Zurück-Taste wieder zurück. Es wird Zeit, die Schwärze der vorgefertigten Layouts gegen etwas Ansehnlicheres auszutauschen: Grafiken.

5.2 Grafiken einbinden

Die grafischen Elemente der ersten Version der Mückenjagd sind überschaubar: Sie benötigen offensichtlich ein Bild von einer Mücke.

5.2.1 Die Mücke und der Rest der Welt

Je nach Zeichentalent oder Ihrer Geduld, Insekten mit Makroobjektiven aufzulauern, können Sie unterschiedliche Grafiken verwenden. Kommen Sie bitte nicht auf die Idee, das nächstbeste Bild aus der Google-Bildersuche zu kopieren: Jene Bilder sind fast immer urheberrechtlich geschützt. Für die Mückenjagd ist das weniger relevant, weil Sie die kaum in Google Play veröffentlichen werden, aber bei eigenen Spielideen müssen Sie in dieser Hinsicht vorsichtig sein.

Wenn Sie Ihr Zeichentalent ausprobieren wollen, empfehle ich Ihnen das Vektorgrafikprogramm Inkscape, das für alle Betriebssysteme frei verfügbar und auf der Buch-DVD enthalten ist. Mit wenigen Mausklicks zeichnen Sie die wesentlichen Körperteile des fraglichen Insekts (Abbildung 5.4). Speichern Sie die Grafik nicht nur im Vektorformat SVG, sondern exportieren Sie sie auch als PNG in einer Auflösung von etwa 50 × 50 Pixeln. Alternativ verwenden Sie meinen Entwurf, den Sie auf der DVD in *eclipse-projekte/Mueckenfang/verschiedenes* finden. Übrigens empfehle ich Ihnen, für nebenbei anfallende Dateien wie die Vektordatei einen Ordner *verschiedenes* im Projektverzeichnis zu erstellen. Auf diese Weise haben Sie immer alle relevanten Dateien griffbereit.

Speichern Sie die Pixeldatei *muecke.png* (ohne Umlaute!) im Verzeichnis *drawable-mdpi*. Vermutlich müssen Sie in Eclipse einmal das Verzeichnis aktualisieren (F5), damit die PNG-Datei auftaucht. Benennen Sie *drawable-mdpi* in *drawable* um, denn vorerst kümmern wir uns nicht um niedrige oder hohe Bildschirmauflösungen.

Löschen Sie die anderen *drawable*-Verzeichnisse, und löschen Sie außerdem die Datei *icon.png*, die das weiße Standard-Icon enthält. Legen Sie stattdessen eine Kopie der Mücke unter diesem Namen an die gleiche Stelle. Klicken Sie dazu *muecke.png* an, und drücken Sie Strg+C zum Kopieren und Strg+V zum Einfügen. Eclipse bittet Sie automatisch, einen anderen Dateinamen zu vergeben: Geben Sie dann »icon.png« ein.

Mücken zu fangen, macht auf einem nachtschwarzen Bildschirm wenig Spaß. Natürlich könnten Sie auch für den Hintergrund eine Zeichnung verwenden. Alternativ schießen Sie einfach ein Foto mit der ins Handy eingebauten Kamera (hochkant). Skalieren Sie das Foto auf 640 Pixel Breite und 800 oder 854 Pixel Höhe (je nach Seitenverhältnis Ihrer Kamera und Auflösung Ihres Handy-Bildschirms). Speichern Sie das Bild als *hintergrund.jpg* im *drawable*-Verzeichnis.

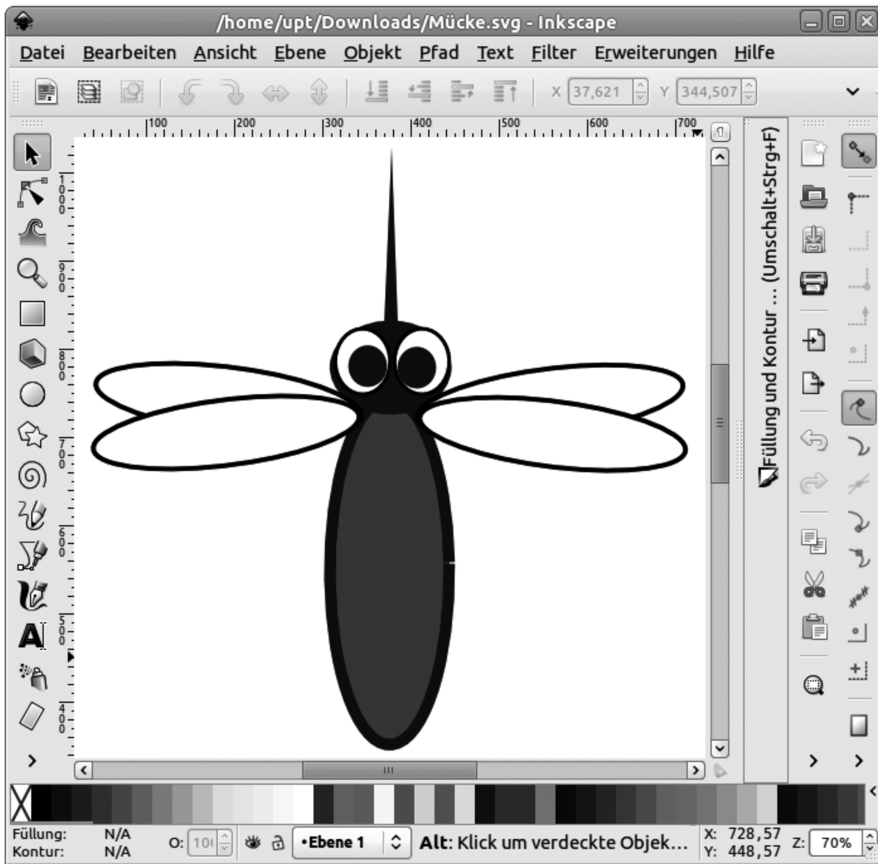

Abbildung 5.4 Entwerfen Sie Ihre Mücke z. B. mit Inkscape, wenn Sie mit dem Makroobjektiv keinen Erfolg haben.

5.2.2 Grafiken einbinden

Nun sind die Hintergrundgrafik sowie die Mücke Teil Ihres Projekts. Im nächsten Schritt werden Sie die Bilder in die Layouts einbauen. Beginnen Sie mit der Datei *main.xml*: Öffnen Sie das Layout, dann klicken Sie mit rechts in den leeren, schwarzen Hintergrundbereich. Wählen Sie im Popup-Menü PROPERTIES • BACKGROUND..., und es erscheint der Reference Chooser, der unter anderem die *drawable*-Ressourcen, die dem Projekt bekannt sind, auflistet (Abbildung 5.5).

In diesem Fall ist HINTERGRUND die offensichtlich richtige Wahl. Fertig: In der Vorschau ist Ihr Foto zu sehen.

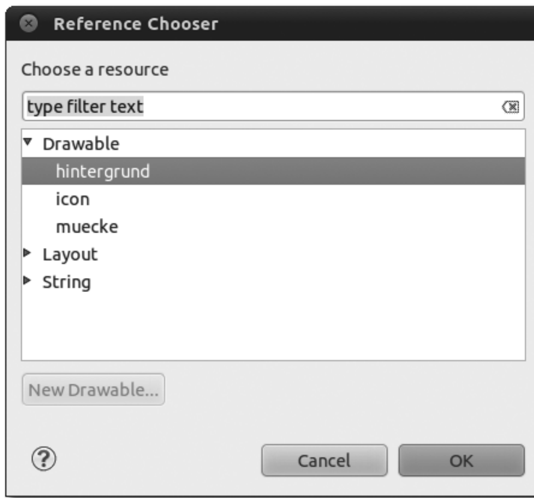

Abbildung 5.5 Im Reference Chooser wählen Sie die richtige Grafik aus.

Fügen Sie als Nächstes aus der Palette eine `ImageView` zwischen `TextView` und START-Button ein. Diesmal präsentiert Ihnen Eclipse automatisch den Resource Chooser, in dem Sie die Mücke auswählen. Verschönern Sie den Bildschirm weiter, indem Sie dem Hintergrund die Gravity `center` befehlen, die Breite der `TextView` auf `wrap_content` ändern und die `TextSize` auf `20sp` erhöhen. Dabei steht `sp` für *scale-independant pixels*. Diese Einheit sorgt dafür, dass die Schrift auf allen Geräten im Verhältnis zu anderen Elementen gleich groß ist – egal, wie hoch die physikalische Auflösung des Bildschirms ist, außerdem wird die vom Benutzer gewählte Font-Größe berücksichtigt. Versehen Sie schließlich die `ImageView` mit einem Padding von etwa `20dp`, um Abstand zwischen der Mücke und den restlichen Elementen zu schaffen.

Der Standardbildschirm hat eine Breite von 320 Bildpunkten, daher ergibt ein Rand von 20 ein brauchbares Erscheinungsbild. Die Einheit `dp` steht für *device independant pixels* und ähnelt der Einheit `sp`, berücksichtigt allerdings nicht die benutzerspezifische Font-Größe. Folglich verwenden Sie `sp` immer bei `TextSize`-Angaben und `dp` bei allen anderen. All diese Einstellungen nehmen Sie entweder über das Kontextmenü oder über die PROPERTIES-View vor.

Experimentieren Sie mit den verschiedenen Einstellungen, bis Ihnen der Screen gefällt (Abbildung 5.6).

Zum Schluss ordnen Sie dem Layout *game.xml* ebenfalls Ihr Hintergrundbild zu. Wenn Sie möchten, können Sie natürlich zwei verschiedene Bilder verwenden, die Sie mit unterschiedlichen Dateinamen versehen.

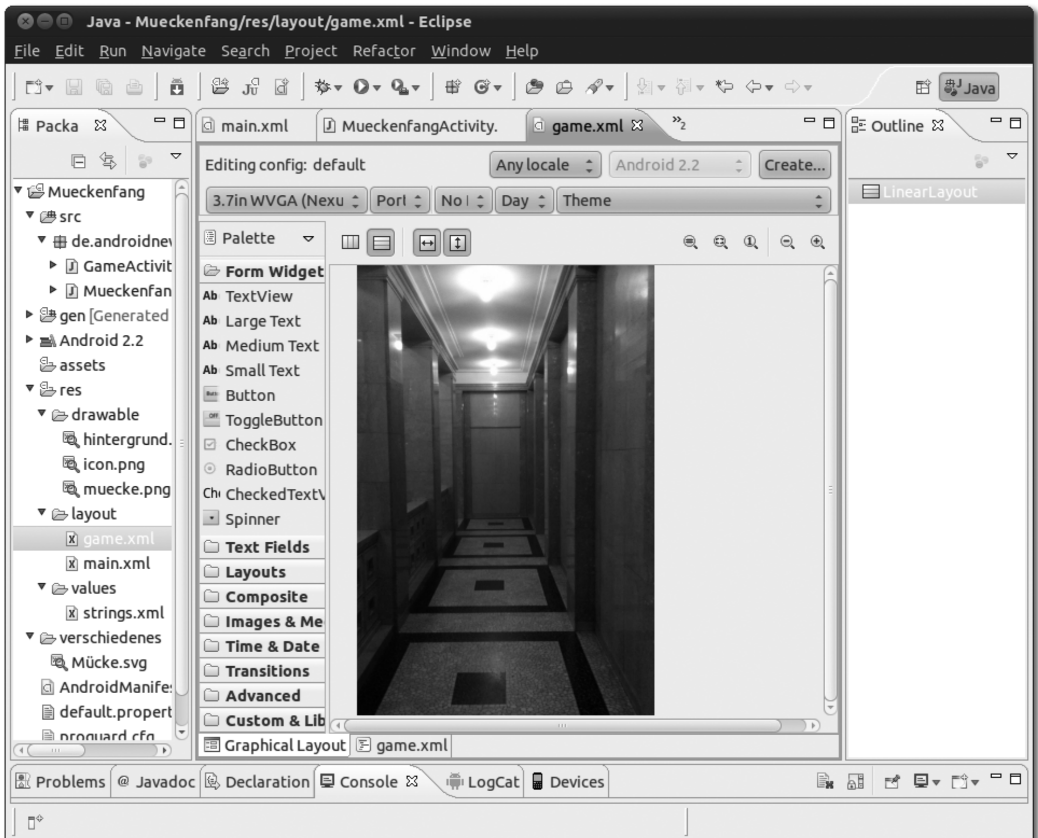

Abbildung 5.6 Basteln Sie bitte ein hübscheres Startscreen-Layout als ich.

Probieren Sie Ihre App auf dem Handy aus, und kochen Sie sich frischen Kaffee oder Tee, bevor die Arbeit am eigentlichen Spiel beginnt.

5.3 Die Game Engine

Vielleicht spielen Sie gelegentlich Brettspiele. Egal, ob Schach, Siedler von Catan oder Monopoly: Alle Spiele haben eine wichtige Gemeinsamkeit, nämlich Spielregeln. Üblicherweise sind sie in Schriftform beigelegt, oder alle Teilnehmer haben sie im Kopf, weil sie übersichtlich und leicht zu behalten sind (im Fall von Schach). Ohne Regeln wäre ein Spiel sinnlos.

Bei Brettspielen sind die Mitspieler selbst dafür zuständig, auf die Einhaltung der Regeln zu achten. Bei Computerspielen funktioniert das nicht, weil ein Teil der Spielfunktionen (manchmal sogar ganze Mitspieler) aus Software bestehen. Deshalb ist es von entscheidender Bedeutung, dass die Spielregeln vollständig in Programmcode vorhanden sind.

Da der Computer üblicherweise auch noch die Darstellung der Spielutensilien übernimmt, ist es nötig, die Regeln und die Bildschirmausgabe zu koppeln. Um all das kümmert sich eine Game Engine.

5.3.1 Aufbau einer Game Engine

Auf den ersten Blick mag es sinnvoll erscheinen, die Game Engine als eigene Java-Klasse zu implementieren. Je nach Komplexität eines Spiels genügt allerdings eine einzige Klasse nicht – Sie benötigen ein ganzes Package. Bei einfachen Android-Spielen wie der Mückenjagd, die noch dazu sehr eng an die grafische Darstellung gebunden sind, ist es oft möglich, die Game Engine in einer `Activity`-Klasse unterzubringen. In unserem Fall wäre das die Klasse `GameActivity`.

Welche Komponenten benötigt eine Game Engine? Überlegen Sie zunächst, welche Attribute nötig sind, um den jeweils aktuellen Zustand des Spiels zu beschreiben:

- Nummer der laufenden Runde (beginnend mit 1)
- Anzahl zu fangender Mücken in der laufenden Runde
- Anzahl schon gefangener Mücken in der laufenden Runde
- verbleibende Zeit für die laufende Runde (zu Beginn jeder Runde, beginnend bei 60 Sekunden)
- Anzahl erzielter Punkte

Halten Sie sich vor Augen, dass Sie vermutlich jedes dieser Attribute in Ihrer Klasse wiederfinden werden.

Schließlich überlegen Sie, welche Methoden erforderlich sind, um Spielereignisse und Aktionen des Spielers auszuführen:

- ein neues Spiel starten
- eine neue Runde starten
- den Bildschirm aktualisieren
- die verbleibende Zeit herunterzählen
- prüfen, ob das Spiel vorbei ist
- eine Mücke anzeigen

- eine Mücke verschwinden lassen
- das Treffen einer Mücke mit dem Finger verarbeiten
- »Game Over«

Sie können sich schon denken, dass für jeden Punkt in dieser Liste eine Methode in Ihrer Game Engine erforderlich ist. Auf den ersten Blick sieht das nach einer ganzen Menge Arbeit aus für ein so einfaches Spiel, und damit liegen Sie nicht ganz falsch.

Bedenken Sie jedoch, dass Ihrem Spielcomputer selbst die simpelsten und selbstverständlichsten Regeln (treffe ich eine Mücke, verschwindet sie, und ich erhalte einen Punkt) fremd sind. Sie müssen jede Kleinigkeit explizit programmieren. Dass Sie sich zum jetzigen Zeitpunkt bereits eine Menge Gedanken gemacht haben, wird Ihnen beim Programmieren viel Zeit sparen. Denn die meisten Programmzeilen werden sich fast von allein ergeben. Lassen Sie uns zunächst aufschlüsseln, was in jeder der Methoden geschehen muss.

5.3.2 Ein neues Spiel starten

Die Methode zum Start eines neuen Spiels wird offensichtlich aufgerufen werden, wenn der Benutzer auf den START-Button drückt. Überlegen Sie, welche Attribute gesetzt werden müssen:

- laufende Runde = 0 (Sie werden gleich sehen, warum 0 und nicht 1)
- Anzahl erzielter Punkte = 0
- eine neue Runde starten

Beachten Sie, dass »eine neue Runde starten« für jede Runde gleichermaßen funktionieren soll. Es gibt keine separate Methode »erste Runde starten«.

Folglich sieht die Methode zum Starten eines neuen Spiels sehr übersichtlich aus:

```
private void spielStarten() {
   spielLaeuft = true;
   runde = 0;
   punkte = 0;
   starteRunde();
}
```

Sie vermissen vielleicht die anderen Attribute. Aber um die kümmert sich die nächste Methode. Überlegen Sie immer genau, an welcher Stelle eine Aktion auszuführen ist. Oft können Sie so redundanten Programmcode vermeiden. Beispielsweise wäre es nicht falsch, in dieser Methode die Anzahl schon gefangener Mücken auf 0 zu setzen. Da dies

aber in jeder Runde geschehen muss und nicht bloß am Anfang des Spiels, genügt es, den Code in die Rundenstart-Methode zu schreiben. Und zu der kommen wir als Nächstes.

5.3.3 Eine Runde starten

Welche Aktionen sind beim Start einer Runde nötig? Beachten Sie, dass diese Methode sowohl für die *erste* als auch für jede weitere Runde funktionieren muss, und zwar möglichst ohne komplizierte Spezialbehandlung:

- Nummer der laufenden Runde um 1 erhöhen (Jetzt verstehen Sie, warum dieses Attribut beim Spielstart auf 0 gesetzt wird, nicht wahr? Halten Sie sich vor Augen, dass diese einfache Zeile dank dieses Tricks in jeder weiteren Runde gleichermaßen funktioniert!)
- Anzahl der zu fangenden Mücken in dieser Runde auf einen bestimmten Wert setzen, der in jeder Runde immer größer wird, Beispiel: 10, 20, 30, ... also das Zehnfache der Nummer der Runde
- Anzahl der schon gefangenen Mücken in dieser Runde = 0
- verbleibende Zeit für die laufende Runde = 60 Sekunden
- den Bildschirm aktualisieren

Auch in dieser Methode finden Sie keine höhere Magie. Je komplizierter ein Spiel ist, umso kniffliger ist es allerdings, sich die richtigen Operationen zu überlegen. Manchmal liegen Sie mit Ihrem ersten Versuch schief. Das macht nichts, denn im Gegensatz zu einem Brettspiel, das Sie vielleicht plötzlich mit Flughäfen anstelle von Bahnhöfen bedrucken müssten, bedarf es nur weniger Änderungen am Programmcode, um ein ganz unterschiedliches Verhalten des Spiels zu erreichen.

Die Methode wird wie folgt aussehen:

```
private void starteRunde() {
   runde = runde +1;
   muecken = runde * 10;
   gefangeneMuecken = 0;
   zeit = 60;
   bildschirmAktualisieren();
}
```

Sie sehen, dass diese Methode die erste ist, die mit dem Bildschirm interagiert. Werfen wir als Nächstes einen genaueren Blick darauf, was der Spieler zu sehen bekommt.

5.3.4 Den Bildschirm aktualisieren

Schließen Sie die Augen (oder starren Sie auf ein leeres Blatt Papier), um sich vorzustellen, wie der Spielbildschirm aussehen soll. Natürlich nimmt die Fläche, auf der die Mücken erscheinen, den größten Raum ein. Davon abgesehen, möchte der Spieler aber ständig einige Informationen sehen können:

- aktuelle Punktzahl
- Nummer der aktuellen Runde
- Anzahl gefangener und noch zu fangender Mücken
- verbleibende Zeit

Entscheiden Sie für jede der Informationen, wie wichtig sie ist und welches der beste Weg ist, sie dem Spieler zu vermitteln. Beispielsweise ist eine numerische Anzeige der verbleibenden Zeit gut und schön, aber im Eifer des Spiels alleine nicht günstig. Viel praktischer ist ein Balken, der immer kürzer wird, bis die Zeit abgelaufen ist.

Ähnliches gilt für die Anzahl zu fangender Mücken: In einem Spiel, in dem es auf Tempo ankommt, sollte der Spieler keine Ziffern ablesen müssen. Wählen Sie also auch hier einen zusätzlichen Balken: Immer wenn eine Mücke gefangen wird, verlängert sich der Balken, bis er bei erfolgreichem Beenden der Runde die volle Bildschirmbreite erreicht hat.

Für den Anfang positionieren wir beide Balken vor unserem geistigen Game-Design-Auge am unteren Bildschirmrand, aber in verschiedenen Farben. Die aktuelle Punktzahl und die laufende Runde können prima am oberen Rand in der linken und der rechten Ecke erscheinen. Aufgabe der Methode wird es also sein, die korrekten Zahlen in die Layout-Elemente einzutragen und die Länge der Balken richtig zu setzen. Schreiten Sie zur Tat, und fügen Sie die nötigen Elemente in das Layout *game.xml* ein.

Derzeit besteht das Layout lediglich aus einem `LinearLayout`-Element mit vertikaler Aufteilung. Das ist ein brauchbarer Ausgangspunkt, aber um das gewünschte Design zu erhalten, müssen Sie weitere Layout-Elemente verschachteln.

Die obere Punkteleiste soll eine Anzeige links und eine rechts enthalten. Das entspricht zwei `TextView`-Elementen, wobei das eine eine linksseitige *Layout Gravity* (nicht *Gravity*!) erhält und das andere eine rechtsseitige. Verwenden Sie ein `FrameLayout`, um die beiden `TextViews` zu umschließen, ohne dass sie einander in die Quere kommen.

Ziehen Sie als Erstes ein `FrameLayout` aus der Palette, und es wird sich am oberen Rand des Bildschirms anordnen. Pflanzen Sie zwei `TextViews` mit großer Schrift (LARGE TEXT) hinein, ändern Sie deren IDs mit dem Kontextmenü EDIT ID... auf `points` bzw. `round`, und setzen Sie bei dem einen das Property LAYOUT GRAVITY auf `right`. Sie müssen

keine Strings für diese `TextView`s erzeugen, denn die richtigen Zahlenwerte schreibt die Methode BILDSCHIRM AKTUALISIEREN später einfach direkt hinein. Setzen Sie schließlich den TEXT STYLE auf `bold`.

Ändern Sie die Farbe des Textes (Property TEXT COLOR), sodass sie zu Ihrem Bildschirmhintergrund passt. Leider müssen Sie zuerst die gewünschten Farben in einer neuen Datei definieren, um sie im Kontextmenü auswählen zu können. Die schnelle Lösung ist an dieser Stelle die PROPERTIES-View. Dort können Sie als TEXT COLOR einfach einen Hexadezimalwert (wie in HTML) eintragen, beispielsweise `#00FF00` für Grün, `#FF0000` für Rot oder `#0000FF` für Blau.

Farbressourcen

Sie werden oft in Apps dieselbe Farbe an mehreren Stellen verwenden wollen. Was geschieht, wenn Sie feststellen, dass Pink doch nicht die richtige Wahl war? Sie müssen in jedem einzelnen Element den Farbwert ändern.

Auf den ersten Blick umständlicher, am Ende aber wesentlich effizienter ist der Weg über eine Datei mit Farbressourcen. Darin definieren Sie Platzhalter für Farben, die von Layout-Elementen referenziert werden. Ändern Sie die Farbe dann nur noch in der Farbendatei, und alle Elemente übernehmen das automatisch.

Erzeugen Sie eine neue Farbendatei, indem Sie `Strg`+`N` drücken und den Wizard NEW ANDROID XML FILE auswählen. Geben Sie darin als Dateinamen *colors.xml* an, und wählen Sie bei TYPE OF RESOURCE bitte VALUES (nicht COLORLIST!).

Mit dem Button ADD können Sie leicht Farben hinzufügen. Jeder Eintrag besteht aus einem Platzhalternamen für eine Farbe und einem HTML-Farbwert. Beachten Sie, dass für Platzhalter die üblichen Regeln gelten: keine Leerzeichen, keine Umlaute oder Sonderzeichen. Lediglich Ziffern und der Unterstrich sind erlaubt. Am besten verwenden Sie nur Kleinbuchstaben (Abbildung 5.7).

Noch ein Wort zu den HTML-Farbcodes: Darin stehen jeweils zwei hexadezimale Ziffern für eine der Grundfarben Rot, Grün und Blau. Ich kann Ihnen an dieser Stelle das hexadezimale Zahlensystem nicht erklären, aber es gibt im Netz (z. B. *http://colorblender.com*) und in Programmen wie Inkscape, GIMP (beide auf der Buch-DVD) oder Photoshop einfache Möglichkeiten, den HTML-Code einer Farbe zu ermitteln.

Grundsätzlich sind auch achtstellige Farbcodes erlaubt. Die beiden zusätzlichen Ziffern stehen vor den anderen sechs und bestimmen die Alpha-Transparenz, wobei `00` für Unsichtbarkeit und `FF` für Sichtbarkeit steht. Mittlere Werte wie `88` erzeugen einen hübschen halbtransparenten Effekt, ohne den heutzutage keine visuelle Präsentation auskommt (sehen Sie nur ein paar Minuten fern, dann wissen Sie, was ich meine).

Sobald Sie die Datei *colors.xml* gespeichert haben, stehen Ihnen die eingetragenen Platzhalter im Resource Chooser zur Auswahl zur Verfügung, etwa wenn Sie ein Property wie TEXT COLOR über das Kontextmenü bearbeiten.

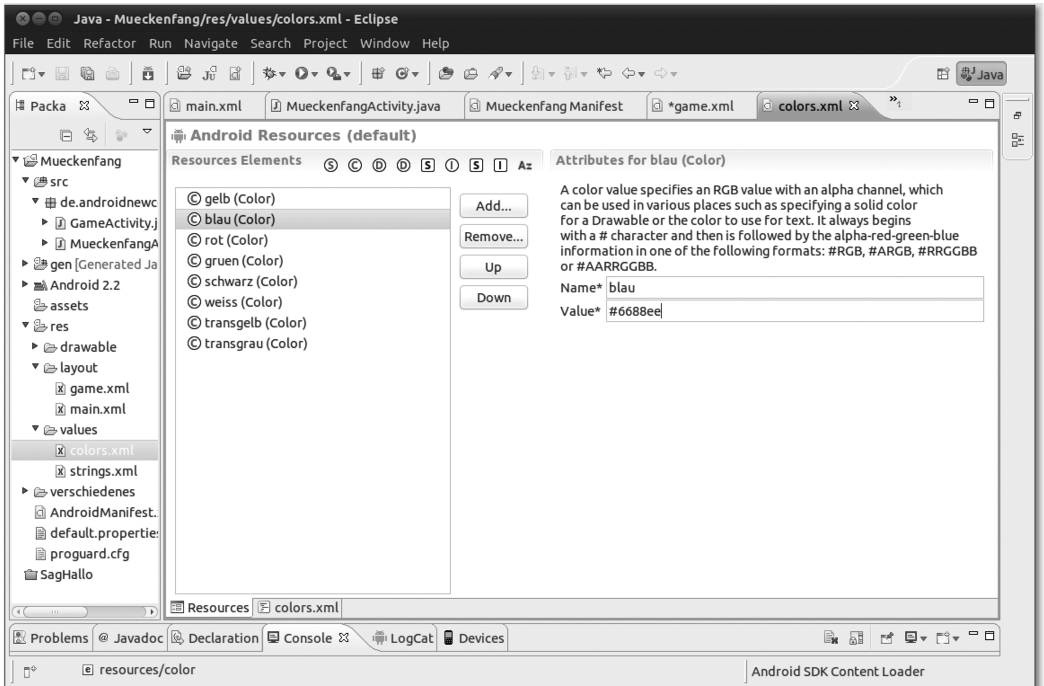

Abbildung 5.7 Erstellen Sie eine Resource-Datei für Farben.

Bevor Sie sich um die Balken kümmern können, die am unteren Ende des Bildschirms erscheinen sollen, steht das eigentliche Spielfeld auf dem Programm, weil das große `LinearLayout`-Element seine Kindelemente vertikal übereinander anordnet. In der Reihenfolge von oben nach unten ist nach den Elementen am oberen Rand das Spielfeld an der Reihe.

Fügen Sie dem Wurzel-`LinearLayout` dazu ein `FrameLayout` aus der Rubrik LAYOUTS hinzu, und stellen Sie das Property LAYOUT WEIGHT auf den Wert 1 (Sie können dazu auch das Icon mit dem stilisierten Porträt verwenden). Verpassen Sie diesem Element die ID `spielbereich`, denn dort werden später die Mücken erscheinen.

Kommen wir also zum unteren Bildschirmbereich. Dort positionieren Sie ein vertikales `LinearLayout`, das alle Balken und deren Beschriftungen übereinander darstellen wird.

Fügen Sie zwei FrameLayouts hinzu. Jeder erhält ein weiteres, inneres `FrameLayout`, das wir als Balken zweckentfremden (man könnte auch ein anderes Element verwenden). Stellen Sie die Properties wie folgt ein:

- Layout Gravity = `center_vertical`
- Layout Width = `50dip` (diese Breite wird später vom Spiel verändert)
- Layout Height = `5dip`
- Background = eine Farbe Ihrer Wahl

Legen Sie als ID des Balkens `bar_hits` bzw. `bar_time` fest.

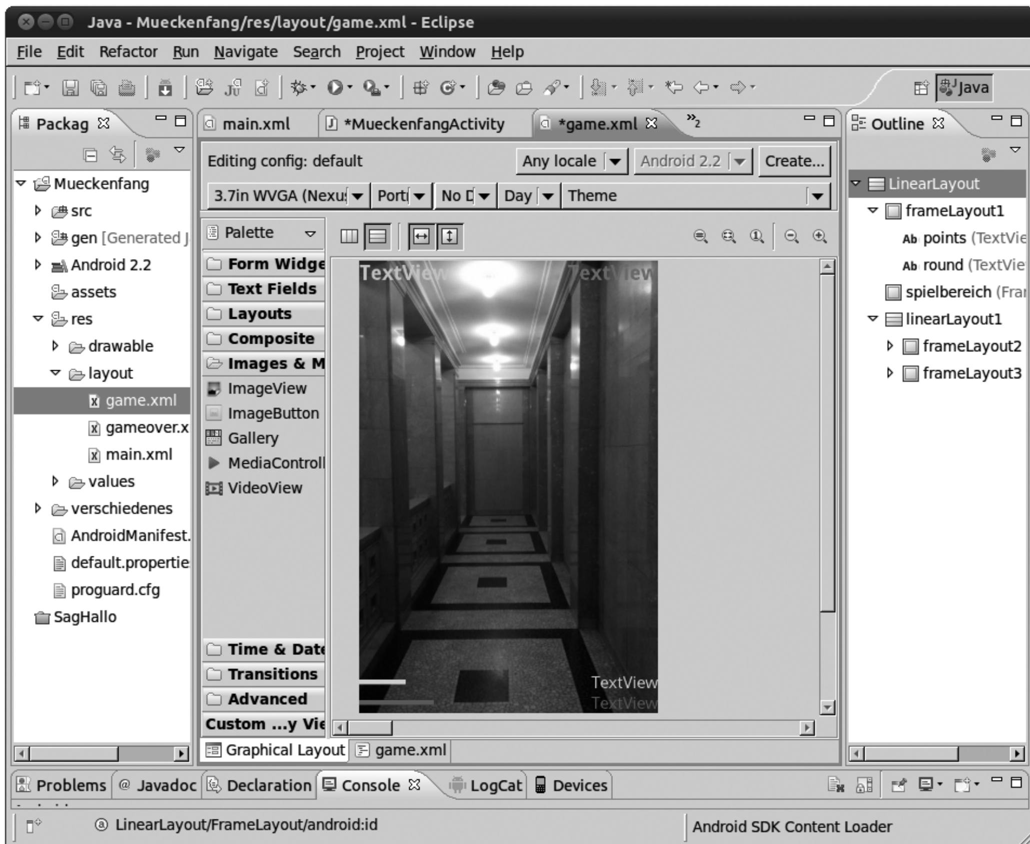

Abbildung 5.8 Achten Sie darauf, dass die Elemente in der »Outline«-View hierarchisch korrekt angeordnet sind.

Fügen Sie schließlich in jedes der beiden FrameLayouts eine `TextView` mit LAYOUT GRAVITY `right` ein, um einen Zahlenwert anzuzeigen. Setzen Sie deren IDs auf `hits` beziehungsweise `time`, und verpassen Sie Ihnen die passende TEXT COLOR.

Puh, geschafft: Wenn Ihr *game.xml*-Layout jetzt in etwa so aussieht wie meines (Abbildung 5.8), können Sie den nächsten Schritt in Angriff nehmen.

Genug Layout-Gefummel; lassen Sie uns die Methode `bildschirmAktualisieren()` schreiben, die die Elemente mit den richtigen Werten füllt. Für die Punktzahl sieht das wie folgt aus:

```
private void bildschirmAktualisieren() {
   TextView tvPunkte = (TextView)findViewById(R.id.points);
   tvPunkte.setText(Integer.toString(punkte));
}
```

Wie üblich holen Sie sich also eine Referenz auf die betreffende View, in diesem Fall eine `TextView`. Deren Objektname (`tvPunktzahl`) ist beliebig; ich setze hier eine Abkürzung der zugehörigen Klasse davor, um nicht einem gleichnamigen Attribut in die Quere zu kommen.

Wichtig ist die explizite Umwandlung des `int`-Attributs `punkte` in einen String, denn eine `TextView` stellt immer Strings dar, selbst wenn die nur aus Ziffern bestehen.

Füllen Sie in derselben Methode die Runde analog:

```
TextView tvRunde = (TextView)findViewById(R.id.round);
tvRunde.setText(Integer.toString(runde));
```

Die Zahl getroffener Mücken und die Restzeit funktionieren auf dieselbe Weise.

Spannender werden die beiden Balken. Holen Sie sich zunächst die zugehörigen Objekte:

```
FrameLayout flTreffer = (FrameLayout)findViewById(R.id.bar_hits);
FrameLayout flZeit = (FrameLayout)findViewById(R.id.bar_time);
```

Offensichtlich besteht die Aufgabe jetzt darin, den beiden FrameLayouts die richtige Breite zu verpassen. Dazu müssen Sie allerdings ein wenig rechnen, denn die Maße sind in Bildschirmpixeln anzugeben, nicht in *device independant pixel* (`dp` oder `dip`) wie im Layout-Editor.

Um die Maße der Balken in Bildschirmpixeln zu ermitteln, müssen Sie die gewünschte `dp`-Breite mit der Pixeldichte des Bildschirms multiplizieren. Sie ermitteln diesen Maßstab aus dem `DisplayMetrics`-Objekt Ihrer App wie folgt:

```
massstab = getResources().getDisplayMetrics().density;
```

Da Sie diesen konstanten Wert an einer anderen Stelle noch einmal benötigen werden, schreiben Sie ihn in der `onCreate()`-Methode in ein `final`-Feld der Activity:

```
private float massstab;
```

Die Breite ist Teil der sogenannten *Layout-Parameter* der View. Holen Sie sich das zugehörige `LayoutParams`-Objekt:

```
LayoutParams lpTreffer = flTreffer.getLayoutParams();
```

Achten Sie beim Organisieren der Imports darauf, dass Sie an dieser Stelle `ViewGroup.LayoutParams` erwischen und keine der anderen existierenden Varianten.

Ändern Sie nun die Breite auf einen geeigneten Wert:

```
lpTreffer.width = Math.round( massstab * 300 *
Math.min( gefangeneMuecken,muecken) / muecken );
```

Die statische Methode `round()` aus der Klasse `Math` liefert als Rückgabewert einen `long`, das Resultat der Multiplikation des `float massstab` mit den anderen Werten ergibt wiederum einen `float`. Das Ergebnis hat immer den genauesten Typ, wenn Sie unterschiedliche Typen in eine Formel stecken. `Math.min()` ermittelt den kleineren der beiden Werte in Klammern und verhindert so, dass der Balken je zu lang wird.

An der Formel in Klammern können Sie ablesen, dass der Balken anfangs die Länge 0 hat (weil `gefangeneMuecken` anfangs 0 ist), und wenn alle oder mehr Mücken gefangen wurden, 300 dip. Da der ganze Bildschirm 320 dip Platz bietet, bleibt rechts noch genügend Raum für die numerische Anzeige.

Zur Erinnerung: Statische Methoden

Methoden, die weder auf Attribute eines Objekts zugreifen noch auf andere Methoden, die das tun, funktionieren unabhängig von der Existenz des Objekts. Allein die Existenz der Klasse genügt.

Sie können die Methode als `static` deklarieren und ohne vorhandenes Objekt verwenden – es genügt, anstelle eines Objektnamens den Klassennamen voranzusetzen. Eclipse kennzeichnet statische Methoden, indem es ihre Namen kursiv schreibt.

Häufigster Einsatzfall für statische Methoden sind Hilfsfunktionen, die mehrere Rechenschritte zusammenfassen und dabei nur auf einfache Eingabewerte angewiesen sind:

```
class Math {
   public static int min(int a, int b) {
      if(a<b) {
         return a;
      }
      return b;
   }
}
```

So ähnlich könnte die Methode `Math.min()` implementiert sein.

Nun zum finalen Countdown: Bei 60 Sekunden Gesamtzeit pro Runde entspricht jede Sekunde 5 dip Balkenbreite (300 : 60 = 5). In Java-Code sieht das dann so aus:

```
LayoutParams lpZeit = flZeit.getLayoutParams();
lpZeit.width = Math.round( massstab * zeit * 300 / 60 );
```

(Ich habe absichtlich 300/60 statt 5 hingeschrieben, Sie werden später sehen, warum.)

Natürlich wollen Sie den Bildschirm nicht nur beim Start einer Runde aktualisieren, sondern mindestens jede Sekunde. Und das bringt uns zum nächsten Thema.

5.3.5 Die verbleibende Zeit herunterzählen

Wir werden später dafür sorgen, dass die Methode zum Herunterzählen der Zeit automatisch einmal pro Sekunde aufgerufen wird. Welche Aktionen muss diese Methode dann durchführen?

- die verbleibende Zeit um 1 verringern
- manchmal eine neue Mücke anzeigen
- falls eine Mücke lange genug angezeigt wurde, die Mücke entfernen
- den Bildschirm aktualisieren
- prüfen, ob die Runde vorbei ist
- prüfen, ob das Spiel vorbei ist

Hier geschieht eine ganze Menge mehr als nur der Ablauf eines Countdowns. Für das Anzeigen und Entfernen einer Mücke könnte man eine eigene Methode erfinden – das

wäre nicht falsch, aber auch jene Methode müsste dann in regelmäßigen Abständen aufgerufen werden. Wir erledigen lieber alles an zentraler Stelle.

Bleiben zwei Details zu klären:

Erstens: Wann verschwindet eine Mücke? Dazu werden wir jeder Mücke ihr »Geburtsdatum« mitgeben, um ihr »Alter« berechnen zu können. Überschreitet sie ein Alter von z. B. zwei Sekunden, fliegt sie mit dem Blut des Spielers hinfort.

Zweitens: das »Manchmal«. Wann genau muss eine neue Mücke erscheinen? Das ist die kniffligere Frage.

Die Mücken sollen nicht in kalkulierbaren Abständen, sondern zufällig erscheinen. Deshalb müssen wir mit Wahrscheinlichkeiten arbeiten. Wie groß ist also in jeder Sekunde die Wahrscheinlichkeit, dass wir eine Mücke anzeigen müssen?

Die Gesamtanzahl Mücken geteilt durch die Dauer der Runde ist die gewünschte Wahrscheinlichkeit.

Überzeugen Sie sich anhand von fünf Beispielen von der Richtigkeit dieser Überlegung:

- Bei zehn Mücken ist die Wahrscheinlichkeit 10/60, das ist ein Sechstel, also 16,7 %. Das entspricht der Wahrscheinlichkeit, mit einem sechsseitigen Würfel eine bestimmte Zahl zu werfen. Probieren Sie es aus: Würfeln Sie 60-mal. Sie werden ungefähr zehnmal sechs Augen werfen. Da unsere Methode jede Sekunde einmal aufgerufen wird, insgesamt also 60-mal, entspricht das genau unserem Ziel.
- Bei 20 Mücken ist die Wahrscheinlichkeit 20/60, also ein Drittel. Bei etwa jedem dritten Aufruf wird unsere Methode also eine Mücke erzeugen, das entspricht bei 60 Aufrufen den gewünschten 20 Stück.
- Bei 60 Mücken ist die Wahrscheinlichkeit 60/60, also 100 %. Bei jedem Aufruf wird eine Mücke erscheinen, macht 60 Stück.
- Bei 90 Mücken ist die Wahrscheinlichkeit 90/60, also 150 %. Wir müssen auf jeden Fall jede Sekunde eine Mücke zeigen und bei jedem zweiten Aufruf eine zweite.

Wie viele Mücken zeigen wir denn nun in jeder Runde an?

Auf diese Frage gibt es unterschiedliche Antworten, die eng mit dem Schwierigkeitsgrad des Spiels verknüpft sind. Nehmen wir zunächst den einfachsten Fall: Die Anzahl der Mücken, die der Spieler in einer Runde treffen muss, ist die vorgegebene Siegbedingung. Wir erlauben ihm Fehlschläge und zeigen daher 50 % mehr Mücken an, weil der Zufallsgenerator sonst manchmal nicht genug erzeugt.

Sie sehen, dass Sie beim Bau einer Game Engine um ein bisschen Mathematik nicht herumkommen. Gerade beim Umgang mit dem Zufallsgenerator, der uns bevorsteht, ist außerdem große Vorsicht geboten, denn ein anständiges Gefühl für Wahrscheinlichkeiten liegt uns nicht in den Genen. Sonst würde nämlich kein Mensch Lotto spielen.

Spezialfälle

Ist Ihnen aufgefallen, dass eine Situation eintreten könnte, in der ein Spieler eine Runde unmöglich gewinnen kann, wenn die Mücken in immer gleichen Zeitintervallen erscheinen? Wurden beispielsweise erst fünf Mücken getroffen, verlangt werden aber insgesamt zehn, und es ist nur noch Zeit, um zwei anzuzeigen, ist die Runde unmöglich zu schaffen. Solche Sonderfälle können bei Spielregeln leicht auftreten, und manchmal dauert es eine Weile, bis man darauf kommt. Je früher man sich eine bessere Regel überlegt, desto besser.

Es gibt zwei Möglichkeiten, mit dem genannten Fall umzugehen:

a) mehr Mücken anzeigen

b) sofortiges »Game Over«

Solche Spezialfälle können Programmcode sehr kompliziert machen, und deshalb werden wir das für den Moment außen vor lassen. Wer nicht genug Mücken trifft, muss die Runde also zu Ende spielen, obwohl er vielleicht keine Chance mehr hat, sie zu gewinnen.

Greifen wir nun zum digitalen Würfel, einem sogenannten *Zufallsgenerator*. Auch dafür bietet Java selbstverständlich eine passende Klasse: `Random`.

Um einen Zufallsgenerator als privates Attribut einer Activity zu erzeugen, schreiben Sie einfach:

```
private Random zufallsgenerator = new Random();
```

Ein solcher Generator ist nicht ganz so zufällig wie die Lottozahlen, aber für die meisten Zwecke ausreichend. Er liefert beispielsweise Kommazahlen von 0 bis 1 (aber nie genau 1):

```
float zufallszahl = zufallsgenerator.nextFloat();
```

Die Wahrscheinlichkeit, dass eine solche Zufallszahl kleiner als ein bestimmter Prozentwert ist, entspricht genau diesem Prozentwert.

Sprich: Die Bedingung (`zufallszahl < 0.5`) trifft auf etwa 50 % der Zufallszahlen zu, (`zufallszahl < 1.0`) immer (100 %) und (`zufallszahl < 0.0`) nie (0 %). Das bedeutet für die neuen Mücken folgende einfache Bedingung:

```
if ( zufallszahl < muecken*1.5/60 ) {
   eineMueckeAnzeigen();
}
```

Die Multiplikation mit 1,5 entspricht der gewünschten Zugabe von 50 %, die Division durch 60 ist die zeitliche Skalierung.

Hieraus können Sie sich ausrechnen, dass die Bedingung ab 60/1,5 gleich 40 Mücken immer erfüllt ist. Ab Runde 4 zeigen wir also schon in jeder Sekunde eine neue Mücke an.

In dem Fall müssen wir also manchmal zwei Mücken erzeugen! Schon wird's relativ kompliziert:

```
double wahrscheinlichkeit = muecken * 1.5f / ZEITSCHEIBEN;
if ( wahrscheinlichkeit > 1 ) {
   eineMueckeAnzeigen();
   if ( zufallszahl < wahrscheinlichkeit - 1 ) {
      eineMueckeAnzeigen();
   }
} else {
   if ( zufallszahl < wahrscheinlichkeit ) {
      eineMueckeAnzeigen();
   }
}
```

Ist die berechnete Wahrscheinlichkeit größer 1, wird zunächst auf jeden Fall eine Mücke angezeigt und dann mit um 1 verringerter Wahrscheinlichkeit eine weitere. Eine Wahrscheinlichkeit von 123 % erzeugt also eine Mücke plus in 23 % der Fälle eine weitere.

Die Methode sieht also summa summarum wie folgt aus:

```
private void zeitHerunterzaehlen() {
   zeit = zeit -1;
   float zufallszahl = zufallsgenerator.nextFloat();
   double wahrscheinlichkeit = muecken * 1.5 / ZEITSCHEIBEN;
   if ( wahrscheinlichkeit > 1 ) {
      eineMueckeAnzeigen();
      if ( zufallszahl < wahrscheinlichkeit - 1 ) {
         eineMueckeAnzeigen();
      }
   } else {
```

```
        if ( zufallszahl < wahrscheinlichkeit ) {
            eineMueckeAnzeigen();
        }
    }
    mueckenVerschwinden();
    bildschirmAktualisieren();
    if(!pruefeSpielende()) {
        pruefeRundenende();
    }
}
```


Ihnen wird auffallen, dass man bei verschachtelter Programmlogik leicht den Überblick verliert. Daher ist es wichtig, die Zeilen passend einzurücken. So können Sie mit einem Blick erkennen, welche Anweisungen zu welchem Programmzweig gehören.

Das Ende der Methode schaut kompliziert aus – daher lenke ich Ihre Aufmerksamkeit zuerst auf die dortigen Zeilen.

5.3.6 Prüfen, ob das Spiel vorbei ist

Die Bedingung für »Game Over« ist ja bekannt: Wenn die Zeit in einer Runde abgelaufen ist und der Spieler nicht die geforderte Anzahl an Mücken erwischt hat, hat er verloren.

Wichtig ist dabei das Wörtchen *und*: Es handelt sich um eine Verknüpfung von zwei Bedingungen. Nur wenn beide erfüllt sind, wird die Methode »Game Over« aufgerufen.

Die Aussagenlogik des Herrn Bool

Im Gegensatz zur Umgangssprache ist die Bedeutung der Wörtchen *und* und *oder* in allen Programmiersprachen klar und einheitlich definiert. Da jede Aussage (»die Zeit ist abgelaufen«) nur zwei Wahrheitswerte annehmen kann (»wahr« und »falsch«), ist es leicht, alle infrage kommenden Kombinationen aufzuschreiben:

`wahr` **und** `wahr` = `wahr`

`wahr` **und** `falsch` = `falsch`

`falsch` **und** `wahr` = `falsch`

`falsch` **und** `falsch` = `falsch`

Bei *oder* sieht die Sache anders aus. Damit eine Oder-Aussage wahr ist, genügt es, wenn *eine der beiden* verknüpften Aussagen wahr ist. Auch wenn beide wahr sind, ist die Gesamtaussage wahr:

`wahr` **oder** `wahr` = `wahr`

`wahr` **oder** `falsch` = `wahr`

`falsch` **oder** `wahr` = `wahr`

`falsch` **oder** `falsch` = `falsch`

Ach ja, und der Vollständigkeit halber erwähne ich auch den *nicht*-Operator:

nicht `wahr` = `falsch`

nicht `falsch` = `wahr`

Die Booleschen Operatoren schreibt man in Java mit doppelten `&`-Zeichen (und), doppelten `|`-Zeichen (oder) und einfachem Ausrufezeichen (nicht):

```
if ( zeit == 0 && gefangeneMuecken < muecken ) ...
```

Wie Sie in Kürze sehen werden, ist es sehr sinnvoll, wenn diese Methode ein Ergebnis zurückgibt:

```
private boolean pruefeSpielende() {
   if ( zeit == 0 && gefangeneMuecken < muecken ) {
      gameOver();
      return true;
   }
   return false;
}
```

Anstelle von `void` wird diese Methode mit dem Rückgabewert `boolean` definiert. Also muss sie auch passende Werte zurückgeben, und dazu dient das Schlüsselwort `return`, das den Ablauf der Methode sofort beendet und den angegebenen Wert an den aufrufenden Code zurückgibt. Falls das Spiel beendet ist, rufen wir also nicht nur die noch zu schreibende Methode `gameOver()` auf, sondern geben auch den Wahrheitswert `true` zurück. Im anderen Fall ist das Ergebnis der Methode `false`. Beachten Sie, dass die letzte Zeile der Methode nicht ausgeführt wird, wenn die `if`-Bedingung erfüllt ist und dass `return true` in den geschweiften Klammern die Methode vorzeitig verlässt.

Jetzt verstehen Sie auch, was am Ende von `zeitHerunterzaehlen()` geschieht:

```
if(!pruefeSpielende()) {
   pruefeRundenende();
}
```

Nur wenn das Spiel nicht beendet ist, wird überhaupt geprüft, ob die Runde zu Ende ist, denn das hätte überhaupt keinen Sinn. Mehr noch: Es wäre falsch, eine neue Runde zu beginnen, denn genau das geschieht in `pruefeRundenende()`.

Kurz ist gut

Sie sehen, dass diese Methode sehr wenig Code enthält. Das ist eine gute Nachricht! Je weniger Programmzeilen eine Methode umfasst, desto weniger Fehler kann sie enthalten, desto leichter verstehen sie andere Programmierer (oder Sie selbst nach einigen Monaten), desto schneller arbeitet die Java Runtime sie ab.

Als nützliche Regel hat sich eingebürgert, dass sich jede Methode *nur um eine Sache kümmern sollte*. In diesem Fall ist das die Prüfung auf das Ende des Spiels. Unterschätzen Sie nicht, wie wichtig es ist, immer den Überblick zu behalten!

Eine weitere Faustformel lautet: Wenn eine Methode nicht vollständig in Ihr Eclipse-Fenster passt, ist sie zu lang. Teilen Sie sie in mehrere Funktionen auf, oder lagern Sie einen Teil des Codes in eine eigene Methode aus – selbst wenn die nur an dieser Stelle verwendet wird.

Eclipse hilft Ihnen übrigens dabei. Wenn Sie Programmcode in eine eigene Methode auslagern wollen, markieren Sie ihn, klicken Sie mit der rechten Maustaste, und wählen Sie Refactor • Extract Method. Daraufhin prüft Eclipse, ob die Auslagerung möglich ist, und erlaubt Ihnen, einen Namen für die neue Methode festzulegen.

Es gibt eine ganze Menge weiterer Unterstützung für Refactoring, also Umbaumaßnahmen am Programmcode, die ich Ihnen jeweils bei passender Gelegenheit vorstellen werde.

5.3.7 Prüfen, ob eine Runde vorbei ist

Die Methode, die das Ende der Runde erkennt, funktioniert ähnlich wie jene, die das Ende des ganzen Spiels erkennt: Wenn die Zeit abgelaufen ist, beginnt eine neue Runde. Der Code sieht auf den ersten Blick sehr einfach aus:

```
private boolean pruefeRundenende() {
   if (zeit == 0) {
      starteRunde();
      return true;
   }
   return false;
}
```

Da wir davon ausgehen, dass diese Methode nur aufgerufen wird, wenn nicht ohnehin das ganze Spiel vorbei ist, genügt die `zeit`-Bedingung. Grundsätzlich wäre es nicht falsch, hier sicherheitshalber zu prüfen, ob der Spieler genug Mücken getroffen hat. Denn möglicherweise bauen Sie irgendwann das Spiel derart um, dass diese Methode auch unter anderen Umständen aufgerufen wird. Wir belassen es aber für den Moment bei der einfachsten Variante, um den Code übersichtlich zu halten.

5.3.8 Eine Mücke anzeigen

Sie haben in das Layout *game.xml* bereits ein `FrameLayout`-Element eingefügt, in dem die Mücken erscheinen sollen. Das `FrameLayout` ist die einfachste Möglichkeit, um Elemente an einer beliebigen Stelle innerhalb eines rechteckigen Bereichs zu positionieren. Jedes `FrameLayout`-Element kann nämlich eine beliebige Anzahl anderer Elemente enthalten, die alle relativ zur linken oberen Ecke des `FrameLayout` ausgerichtet werden. Fügen Sie probeweise im Layout-Editor eine `ImageView` hinzu, und verpassen Sie ihm Ihre Mücke als Drawable. Sie sehen, dass die Mücke links oben erscheint.

Ändern Sie nun den linken Rand (Layout Margin Left) auf `10dip`, und die Mücke rückt ein Stück nach rechts. Ähnlich funktioniert das für die Vertikale, indem Sie den oberen Rand ändern (Layout Margin Top).

Aber Vorsicht: Die Mücken dürfen nicht außerhalb des Bildschirms landen. Sie müssen daher herausfinden, wie breit und wie hoch das `FrameLayout` ist, und die möglichen Werte für die Mückenpositionen entsprechend begrenzen. Der maximale linke Rand einer Mücke entspricht der Breite des `FrameLayout` abzüglich der Breite der Mücke. In der Vertikalen gilt dasselbe, wobei natürlich die Höhe zu berücksichtigen ist, nicht die Breite.

Höhe und Breite können Sie direkt beim Element erfragen. Sie holen sich zunächst das zugehörige `FrameLayout`-Objekt mit `findViewById()`:

```
FrameLayout spielbereich = (FrameLayout)findViewById(R.id.spielbereich);
```

Ermitteln Sie dann Breite und Höhe in Bildschirmpixeln, indem Sie zwei sehr einfache Methoden der Android-Klasse `View` aufrufen:

```
int breite = spielbereich.getWidth();
int hoehe = spielbereich.getHeight();
```

Da `View` eine Elternklasse von `FrameLayout` ist (wie alle sichtbaren Bildschirmelemente), stehen deren `public`-Methoden `getWidth()` und `getHeight()` (unter anderem) auch im Objekt `spielbereich` zur Verfügung.

Um die Maße der Mücke in Bildschirmpixeln zu ermitteln, müssen Sie die Originalmaße Ihrer Grafik noch mit der Pixeldichte des Bildschirms multiplizieren, weil Android alle Bilder automatisch hochskaliert, wenn der Bildschirm mehr als 320 Punkte breit ist.

Der Maßstab ist eine Kommazahl. Im Grunde möchten Sie sicher lieber mit ganzen Zahlen rechnen, also bauen Sie in die Berechnung der Mückengröße gleich eine Rundung mit ein:

```
int muecke_breite = (int) Math.round(massstab*50);
int muecke_hoehe = (int) Math.round(massstab*42);
```

50 und 42 sind die Breite und Höhe meiner Mückengrafik *muecke.png*. Falls Ihre Mücke andere Maße hat, verwenden Sie natürlich Ihre eigenen.

Die statische Methode `round()` aus der Klasse `Math` liefert als Rückgabewert einen `long`. Da wir sicher sein können, dass unsere Werte nie derart groß werden, dass wir irgendwann mit `long`-Werten hantieren müssen, führen wir eine Typumwandlung in `int` durch. Dazu dient der vorangestellte Ausdruck `(int)` – ein *Type-Casting*.

Casting

Keine Schönheiten sind bei dieser Art Casting am Start, sondern unterschiedliche primitive Typen oder Klassen, die einander zugewiesen werden sollen:

```
long a = 100;
int x = (int) a;
```

Sie können nur Typen ineinander umwandeln, wenn sie passen. Versuchen Sie nicht, auf diese Weise einen String in eine `TextView` zu verwandeln – es wird schiefgehen, weil die Ausgangsklassen nicht kompatibel sind.

Unterscheiden muss man hier zwischen:

- Inkompatibilitäten, die schon beim Kompilieren auffallen und demzufolge von Eclipse rot angestrichen werden
- Inkompatibilitäten, die erst zur Laufzeit des Programms auftreten. In dem Fall bricht Java den Ablauf mit einer Exception ab.

Führen Sie nur Castings durch, wenn Sie genau wissen, was Sie tun – oder wenn Sie einen Film drehen wollen.

Da die Mücken an zufälligen Orten erscheinen sollen, benötigen Sie einen Zufallsgenerator, der die nötigen Koordinaten erzeugt. Sie haben bereits einen für das zufällige Erscheinen von Mücken, wozu also einen neuen erzeugen?

Abhängig davon, was für Zufallszahlen Sie gerade benötigen, können Sie unterschiedliche Methoden aufrufen. Für die Position der Mücke benötigen Sie ganze Zahlen zwischen 0 und einem maximalen Wert, nämlich der Breite des Spielfeldes minus der Breite der Mücke (und analog für die Höhe). Verwenden Sie dazu die Methode `nextInt()`:

```
int links = zufallsgenerator.nextInt( breite - muecke_breite );
int oben = zufallsgenerator.nextInt( hoehe - muecke_hoehe );
```

Damit liegen alle sachdienlichen Hinweise zum Positionieren der Mücke vor. Es wird Zeit, das eigentliche grafische Element zu erzeugen: die `ImageView`.

Bisher haben Sie alle Elemente erzeugt, indem Sie sie mithilfe des Layout-Editors in die Datei *game.xml* eingebaut haben. Natürlich tut Android beim Aufbau eines Screens nichts anderes, als anhand Ihrer Angaben bestimmte Objekte zu erzeugen und Attribute zu setzen.

Was Android kann, können Sie schon lange! Also erzeugen Sie eine `ImageView` für die Mücke:

```
ImageView muecke = new ImageView(this);
```

Der Konstruktor der Klasse `ImageView` erwartet als Parameter einen Context. Da jede Activity von der abstrakten Basisklasse `Context` erbt, können Sie einfach `this` übergeben und müssen sich für die Details nicht interessieren.

Verpassen Sie dem Objekt `muecke` nun die richtige Grafik:

```
muecke.setImageResource(R.drawable.muecke);
```

Diese Zeile entspricht der Auswahl des darzustellenden Bildes durch Rechtsklick im Layout-Editor.

Ich verrate Ihnen nicht zu viel aus dem übernächsten Kapitel, wenn ich Ihnen jetzt sage, dass Sie am besten gleich den `OnClickListener` der Mücke setzen, und zwar auf die `GameActivity` selbst, die dazu das `OnClickListener`-Interface implementieren muss:

```
public class GameActivity extends Activity implements OnClickListener
```

Fügen Sie vorerst eine leere `onClick()`-Methode ein, um das `OnClickListener`-Interface zu bedienen:

```
@Override
public void onClick(View v) {
}
```

Verknüpfen Sie die neue Mücke mit dieser Methode:

```
muecke.setOnClickListener(this);
```

Damit reagiert die App auf eine Mückenberührung durch Aufruf der `onClick()`-Methode der Activity.

Fast fertig. Um die Mücke an der gewünschten Stelle anzuzeigen, müssen Sie ein `LayoutParams`-Objekt mit den richtigen Werten füllen. Erzeugen Sie also zunächst eines:

```
FrameLayout.LayoutParams params =
  new FrameLayout.LayoutParams(muecke_breite,muecke_hoehe);
```

Leider gibt es verschiedene Klassen namens `LayoutParams`, die alle in andere Klassen eingebettet sind. Weiter oben haben Sie bereits `LayoutParams` aus `ViewGroup` verwendet, hier benötigen Sie `FrameLayout.LayoutParams`.

Setzen Sie nun den linken und den oberen Abstand, indem Sie die betreffenden Attribute des `LayoutParams`-Objekts füllen:

```
params.leftMargin = links;
params.topMartin = oben;
```

Schließlich setzen Sie die Gravitation auf links oben:

```
params.gravity = 51;
```

Jetzt fragen Sie sich zu Recht: Wie zum Teufel soll ein Mensch auf diese 51 kommen? Die Antwort: gar nicht. Schreiben Sie die Zeile lieber wie folgt:

```
params.gravity = Gravity.TOP + Gravity.LEFT;
```

Diese Version können Sie auf Anhieb lesen, sie macht aber genau dasselbe. Denn um unverständliche Zahlencodes wie die 51 zu vermeiden, definiert Android Konstanten, die Sie stattdessen verwenden wollten. `Gravity.TOP` hat den Wert 48 und `Gravity.LEFT` den Wert 3. Beides müssen Sie sich natürlich nicht merken, denn die Klasse `Gravity` hält ja die simplen Konstanten bereit.

Endlich ist die Zeit gekommen, die Mücke auf den Bildschirm zu verfrachten:

```
spielbereich.addView(muecke,params);
```

Dieser Aufruf der Methode `addView()` fügt dem `FrameLayout spielbereich` die gewünschte `ImageView muecke` mit der Mücke an der zufälligen Stelle hinzu.

Eine Kleinigkeit fehlt noch. Sie erinnern sich, dass wir in Abschnitt 5.3.5, »Die verbleibende Zeit herunterzählen«, beschlossen haben, der Mücke ihr »Geburtsdatum« mitzugeben. Das ist offenbar eine spezielle Anforderung unseres Spiels, daher steht kaum zu erwarten, dass die Klasse `ImageView` über eine Methode `setBirthdate()` verfügt, die wir verwenden können.

Aber die Macher von Android haben unseren Bedarf vorausgeahnt. Sie haben eine Möglichkeit geschaffen, einer View nahezu beliebige Objekte anzukleben, die später wieder ausgelesen werden können: Tags. Sie kennen Tags (Aufkleber, Etiketten) vielleicht von Blogs, Fotoverwaltungen oder anderen Anwendungen.

Um mehrere verschiedene Tags ankleben und später wieder unterscheiden zu können, müssen wir jeweils eine ID definieren. Das darf leider nicht irgendeine Zahl sein, sondern Android fordert einen applikationsspezifischen Wert, den Sie als Ressource definieren müssen.

Legen Sie dazu mit dem Wizard New Android XML File eine neue Resource-Datei namens *ids.xml* im Verzeichnis *values* an. Fügen Sie der leeren Datei ein `Item`-Element hinzu. Geben Sie als Namen »geburtsdatum« ein, und legen Sie als Typ `id` fest.

Diese ID finden Sie dank der Hintergrundarbeit des Android Resource Managers unter dem Bezeichner `R.id.geburtsdatum` wieder. Pappen Sie nun den Geburtsdatum-Aufkleber auf die Mücke:

```
muecke.setTag(R.id.geburtsdatum, new Date());
```

Achten Sie beim Organisieren der Importe darauf, dass Sie `java.util.Date` erwischen. Immer wenn Sie ein Objekt dieser Klasse mit `new` erzeugen, merkt es sich das aktuelle Datum und die Uhrzeit. Später können Sie dieses `Date` mit dem dann aktuellen vergleichen, um zu entscheiden, ob die Mücke verschwinden muss.

Womit wir beim nächsten Thema wären.

5.3.9 Eine Mücke verschwinden lassen

Genau wie das Hinzufügen einer Mücke erfolgt auch das Gegenteil abhängig von einer bestimmtem Bedingung: Wenn eine Mücke lange genug auf dem Bildschirm zu sehen war, soll sie verschwinden.

Formulieren wir diese Bedingung etwas anders, indem wir berücksichtigen, dass jede Mücke ihr eigenes Geburtsdatum kennt: Wenn eine Mücke auf dem Bildschirm ist, deren Geburtsdatum länger zurückliegt als eine bestimmte Zeitspanne, soll sie verschwinden.

Sie sehen, dass die Verschwinden-Methode dazu alle Mücken auf dem Bildschirm in Betracht ziehen muss. Da Sie alle Mücken dem `spielbereich`-Objekt hinzugefügt haben, können Sie sich darauf verlassen, dass es Ihnen alle Mücken liefern kann, ohne dass Sie noch irgendwo sonst eine Liste speichern müssen.

Die Anzahl der Mücken ist beispielsweise:

```
spielbereich.getChildCount()
```


Der Name dieser Methode der Klasse `ViewGroup` (von der `FrameLayout` erbt) spiegelt die hierarchische Struktur der Views wider. Alle Views, die der Klasse `ViewGroup` hinzugefügt wurden, heißen *Kinder* (engl. *children*).

Die Kinder des Spielbereichs (also die Mücken) sind direkt über eine fortlaufende Nummer erreichbar:

```
spielbereich.getChildAt(nummer)
```

Beachten Sie, dass Programmierer fast immer bei 0 anfangen zu zählen, nicht bei 1, wie es die Sesamstraße lehrt. Die erste Mücke bekommen Sie also mit:

```
spielbereich.getChildAt(0)
```

Die Nummer der letzten ist folglich die Anzahl minus 1. Sie ahnen vielleicht schon, dass Sie eine Laufvariable vom Typ `int` benötigen, um alle Kinder des Spielbereichs zu erwischen. Diese Variable wird bei 0 beginnen und als Höchstwert die Anzahl minus 1 annehmen.

Um alle Mücken der Reihe nach zu betrachten, benötigen wir eine Programmierstrategie, um gewisse Codezeilen (die Altersprüfung) mehrfach zu durchlaufen. Ein solches Konstrukt heißt *Schleife* (engl. *loop*). Es gibt verschiedene Möglichkeiten, die gewünschte Schleife zu programmieren. Wir benutzen in diesem Fall das folgende Schlüsselwort:

```
while(bedingung) {
   ...
}
```

In die runden Klammern schreiben Sie eine Bedingung. Solange diese Bedingung erfüllt ist (also den Wert `true` hat), wird der Code, der in den geschweiften Klammern steht, ausgeführt. Immer, wenn diese Kommandos ausgeführt wurden, kehrt die Ausführung zum `while` zurück und prüft erneut die Bedingung. Ist sie irgendwann `false`, wird der Schleifencode nicht noch einmal durchlaufen, sondern die Ausführung setzt dahinter wieder ein.

Schauen Sie sich die Schleife in Ruhe an:

```
int nummer = 0;
while(nummer < spielbereich.getChildCount() ) {
   ImageView muecke = (ImageView) spielbereich.getChildAt(nummer);
   nummer = nummer+1;
}
```

Achten Sie vor allem auf die `while`-Bedingung. Halten Sie sich vor Augen, dass sie, wie gewünscht, funktioniert, indem Sie im Kopf durchspielen, was passiert. Zunächst hat die Laufvariable `nummer` den Wert 0. Das ist kleiner als die Anzahl der Mücken, es sei denn, es gibt gerade keine. In dem Fall hat die Bedingung den Wert `false`, und die Schleife wird kein einziges Mal durchlaufen.

Gibt es Mücken, wird der Schleifencode ausgeführt. Die Methode `getChildAt(nummer)` wird die erste Mücke zurückgeben. Ähnlich wie schon bei der Verwendung von `findViewById()` müssen Sie auch hier den Rückgabewert explizit in eine `ImageView` umwandeln, weil Java an dieser Stelle nicht weiß, worum es sich bei dem Kind genau handelt – Sie schon.

Als derzeit letzte Zeile innerhalb der Schleife wird die Laufvariable um 1 erhöht. Zwar sieht diese Zeile aus Sicht eines Mathematikers fürchterlich falsch aus, aber wie Sie wissen, bedeutet das Zeichen `=` hier eine Zuweisung, keinen Vergleich.

Abkürzungen

Eine Zeile wie die folgende lässt sich auf mehrere Arten schreiben:

```
nummer = nummer +1;
```

Da Programmierer bekanntlich faul sind, haben sie meist keine Lust, den Bezeichner zweimal hinzuschreiben, und tippen nur:

```
nummer += 1;
```

Analog existieren auch Operatoren zum Subtrahieren, Multiplizieren und Dividieren:

```
nummer -= 1;
```

```
nummer *= 2;
```

```
nummer /= 2;
```

Und auch das ist noch nicht kurz genug. Wenn eine `int`-Variable genau um 1 erhöht werden soll, können Sie auch schreiben:

```
nummer++;
```

Hierzu allerdings eine Warnung: Kürzerer Code ist nicht immer übersichtlicher. Gerade der ++-Operator wird gerne dermaßen in weitere Operationen verstrickt, dass man gewisse Nebenwirkungen übersieht. Wenn Sie Operatoren wie ++ oder -- verwenden möchten, schreiben Sie sie sicherheitshalber in eine eigene Zeile, es sei denn, Sie wissen, was Sie tun.

Die Schleife zum Durchlaufen aller Mücken ist jetzt bereit. Fehlt also nur die Altersprüfung. Ermitteln Sie zunächst das Geburtsdatum:

```
Date geburtsdatum = (Date) muecke.getTag(R.id.geburtsdatum);
```

Um jetzt nicht mit Tag, Monat, Jahr, Sekunden, Minuten, Stunden, Sommerzeit und Zeitzonen hantieren zu müssen (ja, so kompliziert ist unsere Zeitrechnung!), machen wir es uns einfach: Wir verwenden eine simple, wenngleich große Zahl, nämlich die Anzahl der Millisekunden seit dem 1.1.1970. Diesen auf den ersten Blick sinnlosen Wert können Sie leicht ermitteln, weil er ohnehin intern von Java zur Zeitrechnung eingesetzt wird:

```
geburtsdatum.getTime()
```

Subtrahieren Sie diesen Wert von der aktuellen Zeit, erhalten Sie das Alter der Mücke in Millisekunden:

```
long alter = (new Date()).getTime() - geburtsdatum.getTime();
```

Jetzt können Sie sehr leicht prüfen, ob die Mücke zu alt ist und entfernt werden muss:

```
if(alter > HOECHSTALTER_MS) {
   ...
}
```

Sie sehen, dass ich `HOECHSTALTER_MS` in Großbuchstaben geschrieben habe, wie es typisch für Konstanten ist. Denn diesen Wert möchten Sie möglicherweise irgendwann einmal ändern, und dann wäre es ungünstig, irgendwo mitten im Code nach der betreffenden Zahl zu suchen. Die Konstante dagegen steht ganz oben in der Klasse und ist daher leicht zu finden. Nicht nur das: Der Name verrät ihre Bedeutung, in diesem Fall inklusive der Einheit. Denken Sie sich das `_MS` einmal weg: Erinnern Sie sich in ein paar Monaten noch daran, dass Sie hier einen Wert in Millisekunden und nicht in Sekunden oder Jahren eintragen müssen?

Setzen Sie die Konstante zunächst auf zwei Sekunden:

```
private static final long HOECHSTALTER_MS = 2000;
```

Und was ist zu tun, wenn das Alter zu groß ist? Die View `muecke` muss vom `spielbereich` entfernt werden:

```
spielbereich.removeView(muecke);
```

Allerdings offenbart diese Anweisung eine kleine Lücke im bisherigen Code. Denn durch das Löschen einer View aus dem Spielbereich ändert sich sofort die Anzahl ihrer Kinder! Nicht nur das: Falls es eine weitere Mücke gibt, rückt die in der numerischen Liste eins nach vorn. Würden wir jetzt einfach die Laufvariable `nummer` um 1 erhöhen, würde der nächste Schleifendurchlauf die *übernächste* Mücke erwischen und damit eine übersehen! Sie dürfen also `nummer` nur erhöhen, wenn Sie die aktuelle Mücke nicht entfernt haben, wenn sie also nicht alt genug war. Das ist genau der richtige Moment, um Ihnen das Schlüsselwort `else` näherzubringen:

```
if(alter > HOECHSTALTER_MS) {
   spielbereich.removeView(muecke);
} else {
   nummer++;
}
```

Hinter dem `else` steht ein weiterer durch geschweifte Klammern begrenzter Codeblock. Dieser wird genau dann und *nur* dann ausgeführt, wenn die `if`-Bedingung nicht wahr ist, sondern falsch.

5.3.10 Das Treffen einer Mücke mit dem Finger verarbeiten

Sie haben allen Mücken bereits den richtigen `OnClickListener` verpasst und eine leere Methode `onClick()` geschrieben. Nun gilt es, auf den Fingerdruck zu reagieren.

Überlegen Sie, was beim Treffen einer Mücke alles geschehen muss:

- Anzahl getroffener Mücken um 1 erhöhen
- Punktzahl erhöhen
- den Bildschirm aktualisieren
- die Mücke entfernern

Komplikationen sind hier nicht in Sicht – Sie wissen schon ganz genau, wie Sie das alles bewerkstelligen können. Vermutlich haben Sie die nötigen vier Zeilen schon eingetippt,

während Sie dies hier lesen. Daher behellige ich Sie nur mit einem kleinen Hinweis: Spieler lieben es, viele Punkte zu bekommen. Deshalb lieben sie Flipperautomaten. Kommen Sie also nicht auf die Idee, für jede Mücke bloß einen Punkt zu verteilen – geben Sie dem Spieler gleich 100. Weder programmiertechnisch noch spieltechnisch macht es einen Unterschied – aber ein Highscore von 6.700 klingt einfach viel besser als 67!

Auch das Entfernen von Mücken kennen Sie schon – und da der Methode `onClick()` die angeklickte View (also die Mücke) freundlicherweise als Parameter übergeben wird, sieht das Resultat wirklich übersichtlich aus:

```
public void onClick(View muecke) {
  gefangeneMuecken++;
  punkte += 100;
  bildschirmAktualisieren();
  spielbereich.removeView(muecke);
}
```

5.3.11 »Game Over«

Das war's! Aus und vorbei! Vor allem bedeutet das: Es dürften keine Mücken mehr erscheinen. Der weitere Aufruf der Countdown-Methode muss also unterbleiben. Außerdem zeigen wir dem Spieler den Schriftzug »Game Over« an. Wenn er drauftippt, schicken wir ihn zurück zum Hauptbildschirm.

Den Schriftzug werden wir mit einem Dialog realisieren. Das sind grafische Elemente, die im Vordergrund des aktuellen Bildschirms eingeblendet werden. Gleichzeitig setzen sie alle Funktionen, die sie verdecken, außer Kraft.

Ein solcher Dialog basiert auf einer Layout-Datei. Im vorliegenden Fall ist die nicht sonderlich kompliziert: Sie besteht aus einem halbtransparenten Hintergrund und einem Game-Over-Schriftzug.

Legen Sie also ein neues Layout unter dem Namen *gameover.xml* an. Drücken Sie [Strg]+[N], und wählen Sie den Wizard für eine neue Android-XML-Datei. Tragen Sie den Dateinamen *gameover.xml* ein, wählen Sie LAYOUT als Resource-Typ und unten im Wizard FRAMELAYOUT als Wurzelelement.

Ändern Sie das Property BACKGROUND des `FrameLayout`, um es halbdurchsichtig zu machen. Ein passender Farbcode ist `#88888888`.

Fügen Sie dem `FrameLayout` eine `TextView` hinzu, der Sie einen neuen String mit dem Inhalt »Game Over« verpassen. Setzen Sie LAYOUT GRAVITY auf `center`, setzen Sie die

Schriftgröße auf 16sp, wählen Sie eine hübsche TEXT COLOR, und setzen Sie den TEXT STYLE auf `bold` (Abbildung 5.9).

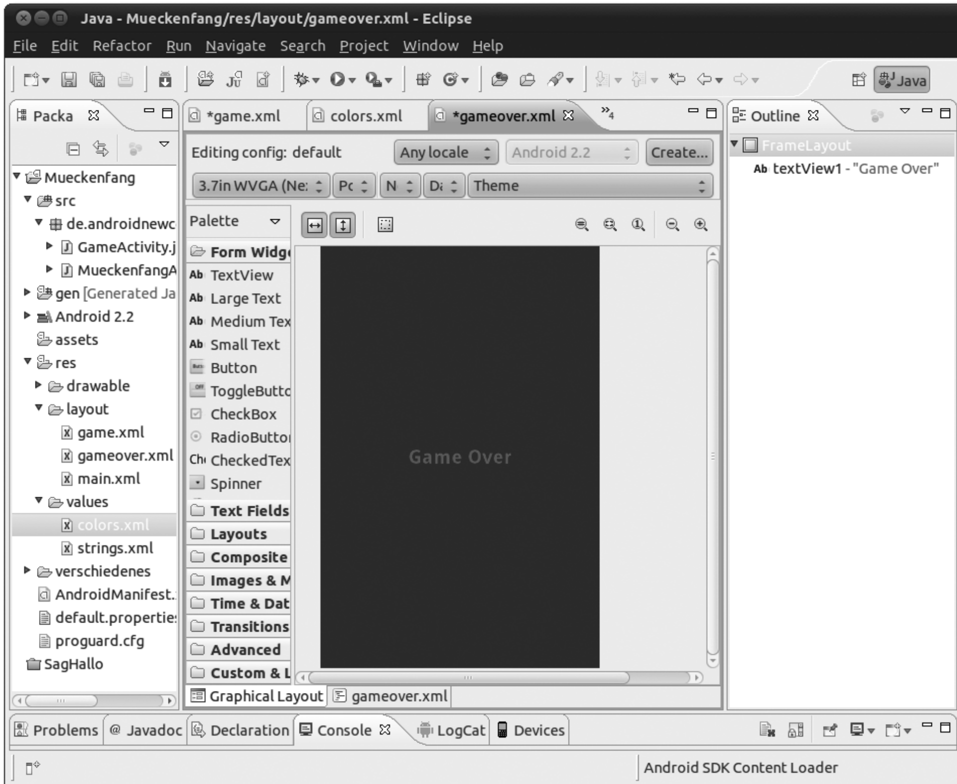

Abbildung 5.9 Die berühmten letzten Worte

Um diesen Dialog anzuzeigen, erzeugen Sie in der Methode `gameOver()` zunächst ein passendes `Dialog`-Objekt:

```
Dialog dialog = new Dialog(this, android.R.style.
  Theme_Translucent_NoTitleBar_Fullscreen);
```

Dabei müssen Sie einen Style angeben; in diesem Fall wählen wir einen ohne irgendwelche Ausschmückungen, die nur stören würden.

Damit dieser Dialog das richtige Layout anzeigt, verwenden Sie die altbekannte Methode `setContentView()`, nur eben nicht in Ihrer Activity, sondern im Dialog:

```
dialog.setContentView(R.layout.gameover);
```

Schließlich zeigen Sie den Dialog einfach an:

```
dialog.show();
```

Viel mehr über Dialoge erfahren Sie in Abschnitt 10.3, »Dialoge«.

Sie haben sicher bemerkt, dass ich Ihnen noch nicht verraten habe, wie Sie das Erscheinen weiterer Mücken unterbinden. Das hängt hauptsächlich damit zusammen, dass ich Ihnen auch noch nicht erklärt habe, wie Sie welche erscheinen lassen. Denn die zentrale Methode `zeitHerunterzaehlen()` wird ja überhaupt noch nicht aufgerufen.

Sie ahnen es schon: Dazu kommen wir als Nächstes.

5.3.12 Der Handler

Nein, in dieser Überschrift fehlen keine Pünktchen auf dem a. Gemeint ist der englische Begriff *Handler*, für den es keine Übersetzung gibt, die verständlich machen würde, worum es sich eigentlich handelt. Deshalb müssen Sie sich wohl oder übel einfach den Begriff merken, und ich erkläre Ihnen jetzt, welches Mysterium sich dahinter verbirgt.

Halten Sie sich vor Augen, dass auf Ihrem Handy eine Menge Dinge quasi gleichzeitig passieren: Sie spielen ein Spiel, es erscheint eine neue E-Mail für Sie, die Bildschirmhelligkeit wird automatisch angepasst, und manchmal klingelt das Gerät sogar, weil Sie jemand anruft. Was aber tut das Android-System dabei am meisten? Warten.

Denn nicht nur Tastendrücke werden ereignisorientiert verarbeitet, sondern im Grunde alles. Das System wartet, bis etwas geschieht, und erledigt dann die zugehörige Aufgabe, um anschließend mit dem Warten fortzufahren.

In der Steinzeit der Computerprogrammierung (also vor 20 bis 30 Jahren) sah die Lage meistens noch anders aus: Sobald man ein Programm startete, lief es, bis es fertig war, und kein anderes konnte währenddessen etwas anderes tun (ausgenommen solche mit Sondererlaubnis). Wenn ein Programm auf eine Benutzereingabe wartete, tat es das in einer tumben Endlosschleife. Während dieser nutzlos verschwendeten Zeit durfte kein anderes Programm aktiv werden. Dieses *Single Processing* ist längst auf dem Müllhaufen der Geschichte gelandet, genau wie Dampflokomotiven, Röhrenfernseher und Wählscheibentelefone.

Praktisch alle Systeme, egal, ob Windows, Linux, Mac oder Android, erlauben heutzutage Multitasking. Dabei dürfen beliebig viele Programme (*Tasks* oder *Processes*) gleichzeitig laufen (siehe Abbildung 5.10) – indem sie die ganze Zeit fast nichts anderes tun, als zu *warten*.

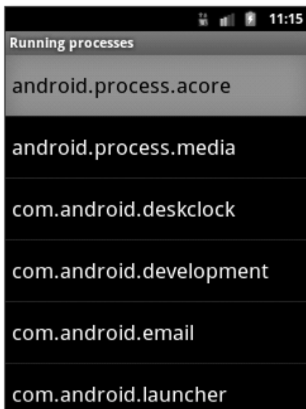

Abbildung 5.10 Die Dev Tools – eine App, die im Emulator vorinstalliert ist – zeigen Ihnen die laufenden Prozesse. Es gibt auch eine Reihe Apps im Market, die das können.

In Android gibt es für jede App eine *Ereigniswarteschlange* (engl. *eventqueue*). Wenn mindestens ein *Ereignis* (engl. *event*) in der Warteschlange steht, für das eine App zuständig ist, erledigt sie baldmöglichst die damit verbundenen Aufgaben und schnappt sich das nächste Ereignis, falls vorhanden. Die Reaktion auf das Antippen des Bildschirms (`onClick()`) ist ein Beispiel dafür: Ihre App führt ein paar Programmzeilen aus und gibt die Kontrolle wieder ab. So wird keine Rechenzeit verschwendet.

Es ist wichtig, dass die Verarbeitung schnell geschieht, weil die App erst danach weitere Aufgaben erledigen kann, die unterdessen in der Warteschlange gelandet sind. Bildlich gesprochen: Ihre App kann nicht ohne Weiteres an mehreren Stellen gleichzeitig sein.

Wenn Sie eine Methode schreiben würden, die eine mehrere Sekunden dauernde Berechnung durchführt, dann halten Sie sich vor Augen, dass in dieser Zeit das Antippen von Bedienungselementen keine sichtbare Wirkung hat! Das ist ein derart unerwünschtes Verhalten, dass Android es gnadenlos bestraft. Sie haben vielleicht schon diese Dialoge gesehen, die Ihnen erklären, dass eine App nicht antwortet. Ich bin sicher, in solchen Fällen haben Sie schon oft entnervt den BEENDEN-Button angeklickt. Recht so: Es ist die verdiente Strafe für eine benutzerunfreundliche Programmierung.

Bis hierher habe ich Ihnen erklärt, dass Sie nicht einfach schreiben können:

```
while(!spielZuende) {
   zeitHerunterzaehlen();
   warteEineSekunde();
}
```

Dies ist nämlich ein Beispiel für *blockierende* Programmierung anno 1985.

Die Antwort auf die Frage, wie man so etwas denn nun im 21. Jahrhundert löst, führt uns zurück zum Handler. Der Handler erlaubt den Zugriff auf die aktuelle Ereigniswarteschlange Ihrer App. Sie können selbst Ereignisse erzeugen und in die Warteschlange stellen. Und der Clou an der Sache ist: Dabei können Sie eine zeitliche Verzögerung angeben! Auf diese Weise erzeugen Sie Ereignisse, die erst nach einer gewissen Zeit ausgeführt werden – z. B. den Aufruf von `zeitHerunterzaehlen()` nach einer Sekunde.

Und damit haben wir alle Bausteine zusammen:

- beim Start der Runde ein `zeitHerunterzaehlen()`-Ereignis in die Warteschlange stellen (mit einer Sekunde Verzögerung)
- am Ende von `zeitHerunterzaehlen()` ein weiteres `zeitHerunterzaehlen()`-Ereignis in die Warteschlange stellen (mit einer Sekunde Verzögerung), wenn nicht Runde oder Spiel zu Ende sind

Wie sieht nun das Ereignis genau aus?

Lassen Sie uns ein Ereignis bauen, das einen Verweis auf den auszuführenden Programmcode enthält (nämlich `zeitHerunterzaehlen()`). Das Stichwort für diese elegante Lösung heißt *Runnable*. Dahinter verbirgt sich zunächst einmal lediglich ein sehr einfaches Java-Interface:

```
public interface Runnable {
   public void run();
}
```

Klassen, die dieses Interface implementieren, müssen also eine Methode namens `run()` besitzen. Damit wird die Klasse *ausführbar* (engl. *runnable*). Wenn Sie Ihrer Activity dieses Interface (und die `run()`-Methode) verpassen, können Sie mit dem Handler ein Ereignis erzeugen, das nach einer Sekunde genau diese `run()`-Methode aufruft.

Erzeugen Sie aber zuerst ein Attribut mit dem Handler selbst, und zwar gleich bei der Deklaration, weil wir in der Activity immer nur genau eine Instanz benötigen:

```
private Handler handler = new Handler();
```

Lassen Sie die `GameActivity` das Interface `Runnable` implementieren:

```
public class GameActivity extends Activity implements OnClickListener,Runnable;
```

Verwenden Sie den Handler, um am Ende von `starteRunde()` das Ereignis verzögert in die Warteschlange zu stellen:

```
private void starteRunde() {
   ...
   handler.postDelayed(this,1000);
}
```

Dabei verwenden Sie die Methode `postDelayed()` und übergeben ihr ein Runnable (nämlich die eigene Activity). Um was für eine Klasse es sich genau handelt, ist der Methode piepegal: Sie interessiert sich nur für das Runnable, und sie wird auch nichts anderes tun, als dafür zu sorgen, dass die Methode `run()` aufgerufen wird. Das ähnelt dem Bestellen beim Pizzataxi: Welcher Pizzabäcker sich darum kümmert und wie er aussieht, ist relativ egal – Hauptsache, er liefert.

Der zweite Parameter ist die gewünschte Verzögerung, in diesem Fall 1.000 Millisekunden, also eine Sekunde.

Schreiben Sie nun die simple Methode `run()`:

```
@Override
public void run() {
   zeitHerunterzaehlen();
}
```

Diese Methode muss `public` sein, denn sie wird von außen (durch die Ereignisverwaltung) aufgerufen. In der Methode rufen Sie natürlich `zeitHerunterzaehlen()` auf, und an deren Ende erzeugen Sie das nächste Ereignis:

```
private void zeitHerunterzaehlen() {
   ...
   handler.postDelayed(this, 1000);
}
```

Halten Sie sich vor Augen, was geschieht: Beim Start der Runde wird ein Ereignis in die Warteschlange gestellt, das nach einer Sekunde verarbeitet wird und zum Aufruf Ihrer Methode `run()` in der Activity führt. Dann wird die Game Engine aktiv und zählt die Zeit herunter – mit allem, was dazugehört. Schließlich erzeugt sie das nächste Ereignis, und eine Sekunde später passiert dasselbe noch mal.

Irgendwann aber muss die schönste Endlosschleife beendet werden, sonst würden immer mehr Mücken erscheinen, obwohl das Spiel längst vorbei ist. Die betreffenden Bedingungen kennen Sie bereits, und auch die Prüfmethoden sind schon darauf vorbereitet. Sorgen Sie also dafür, dass nur dann ein weiteres Ereignis erzeugt wird, wenn nicht Spiel oder Runde beendet sind, indem Sie den `postDelayed()`-Aufruf verschieben:

```
if(!pruefeSpielende()) {
   if(!pruefeRundenende()) {
      handler.postDelayed(this, 1000);
   }
}
```


Diese ineinander verschachtelten `if`-Bedingungen sorgen für das gewünschte Verhalten: Wenn das Spiel nicht zu Ende ist, wird geprüft, ob die Runde zu Ende ist, und nur wenn auch das nicht der Fall ist, geht das Spiel weiter mit dem nächsten Ereignis.

Sie könnten diese Bedingungen auch in einer Zeile schreiben, aber ich finde die obige Variante übersichtlicher. Entscheiden Sie selbst, hier zum Vergleich die Alternative:

```
if(!pruefeSpielende() && !pruefeRundenende()) {
   handler.postDelayed(this, 1000);
}
```

5.4 Der erste Mückenfang

Haben Sie den Code eingetippt und nachvollzogen? Alle eventuellen Tippfehler beseitigt? Wenn nicht, macht das auch nichts: Sie finden das fertige Projekt natürlich auf der Buch-DVD im dortigen *eclipse-projekte*-Verzeichnis.

Es wird Zeit, das Spiel zu starten (Abbildung 5.11)!

Abbildung 5.11 Möge die Jagd beginnen.

5.4.1 Retrospektive

Die ersten paar Spielrunden werden Ihnen eine ganze Reihe an Erkenntnissen bringen. Auf dem Emulator werden Sie ziemlich schnell feststellen, dass Sie mit der Maus relativ langsam sind. Hier zeigt sich der Unterschied zwischen den Welten: PC-Spiele müssen sinnvoll mit der Maus bedienbar sein, Android-Spiele mit Touchscreen und Konsolenspiele möglichst mit dem mitgelieferten Controller. Berücksichtigen Sie das, wenn Sie sich ein Spielkonzept überlegen.

Im Gegensatz zu anderen Apps kommt es bei Spielen nicht nur darauf an, ob sie fehlerfrei funktionieren. Sie müssen außerdem weitere Kriterien erfüllen:

- Spiele dürfen nicht zu schwer sein: Sie müssen leicht beginnen und langsam schwieriger werden.
- Spiele müssen sofort verständlich sein. Niemand nimmt sich die Zeit, eine Anleitung zu lesen. Anspruchsvollere Spiele müssen daher zwingend ein Tutorial enthalten, das den Spieler an die Hand nimmt.
- Spiele müssen Spaß machen.

Das ist alles leicht gesagt, und die Schlussfolgerung lautet: Sie müssen Spiele noch mehr testen als andere Apps. Geben Sie ein Spiel mehreren Spielern in die Hand: Leuten mit schnellen Fingern, aber auch Personen, die noch nie ein Smartphone in der Hand hatten. Beobachten Sie Ihre Tester, und ziehen Sie Rückschlüsse.

5.4.2 Feineinstellungen

Auch wenn Sie das Spiel nicht gleich komplett umschreiben können, an einigen Parametern können Sie leicht drehen.

Ist die Mückenjagd am Anfang zu einfach? Erscheinen so wenige Mücken, dass den Spielern langweilig wird?

Verdoppeln Sie die Anzahl der Mücken in der Methode `starteRunde()`:

```
muecken = runde * 20;
```

Aber Vorsicht, spätestens ab Runde 4 werden Sie von einem Schwarm überfallen, dessen Sie kaum mehr Herr werden.

Ist der Sekundentakt zu langsam?

Nun, im Vergleich mit der Zeit, die Sie beispielsweise mit dem Lesen dieses Kapitels verbringen, ist eine Sekunde ziemlich kurz. Aber sie ist lang in Relation zum Aufbau eines

Fernsehbildes (alle 20 Millisekunden) oder zu der zeitlichen Auflösung, die unser Spiel maximal erreichen kann (1 Millisekunde).

Wir sind nicht auf den Sekundenrhythmus festgelegt. Um den Spielablauf zu verbessern, ändern Sie probeweise die Zeitscheibe (also das zeitliche Intervall unserer Game Engine) von einer ganzen auf eine Zehntelsekunde (100 ms).

Das ist gar nicht so kompliziert, denn Sie müssen dazu nicht viel tun. Die Berechnungen bleiben gleich, lediglich die Anzahl der Zeitscheiben ändert sich von 60 auf 600 und das Intervall für die Handler-Ereignisse von 1.000 auf 100. Um diese Werte leicht an allen Stellen, an denen sie vorkommen, ändern zu können, machen Sie sie zunächst zu Konstanten: Markieren Sie eine `1000` irgendwo in der `GameActivity`, und wählen Sie im Kontextmenü REFACTOR • EXTRACT CONSTANT (Abbildung 5.12).

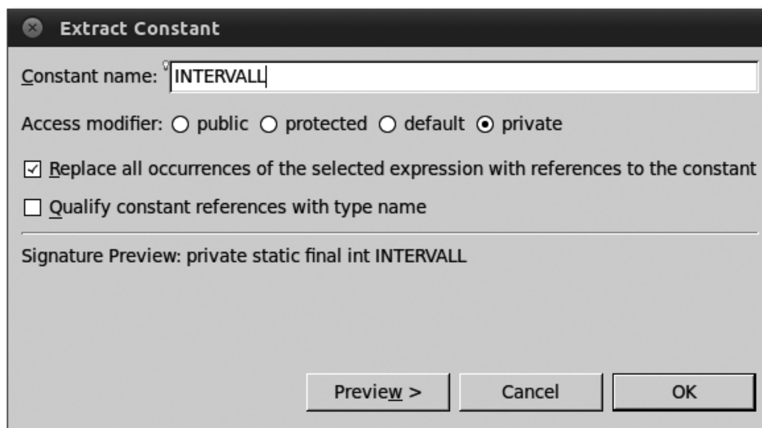

Abbildung 5.12 Lassen Sie Eclipse alle vorkommenden Werte 1.000 durch eine Konstante ersetzen, indem Sie das erste Häkchen setzen.

Auf diese Weise ersetzt Eclipse alle Zahlen 1.000 durch eine einzige Konstante `INTERVALL`, deren Wert Sie dann leicht auf 100 ändern können. Verfahren Sie genauso mit der Zahl 60, indem Sie eine Konstante `ZEITSCHEIBEN` einführen und sie anschließend auf 600 setzen.

Sie sehen jetzt übrigens, warum es sinnvoll war, bei der Berechnung der Länge des Zeitbalkens nicht durch 5 zu teilen, sondern zu schreiben: *300/60. Denn nur so konnte die Zahl 60 durch die Konstante `ZEITSCHEIBEN` ersetzt werden. Sich die einfache Kopfrechenaufgabe 300/60 vom Computer abnehmen zu lassen war im Nachhinein also eine gute Idee!

5.4.3 Hintergrundbilder

Immer denselben Hintergrund anzustarren wird Ihnen langweilig? Gehen Sie an die frische Luft, machen Sie einen Spaziergang, und fotografieren Sie die Gegend. Skalieren Sie die Bilder auf 640 × 854 Pixel, und speichern Sie sie unter Dateinamen mit fortlaufender Nummer im Verzeichnis *drawable*: *hintergrund1.jpg*, *hintergrund2.jpg* etc.

Erweitern Sie die Methode `starteRunde()` um die Anzeige des jeweiligen Hintergrunds. Verpassen Sie dazu zunächst dem äußersten `LinearLayout` im Game-Layout die ID `hintergrund`.

Es gibt hier nur eine kleine Schwierigkeit: Sie können nicht einfach wie bisher die gewünschte Ressource mit `R.drawable.hintergrund` (plus Zahl) referenzieren, weil Sie sich nicht darauf verlassen können, dass der Android Resource Manager fortlaufende Zahlenwerte dafür vergibt. Aber es gibt noch einen weiteren Weg, sich das gewünschte Bild zu holen:

```
int id = getResources().getIdentifier("hintergrund"+Integer.toString(runde),
  "drawable", this.getPackageName());
```

Da wir den richtigen Resource-Identifier nicht kennen, holen wir ihn uns einfach anhand seines Namens, der aus `"hintergrund"` und der angehängten Rundenzahl besteht. Der zweite Parameter der Methode `getIdentifier()` ist der gewünschte Typ der Ressource, der dritte ist der Paketname Ihrer App.

Füttern Sie nun das Bild in den Hintergrund, aber nur, wenn es existiert (vielleicht erreicht ein Spieler überraschend eine Runde, für die Sie kein Foto gemacht haben):

```
if(id>0) {
   LinearLayout l = (LinearLayout) findViewById(R.id.hintergrund);
   l.setBackgroundResource(id);
}
```

Bei der Gelegenheit schalten Sie das ganze Spiel in den Vollbildmodus um: Öffnen Sie das Android-Manifest, und geben Sie als THEME folgende magische Zeile ein:

```
@android:style/Theme.NoTitleBar.Fullscreen
```

Leider können Sie dies nicht mit dem BROWSE-Button auswählen, daher müssen Sie es eintippen.

5.4.4 Elefanten hinzufügen

Sie können ohne weitere Kenntnisse schon jetzt das Spiel um weitere Kniffe erweitern. Wie wäre es z. B. mit einem gelegentlich erscheinenden Elefanten? Wer den antippt, bekommt gemeinerweise 1.000 Punkte abgezogen. Sie können dazu die Methode `eineMueckeAnzeigen()` ändern, indem Sie zufallsgesteuert manchmal anstelle der Mücke ein Bild von einem Elefanten reinmogeln:

```
if(rnd.nextFloat() < 0.05) {
   muecke.setImageResource(R.drawable.elefant);
   muecke.setTag(R.id.tier,ELEFANT);
} else {
   muecke.setImageResource(R.drawable.muecke);
}
```

Bei der gewählten Wahrscheinlichkeit von 0,05 erscheinen in 5 % aller Fälle Elefanten anstelle von Mücken. Um die Tiere voneinander zu unterscheiden, verwenden Sie einfach ein Tag. Dazu müssen Sie eine weitere ID in der Datei *ids.xml* anlegen, außerdem definieren Sie eine Konstante `ELEFANT`:

```
private static final String ELEFANT = "ELEFANT";
```

Der Inhalt des Strings ist egal, aber Sie sollten sich nicht selbst eine Falle stellen, indem Sie etwas anderes hinschreiben. Anstelle eines Strings können Sie auch ein beliebiges anderes Objekt verwenden, z. B. ein `Integer` mit dem Wert 1. Entscheidend ist nicht der Inhalt des Objekts `ELEFANT`, sondern nur seine persönliche Anwesenheit.

In der Methode `onClick()` müssen Sie nun prüfen, ob der Spieler eine Mücke oder einen Elefanten erwischt hat:

```
if(muecke.getTag(R.id.tier) == ELEFANT) {
   punkte -= 1000;
} else {
   gefangeneMuecken++;
   punkte += 100;
}
```

Natürlich zählt der Elefant nicht als Mücke, daher dürfen Sie `gefangeneMuecken` nur im `else`-Fall erhöhen.

Übrigens können Sie sich die Navigation im Code erleichtern, indem Sie die View namens OUTLINE reaktivieren (Sie erinnern sich vielleicht, dass Sie sie kurz nach der

Eclipse-Installation geschlossen haben). Das hilfreiche Fensterchen zeigt Ihnen alle Methoden und Attribute an (Abbildung 5.13).

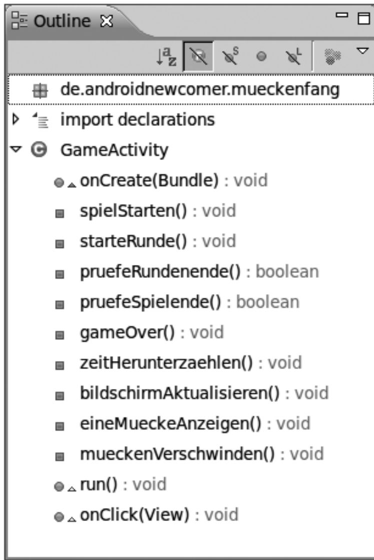

Abbildung 5.13 Die »Outline«-View von Eclipse stellt alle Elemente einer Klasse in einer Baumstruktur dar.

Experimentieren Sie mit den durchgestrichenen Icons: Mit dem ersten können Sie beispielsweise die Attribute ausblenden. Wie es sich gehört, verrät Eclipse Ihnen mit einem kleinen Fähnchen, was ein Icon bewirkt, wenn Sie mit der Maus einen Moment lang darauf verharren. An den farbigen Symbolen vor den Bezeichnern können Sie erkennen, ob diese `public` (grün) oder `private` (rot) deklariert sind. Doppelklicken Sie auf einen Methodennamen oder ein Attribut, um an die betreffende Stelle im Code zu springen.

In weniger als 200 Zeilen Java-Code haben Sie ein einfaches Android-Spiel verwirklicht. Um die Fähigkeiten eines Smartphones richtig auszureizen, fehlt freilich noch einiges. Aber es kommen ja auch noch ein paar Kapitel ...

Index

@Override ... 91
9-Patches ... 343

A

AAC+ ... 183
above ... 355
abstract ... 64, 313
Accelerometer ... 304
Activity ... 91
ActivityNotFoundException ... 331
Adapter ... 247
ADB ... 81, 88, 126
adb ... 88
add() ... 323
addView() ... 161
ADT ... 73, 77, 89, 106
AlertDialog ... 345
AlertDialog.Builder ... 345
Alpha ... 188
Alpha-Transparenz ... 146
andengine ... 37
Android Debug Bridge → ADB
Android Development Tools → ADT
Android Resource Manager ... 88, 106
Android SDK ... 79
Android Virtual Device → AVD
AndroidHttpClient ... 230
Android-ID ... 379
Android-Manifest ... 99, 103, 134, 137, 313
AndroidPI ... 382
AndSMB ... 128
Animation ... 187
AnimationListener ... 195
Annotation ... 66
Apache ... 128
API-Key ... 323
APK ... 106, 358, 359
apkbuilder ... 79
Apotheke ... 34
App Center ... 382
Application Nodes ... 101
AppsLib ... 380
AppStore ... 25
Array, zweidimensionales ... 205
ArrayAdapter ... 251
ArrayList ... 59
ArrayWayOverlay ... 327
Atomreaktoren ... 41
Attribut ... 48
AttributeSet ... 297
Audacity ... 180
Audioformate ... 182
Augmented Reality ... 34, 261, 289
Auswahlliste ... 252
AVD ... 80, 82, 83
Azimuthwinkel ... 290

B

Background ... 139, 338, 339
Background-Thread ... 236
barcoo ... 31
Basisklasse ... 64
Baumstruktur ... 76
Beans ... 39
Bedingung ... 53
Beschleunigungssensor ... 29
Bildschirmausrichtung ... 101
BillingReceiver ... 377
BillingResponseHandler ... 377
BillingService ... 375, 376
bin ... 126
Bit ... 48
boolean ... 48, 49, 156
Boolesche Operatoren ... 156
Breakpoint ... 336
Browser-Spiele ... 41
Build Target ... 90
Bump ... 29
Button ... 111, 112
byte ... 49
Bytecode ... 40

C

C ... 39
C++ ... 39

c:geo ... 30
cacheColorHint ... 250
Calendar ... 350
Camera ... 262
CameraView ... 262
Canvas ... 286
Cast ... 123
Casting ... 159
Caused by ... 332
char ... 50
CheckBox ... 111
checked ... 343
Checkout ... 361
children ... 163
Chrominanz ... 272
class ... 42
clear() ... 323
colors.xml ... 351
Compiler ... 39
Console-View ... 125
constructor ... 44
Content Assist ... 135
Context ... 160
Countdown ... 151
CPU ... 39
Culicidae ... 131
Custom View ... 285, 302
Cut the Rope ... 36

D

Dalvik Debug Monitor Server → DDMS
Dalvik VM ... 40
Date ... 162
Datentypen, primitive ... 49
DatePicker ... 121
DatePickerDialog ... 345, 349
DDMS ... 78
Debug-Perspektive ... 336
Debug-View ... 336
Debug-Zertifikat ... 129, 361
Denglisch ... 43
device independant pixels ... 140
Dialog ... 167, 345
DialogInterface ... 346
Digicam ... 32, 261
Digitale Signatur ... 359
dismiss() ... 346
DisplayMetrics ... 150
doGet() ... 224
double ... 50
draw9patch ... 343
drawable ... 104, 108, 109
drawArc() ... 300
Dropbox ... 128

E

Eclipse ... 41, 47, 73, 85
Editable ... 125
EditText ... 112
else ... 204
Emulator ... 71, 73, 79, 82
enabled ... 343
Entwicklerkonsole ... 362, 371
Entwicklerkonto ... 361
Entwicklungsumgebung ... 73
Erdanziehungskraft ... 303
Ereignis ... 170
Warteschlange ... 170, 171
event ... 170
Exception ... 233
checked ... 235
unchecked ... 235
extends ... 64

F

Farbressourcen ... 146
Farbwert, hexadezimaler ... 146
Fehlerberichte ... 334
FILL ... 286
FILL_AND_STROKE ... 286
final ... 107, 353
findViewById() ... 123, 158, 256, 352
finish() ... 136
flickr ... 33
float ... 50, 295
focused ... 343
Form Widgets ... 111
Fortran ... 39
Fragezeichen ... 87
FrameLayout ... 145
fromHtml() ... 245

G

Galaxy of Fire 2 ... 179
Game Engine ... 142
Garbage Collector ... 44, 88, 185
Geocaching ... 24, 29
Geokoordinaten ... 24
getAnimation() ... 197
getChildAt() ... 163
getChildCount() ... 163
getDefaultSensor() ... 282
getDrawable() ... 245
getIdentifier() ... 176, 207
getRessources() ... 220
getSharedPreferences() ... 214
getSystemService() ... 282, 322
Getter ... 66
getText() ... 219
getWriter() ... 227
GIMP ... 202
Global Positioning System → GPS
GONE ... 218
Google App Engine ... 221
Google Maps ... 25
Google Play ... 16, 25
Google Sky Map ... 27
GPS ... 24, 319
Grafiken ... 104
Gravity ... 118, 140, 161

H

handleMessage() ... 307
Handler ... 169, 171, 307
  statischer ... 317
Hänger ... 88
Hexadezimalzahl ... 108, 111
Hierarchy Viewer ... 78
High Replication Datastore ... 225
Hintergrundfarbe ... 250
HorizontalScrollView ... 121
HTML ... 242
HTML-Farbcodes ... 146
HTTP ... 223
HTTP-Client ... 229
HttpEntity ... 231
HttpGet ... 230
HttpServletResponse ... 225
Hubble-Teleskop ... 28
HVGA ... 83
Hypertext Transfer Protocol → HTTP

I

Icon ... 101, 104, 362
icon.png ... 104
ids.xml ... 162
if ... 53
IllegalArgumentException ... 235
ImageButton ... 121
ImageGetter ... 245
ImageView ... 121
IMarketBillingService ... 375
implements ... 95, 122
Import ... 94
Importieren ... 46
In-App-Payment ... 371
Initialisieren ... 48
Inkscape ... 138, 338, 362
InputStreamReader ... 231
Installation ... 368
Instanz ... 42
Instanziieren ... 44
int ... 50
Integer ... 49
Intent ... 136
Intent Filter ... 101
Interface ... 95, 122
interface ... 95
Interpolation ... 193
Interpolator ... 194
invalidate() ... 285
INVISIBLE ... 217
IOException ... 233, 264, 358
ISO-8859-1 ... 232

J

jar ... 324
Java ... 39
Java Development Kit (JDK) ... 71
Java Runtime Environment (JRE) ... 40
Java-Kompatibilität ... 87
Java-Perspective ... 75
Java-Runtime ... 42, 43

K

Keystore ... 360
Klasse ... 42
  anonyme innere ... 196
  innere ... 191
  lokale ... 308
Kommentar ... 92
Konstante ... 161
Konstruktor ... 44
Koordinaten, sphärische ... 290
Koordinatensystem, kartesisches ... 290
Kopierschutzmechanismus ... 365
Kugelkoordinatensystem ... 290

L

Labyrinth ... 23
Lagesensoren ... 38
landscape ... 102
Laufvariable ... 163
Launch Options ... 83
Launcher ... 101
Layout ... 111, 133, 135, 355
Layout Gravity ... 145
Layout Weight ... 147
layout_weight ... 356
Layout-Datei ... 167
Layout-Editor ... 110, 340
LayoutInflater ... 256, 259, 354
LayoutParams ... 150, 161, 201
LinearLayout ... 117
lineTo() ... 288
ListView ... 120, 247
Lizenzierungsservice ... 365
loadAnimation() ... 190
Location ... 322
LocationManager ... 322
Log Level ... 330
Log.e ... 334
LogCat ... 266, 330
Log-Filter ... 335
Logging ... 265
LOGTAG ... 334
long ... 50
loop ... 163
Luminanz ... 272

M

Magnetfeldsensor ... 23, 282
main.xml ... 111
MapActivity ... 325
MapController ... 327
MapDroyd ... 26
mapsforge ... 324
MapView ... 326
MapViewMode ... 326
Maßstab ... 150
match_parent ... 119
Math ... 150, 159
Math.min() ... 150
MediaPlayer ... 184
Methode ... 51
  statische ... 55
Mikrofon ... 24, 35, 180
Mindestpreise ... 365
modifier ... 46
Modifizierer ... 46
moveTo() ... 288
mp3 ... 183
Mücke ... 20, 132
Multitasking ... 169

N

Namespace-Attribut ... 190
Network based Location ... 319
Netzwerkbasierte Ortsbestimmung ... 319
notifyDataSetChanged() ... 258
NullPointerException ... 235, 318, 332
NV21 ... 272
NV21Image.java ... 278

O

Objekte ... 42
Objektorientierung ... 42
Ogg Vorbis ... 183
onActivityResult() ... 213
onClick() ... 346
OnClickListener ... 122, 160, 306, 314, 320, 353
onClickListener() ... 136
onCreate() ... 91
onDateSet() ... 350
onDateSetListener ... 349

onDestroy() ... 186
onDraw() ... 286
onInit() ... 95
onKeyDown() ... 258
onLocationChanged() ... 322
onPreviewFrame() ... 271
onResume() ... 212
onSensorChanged() ... 284, 294, 307
Openstreetmaps.org ... 324
Operator ... 51
Organize Imports ... 94, 98
OSM ... 26
Outline ... 76, 117, 177
OutOfMemoryError ... 371
Overlay ... 326
OverlayWay ... 326
override ... 66

P

Package ... 45
Package Builder ... 106
Package Explorer ... 76, 91, 99, 104, 110
Padding ... 140
Paint ... 286
Panoramico ... 33
Parameter ... 45, 52
parseInt() ... 225
Pascal ... 39
Path ... 287
PayPal ... 381, 383, 385
Permission ... 102, 229
Perspective ... 75
Pfad ... 287
Physik-Engine ... 36
Pivotpunkt ... 192
Plugins ... 73
PNG ... 105, 338
Polarwinkel ... 290
Polymorphie ... 67
portrait ... 102
postDelayed() ... 172, 348
PreviewCallback ... 270
private ... 54, 65, 93
Privater Schlüssel ... 360
Problems ... 76
Programmbibliothek ... 324
ProgressBar ... 111
ProgressDialog ... 345, 347
protected ... 65
Prozess-ID ... 330
public ... 46
purchaseResponse() ... 377
Pythagoras ... 208

Q

qualified name ... 46
Qualifizierter Name ... 46
Query ... 227

R

R.java ... 107, 123
Radar ... 297
Random ... 153
raw ... 181
Receiver ... 377
Refactor ... 157, 175, 254
Refactoring ... 157
Reference Chooser ... 139
registerListener() ... 283
RelativeLayout ... 355
removeCallbacks() ... 193
removeUpdates() ... 323
removeView() ... 166
replace() ... 233
request ... 223
requestLocationUpdates() ... 322
requestPurchase() ... 376
res ... 103, 108
Resource Chooser ... 113, 140
Resource Manager ... 107, 111
response ... 223
Ressourcen ... 103
Restore ... 76
rotate() ... 288
round() ... 150, 203
Router ... 24
run() ... 171, 236
Runescape ... 41
Runnable ... 171, 236
runOnUiThread() ... 239
RuntimeException ... 235

S

Sampling ... 182
scale-independant pixels ... 140
Schatztruhe ... 19
Schleife ... 61, 163
Screen Orientation ... 304
Screenshots ... 362
ScrollView ... 121
SDK Platform Tools ... 80
SD-Karte ... 82
selected ... 343
selector ... 342
sendEmptyMessage() ... 318
SensorEvent ... 284
SensorEventListener ... 283, 306
SensorManager ... 282, 304
Serveranwendung ... 29, 32, 41
Service ... 312, 319
Servlet ... 223
setAnimationListener() ... 197
setAntiAlias() ... 286
setBackgroundResource() ... 176
setColor() ... 286
setContentView() .. 111, 136, 250, 288, 326, 352
setDisplayOrientation() ... 264
setImageResource() ... 177, 205
setLayoutParams() ... 201
setOneShotPreviewCallback() ... 270
setPreviewDisplay() ... 264
setProgress() ... 349
setResult() ... 213
setStrokeWidth() ... 298
setStyle() ... 286
setTag() ... 291
Setter ... 66
setVisibility() ... 217
setWayData() ... 327
Shaky Tower ... 37
Shared Preferences ... 214, 378
SharesFinder ... 128
short ... 50
SimpleStringSplitter ... 243, 257
Single Processing ... 169
SlideME ... 384
SMB ... 128
Software Development Kit ... 79
Software Sites ... 77
Sound ... 179
Soundformate ... 182
Speicher ... 43
Spiel ... 36, 37
Spielregeln ... 141
Spinner ... 252
Sprachausgabe ... 94
Sprachsuche ... 34
src ... 91
Stacktrace ... 331, 371
Standardkonstruktor ... 57
Star Trek ... 21, 34
StartActivity ... 91, 111
startActivity() ... 136
startActivityForResult() ... 213, 253
startAnimation() ... 190
startPreview() ... 266
startService() ... 316
state_pressed ... 342
static ... 316, 318
Statische Attribute ... 316
Statische Methoden ... 150
Statistik ... 368
Steuerrad ... 23
stopService() ... 316
Stream ... 231
String ... 50
String-Konstante ... 115
String-Ressource ... 113
strings.xml ... 116, 220
String-Verweise ... 116
STROKE ... 286
Stromverschwendung ... 53
Styles ... 339
super ... 93
surfaceChanged() ... 265
surfaceCreated() ... 263
SurfaceHolder ... 262
SurfaceView ... 121, 262
svg ... 138
Synonymwörterbuch ... 20

T

Tag ... 162, 330
Task List ... 76
Telepathie ... 111
Text Color ... 146, 340

Text Style ... 146
TextToSpeech ... 95
TextView ... 111
Theme ... 340
Thesaurus ... 20
this ... 94
Thread ... 236
TimePicker ... 121
TimePickerDialog ... 345, 349
Toast ... 353
Traceview ... 78
translate() ... 288
Transparenz ... 105
Tricorder ... 21
trim() ... 219
try-catch ... 234
Tutorials ... 76
Type-Casting ... 159

U

Überschreiben ... 66
Ubuntu ... 75
UI-Thread ... 236, 239
Unicode ... 232
Update ... 366
URLEncoder ... 241
URL-Parameter ... 224
USB-Debugging ... 85
USB-Kabel ... 69, 85, 89, 97
User Agent ... 230
Uses Permission ... 102
UTF-8 ... 232

V

Vampir ... 131
Variable ... 57
Variablenname ... 43, 44, 48
Vektor ... 198
Venus ... 27
Verantwortung ... 54
Vererbung ... 63, 64
Vergleichsoperator ... 54
Versionscode ... 100
Versionsname ... 100
Versionsnummer ... 366
VideoView ... 121
View ... 123, 158
bewegen ... 198
einblenden ... 188
ViewGroup ... 163
Virtuelle Kamera ... 291
Virtueller Raum ... 289
void ... 52
Vollbildmodus ... 176

W

Wahrheitswert ... 53
Wahrscheinlichkeit ... 152
WAV ... 183
Webcams ... 33
WebView ... 121
while() ... 276
while(bedingung) ... 163
Wikipedia ... 34
Wikitude ... 33
Wizard ... 132
Workspace ... 73
Wrap in Container ... 266
wrap_content ... 119, 140

X

XML ... 267, 342
xmlns ... 190

Y

YCbCr_420_SP ... 272
YouTube ... 363

Z

Zeitmaschine ... 18
Zentrifugalkraft ... 304
Zertifikat ... 359, 360
Zufallsgenerator ... 153, 199
Zugriffsbeschränkung ... 54
Zugspitzbahn ... 19
Zuweisungsoperator ... 49, 54