Charlie **Collins**, Michael **Galpin**, Matthias **Kaeppler**

Android w praktyce

Najlepsze techniki programowania na Androida **w zasięgu ręki**!

Helion

Tytuł oryginału: Android in Practice

Tłumaczenie: Tomasz Walczak

Projekt okładki: Studio Gravite / Olsztyn
Obarek, Pokoński, Pazdrijowski, Zaprucki

ISBN: 978-83-246-4810-8

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: helion@helion.pl
WWW: http://helion.pl (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie?gimpbi
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

15 Pisanie aplikacji na tablety z Androidem

W tym rozdziale

- Stosowanie fragmentów
- Pasek akcji
- Implementowanie przeciągania

Wszystko staje się coraz większe. Dlatego obecnie należy pisać programy w bardziej wyrafinowany sposób.

Bill Budge

Był rok 2001. Microsoft, największa firma technologiczna na świecie, zaprezentował przełomową wersję niezwykle popularnego systemu operacyjnego — Windows XP Tablet PC Edition. Zdaniem Microsoftu miał to być początek ery urządzeń dotykowych. Wiemy, jak to się skończyło. System XP Tablet PC Edition okazał się niewypałem.

Tak naprawdę komputery z systemem XP Tablet nie były pierwszymi urządzeniami z wyświetlaczem dotykowym przeznaczonymi na rynek masowy. Dziesięć lat wcześniej inżynierowie z Apple'a opracowali prototyp, który w przekształconej postaci wprowadzono na rynek jako komputer Newton. Opracowane 10 lat później przez Microsoft komputery Tablet PC były podejrzanie podobne do Newtona. Jednak prototypowa wersja Newtona nigdy nie trafiła do sprzedaży; komputery z tej rodziny stały się za to poprzednikami palmtopów.

Stwierdzenie, że urządzenia dotykowe przez wiele lat były ogłaszane jako następny wielki hit, to poważne niedomówienie. Można uznać, że wcześniejsze

próby zakończyły się niepowodzeniem, ponieważ producenci próbowali przekształcić komputery PC w tablety. Dwadzieścia lat po opracowaniu prototypowego tabletu Newton stało się oczywiste, że naturalnym urządzeniem dotykowym są smartfony i to na nich należy wzorować tablety. W ten sposób powstały iPad Apple'a i Android dla tabletów. W wersji Android 3.0 (Honeycomb) wprowadzono istotne zmiany, opracowane z myślą o tabletach. Co to oznacza dla programistów aplikacji na Android?

Jeśli już udostępniasz w sklepie Android Market kilka aplikacji, być może nasuwa Ci się pytanie: „Co powinienem zrobić, aby programy działały na tabletach?". Liczba użytkowników tabletów jest znacznie mniejsza niż liczba osób korzystających z telefonów. Zyskanie popularności przez nowe urządzenia, jakimi są tablety, wymaga czasu, a użytkowników smartfonów są miliony. Prawdopodobnie dużo większe zyski przyniesie inwestowanie w rozwijanie aplikacji na Android przeznaczonych na smartfony, a nie na tablety. Jest to prawdą zwłaszcza w przypadku nowych aplikacji. Tworząc programy na smartfony, dotrzesz do znacznie większej grupy odbiorców.

Istnieją jednak wyjątki od tej reguły. Niektóre rodzaje aplikacji lepiej nadają się dla tabletów niż dla smartfonów. Na przykład czytniki wiadomości, aplikacje z sieci społecznościowych, sklepy internetowe czy inne programy wyświetlające bogate materiały wyglądają znacznie lepiej na większym ekranie. Na smartfonie powiązane informacje trzeba wyświetlać na dwóch lub trzech ekranach, a na tablecie można je zaprezentować na jednej angażującej czytelnika stronie. Nawet w innych obszarach wczesne wprowadzanie aplikacji na tablety do sklepu Android Market przynosi poważne korzyści. Program może zyskać popularność z uwagi na niewielką konkurencję, co zapewnia dobrą pozycję wyjściową, kiedy więcej osób zacznie kupować tablety i szukać ciekawych aplikacji w sklepie.

Niezależnie od tego, czy lubisz nowinki, czy nie, w pewnym momencie zechcesz utworzyć aplikację na tablety (w przeciwnym razie prawdopodobnie nie czytałbyś tego rozdziału). W tym rozdziale wzbogacamy aplikację DealDroid z rozdziału 2. i tworzymy jej wersję na tablety. Gotowy produkt przedstawiono na rysunku 15.1.

Decyzji o utworzeniu aplikacji na tablety nie podejmuje się stopniowo. Trzeba od początku stwierdzić, że program ma działać w tabletach. Rozwijanie takich aplikacji jest ekscytujące, ponieważ tablety w porównaniu ze smartfonami otwierają przed programistami wiele nowych możliwości. Liczne typowe problemy, na przykład mały ekran, niewielka ilość pamięci, konieczność obsługi dawnych wersji Androida i wolna sieć, w tabletach są mniej dotkliwe. Jednak zanim zaczniemy tworzyć fragmenty pełne widoków `StackView`, warto zastanowić się, jakie elementy aplikacji są już gotowe i jak je wykorzystać.

Rysunek 15.1. Aplikacja DealDroid w wersji na tablety

15.1. Przygotowania do tworzenia aplikacji na tablety

Pisanie programów na tablety nie polega tylko na stosowaniu nowych interfejsów API i większych grafik. Musisz zdecydować, czy chcesz utworzyć odrębną aplikację, czy rozwinąć istniejący program na smartfony, tak aby działał poprawnie także na tabletach. W tym rozdziale koncentrujemy się na pisaniu nowych aplikacji. Pozwala to wykorzystać wszystkie możliwości Androida 3.0. Wszystkie (lub prawie wszystkie) techniki z tego rozdziału można zastosować w czasie tworzenia standardowych aplikacji.

TECHNIKA 87. Wykorzystywanie istniejącego kodu za pomocą projektów bibliotek

Choć tworzymy odrębną aplikację przeznaczoną na tablety z Androidem, nie oznacza to, że program jest zupełnie odmienny od rozwiązań tworzonych na smartfony. Obie wersje aplikacji mają wiele wspólnych funkcji i mogą korzystać nawet z tych samych danych. Dane te można przechowywać lokalnie w urządzeniu lub w chmurze na serwerze. Sposób dostępu do takich danych, a nawet porządkowania ich po pobraniu do pamięci lokalnej jest taki sam jak w smartfonach. Na szczęście istnieje dobry sposób na współużytkowanie kodu przez różne aplikacje na Android.

PROBLEM

W nowej aplikacji na tablety zamierzamy wykorzystać kod istniejącego programu na smartfony. Chcemy, aby istniała tylko jedna kopia kodu. Pozwala to dodawać nowe funkcje, naprawiać błędy i wykonywać podobne zadania w jednym miejscu.

ROZWIĄZANIE

Sposobem na współużytkowanie kodu między aplikacjami na Android jest zastosowanie projektów bibliotek dla Androida. Takie projekty dobrze nadają się do porządkowania kodu i wprowadzono je w tym samym czasie co Android 2.2. Projekty bibliotek nie są przeznaczone tylko dla tabletów i Androida 3.0 —

pozwalają współużytkować kod także między różnymi aplikacjami na smartfony. Taki projekt jest bardzo przydatny przy tworzeniu aplikacji, która ma działać na smartfonach i tabletach.

Wspomnieliśmy już, że projekty bibliotek wprowadzono w tym samym okresie co Android 2.2. Dostosowano je jednak do jeszcze wcześniejszych wersji Androida. Możliwe, że przechowujesz kod w takich bibliotekach i możesz natychmiast wykorzystać je do tworzenia aplikacji na tablety na podstawie istniejącego kodu programów na smartfony. Jeśli tak jest, możesz przejść bezpośrednio do następnej techniki. Jeżeli jednak jeszcze nie stosujesz projektów bibliotek, czeka Cię refaktoryzacja i zmiana uporządkowania kodu.

W tej technice przekształcamy aplikację DealDroid przedstawioną po raz pierwszy w rozdziale 2. na program na tablety. Cały kod aplikacji DealDroid znajduje się w jednym projekcie aplikacji na Android. Musimy to zmienić. Na rysunku 15.2 pokazano nowy sposób uporządkowania kodu, pozwalający wykorzystać ten kod w nowym projekcie aplikacji na tablety.

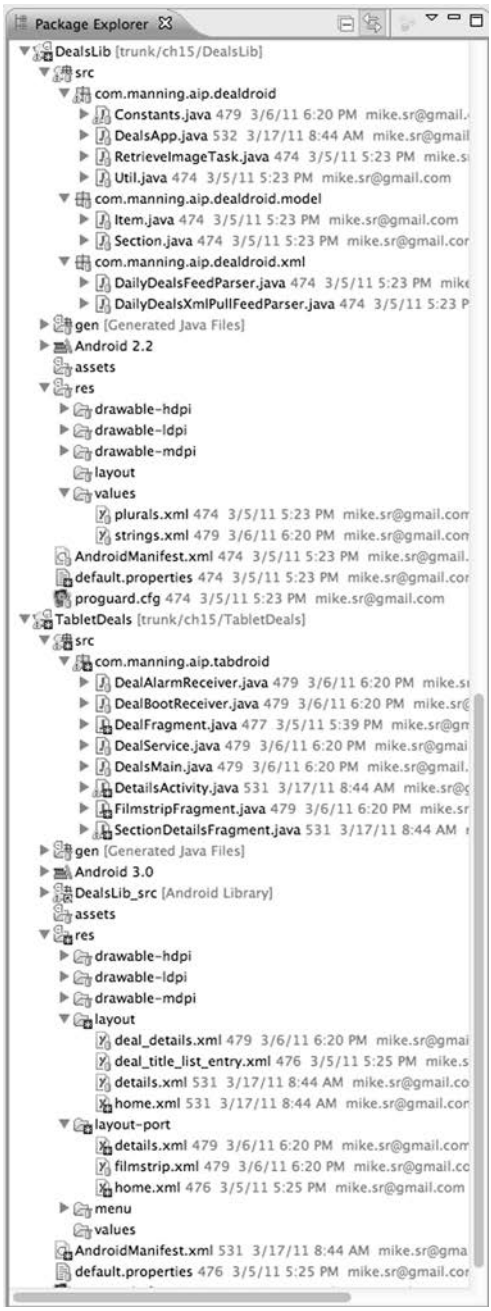
Na rysunku 15.2 widoczne są projekty DealsLib i TabletDeals. DealsLib to projekt biblioteki dla Androida, obejmujący kod współużytkowany przez aplikacje na smartfony i tablety. TabletDeals to projekt aplikacji na tablety. W tym rozdziale opisujemy go bardzo szczegółowo. Na rysunku widać, jaki kod znajduje się w projekcie biblioteki. Umieszczono w niej (w pakiecie `com.manning.aip.dealdroid.xml`) cały kod do pobierania danych z internetu i ich przetwarzania. Kod ten może być taki sam dla tabletów i smartfonów. Dane w formacie XML są przetwarzane na obiekty modelu (określone w pakiecie `com.manning.aip.dealdroid.model`) używane przez aplikację. Także kod tych obiektów jest częścią biblioteki.

W pakiecie najwyższego poziomu (`com.manning.aip.dealdroid`) znajduje się kilka innych klas. Najciekawszą z nich jest `DealsApp`. Jest to klasa typu `Application` używana w obu aplikacjach. Obejmuje pamięć podręczną z danymi, a także stan aplikacji. Stan w aplikacjach na tablety mógłby być inny, jednak tu jest taki sam, dlatego wspomnianą klasę można współużytkować w wersjach programu na smartfony i tablety.

Ponadto między aplikacjami współużytkowane są niektóre zasoby, przede wszystkim pliki *strings.xml* i *plurals.xml*. Współużytkować można także inne zasoby, na przykład obiekty graficzne.

OMÓWIENIE

Projekty bibliotek dają duże możliwości. Programiści — zwłaszcza piszący pakiety aplikacji współużytkujących duże fragmenty kodu — dobitnie domagali się tego mechanizmu. Często wspólny kod służy do obsługi dostępu do sieci i modelu danych, podobnie jak w przykładzie. Nieraz wspólny jest też kod do uwierzytelniania użytkowników i późniejszego zarządzania informacjami o tożsamości (na przykład znacznikami uwierzytelniającymi i (lub) określającymi uprawnienia).



Rysunek 15.2. Kod przygotowany do współużytkowania z aplikacją na tablety

Tego rodzaju kod często obejmuje elementy interfejsu użytkownika, ponieważ programiści zwykle starają się ujednolicić sposób logowania się do aplikacji. Uzyskanie tego efektu nie jest trudne, gdyż w bibliotece można umieścić aktywności, XML-owy kod układu itd. Dla projektu biblioteki trzeba też utworzyć plik

AndroidManifest.xml. Można w nim deklarować aktywności, usługi i inne elementy — tak jak w innych manifestach. Jeśli jednak chcesz wykorzystać w aplikacji aktywność z projektu biblioteki (lub inny komponent deklarowany w manifeście), musisz zadeklarować tę aktywność w pliku *AndroidManifest.xml* aplikacji. Manifest projektu biblioteki pełni funkcję menu dostępnych komponentów, które trzeba zadeklarować w pliku manifestu aplikacji.

Kod można współużytkować między projektami na różne sposoby. Jeśli korzystasz ze środowiska Eclipse (albo Anta, Mavena lub innego narzędzia zarządzającego zależnościami), możesz utworzyć własną bibliotekę z kolekcją kodu w Javie i wykorzystać ją w aplikacji. Zależność między biblioteką a aplikacją może występować na poziomie kodu źródłowego lub na poziomie binarnym. W tym drugim przypadku najpierw należy skompilować projekt biblioteki bądź nawet spakować go do archiwum JAR. Utrudnieniem jest wtedy tylko konieczność upewnienia się, że w kodzie biblioteki nie występują żadne standardowe klasy Javy niedozwolone w Androidzie, a także użycie odpowiedniego archiwum *android.jar*. Przy stosowaniu projektów bibliotek dla Androida zadania te są wykonywane automatycznie.

Inną ważną zaletą projektów bibliotek dla Androida w porównaniu ze standardowymi bibliotekami Javy jest obsługa zarządzania zasobami. W przykładzie w bibliotece umieszczamy standardowy plik *strings.xml*. Pozwala to współużytkować ten plik w wersjach na smartfony i tablety. Można też zastąpić konkretny łańcuch znaków lub dodać nowe fragmenty tekstu przez umieszczenie odrębnego pliku *strings.xml* w projekcie aplikacji. Kompilator scala wtedy zasoby. Dotyczy to także innych zasobów, na przykład stylów i obiektów graficznych, a nawet plików układu.

Teraz, kiedy znasz już dobry sposób na porządkowanie kodu i współużytkowanie go w aplikacjach na smartfony i tablety, można dokładniej zastanowić się nad wersją na tablety. Wspomnieliśmy już, że tworzymy aplikację przeznaczoną tylko na tablety. Nie zamierzamy rozwijać wersji działającej równie dobrze na smartfonach i tabletach. Na szczęście stosowanie naszego podejścia jest proste i przynosi duże korzyści.

TECHNIKA 88. Tworzenie aplikacji przeznaczonej na tablety

Programiści aplikacji na Android niechętnie mówili o zróżnicowaniu urządzeń (ang. *fragmentation*). Określenie to często stosowali przeciwnicy Androida, twierdzący, że zbyt trudno jest tworzyć aplikacje na tę platformę, ponieważ trzeba zapewnić obsługę urządzeń z ekranami o różnej wielkości i z innymi niejednolitymi cechami. Jednocześnie jest to jednak ukryta wartość Androida. Właściwy sposób programowania wymagał, aby nie robić założeń co do wielkości i proporcji ekranu. Programiści mieli wiele narzędzi do projektowania aplikacji z układem dostosowującym się do wyświetlacza. Kiedy więc pojawiały się nowe urządzenia z ekranami o przekątnej wynoszącej 4 lub 4,3 cala albo z mniejszymi,

2,5-calowymi wyświetlaczami, większość programów działała w nich prawidłowo. Nawet na pierwszych tabletach z 7-calowymi ekranami i Androidem 2.2 większość aplikacji funkcjonowała bez problemów (choć były też wyjątki od tej reguły — niektórzy programiści nie stosowali najlepszych praktyk i w czasie projektowania układów robili założenia dotyczące wielkości wyświetlacza). Było to wielką zaletą małych tabletów. Kiedy pojawiły się na rynku, od razu istniało wiele aplikacji, które prawidłowo na nich działały.

Jednak w czasie prac nad większymi tabletami okazało się, że opracowanie systemu operacyjnego pod kątem takich urządzeń ma duże zalety. Dlatego powstał Android 3.0. Platforma ta obejmuje elementy sprawiające, że aplikacje mogą działać równie dobrze zarówno na smartfonach, jak i na tabletach. Ponadto pozwala tworzyć atrakcyjne programy przeznaczone tylko na tablety. Wymaga to jednak zablokowania dostępu do aplikacji użytkownikom mniejszych urządzeń.

PROBLEM

Piszemy aplikację przeznaczoną tylko na tablety. Chcemy wykorzystać duży ekran i wszystkie możliwości platformy dostępne w tabletach. Nie zamierzamy dostosowywać aplikacji do urządzeń z mniejszymi ekranami, niezależnie od wersji Androida działającej na tych urządzeniach.

ROZWIĄZANIE

Możliwe, że już znasz rozwiązanie. W pliku *AndroidManifest.xml* należy określić wszystkie wymagania aplikacji. Następnie filtry w sklepie Android Market sprawią, że aplikacja nie będzie pojawiać się w urządzeniach innych niż tablety. Na listingu 15.1 znajduje się fragment manifestu pozwalający uzyskać ten efekt.

Listing 15.1. W manifeście można określić, że aplikacja jest przeznaczona tylko na tablety

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="com.manning.aip.tabdroid"
      android:versionCode="1"
      android:versionName="1.0">
    <uses-sdk android:minSdkVersion="11" />          ❶
    <supports-screens android:smallScreens="false"
        android:normalScreens="false"
        android:largeScreens="false"
        android:xlargeScreens="true" />              ❷
</manifest>
```

Aby aplikacja była przeznaczona tylko na tablety, w manifeście trzeba podać dwa podstawowe wymagania. Otóż w urządzeniu musi działać Android 3.0 (Honeycomb) lub nowsza wersja tej platformy ❶. Może się wydawać, że to wystarczy. W końcu jeśli dostępny jest interfejs API w wersji 11 (Android 3.0) lub nowszej, można korzystać ze wszystkich interfejsów API potrzebnych w rozwijanej aplikacji. Gdy powstawała ta książka, wersja Android 3.0 była najnowsza i działała tylko na tabletach, ale do czasu trafienia tej pozycji na półki pojawią się prawdopodobnie

nowsze wersje, które będą obejmować wszystkie funkcje Androida 3.0 i pracować zarówno na smartfonach, jak i na tabletach. Dlatego trzeba też określić, że aplikacja działa tylko na urządzeniach z ekranami `xlarge` ❷. Ten rozmiar wyświetlaczy wprowadzono w Androidzie 2.3. Odpowiada on ekranom mającym przynajmniej siedem cali. Po określeniu w manifeście tych dwóch wymagań można mieć pewność, że każde urządzenie, na którym uruchamiana jest aplikacja, to tablet ze zoptymalizowanymi pod jego kątem interfejsami API wprowadzonymi w wersji Honeycomb.

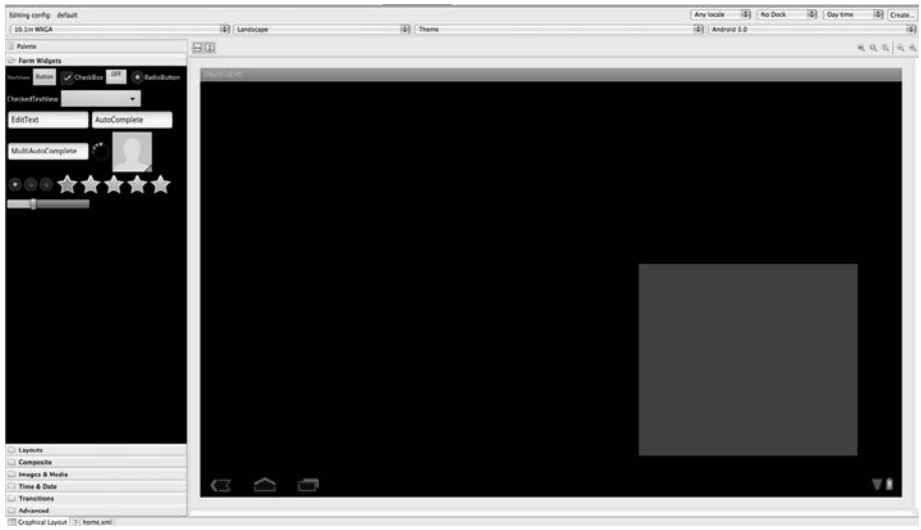
Należy wspomnieć o jeszcze jednym aspekcie programowania aplikacji na tablety. W czasie pisania aplikacji na smartfony programiści często zakładają, że urządzenie zwykle znajduje się w orientacji pionowej. Na szczęście system operacyjny dobrze obsługuje zmiany orientacji, dlatego nawet jeśli programista całkowicie zapomni o przygotowaniu wersji dla poziomego układu ekranu, aplikacja prawdopodobnie będzie działać poprawnie po obróceniu urządzenia. Warto jednak zastanowić się nad trybem poziomym, a czasem dobrze jest nawet przygotować dla niego odrębne układy. W Androidzie standardowo należy utworzyć katalog ze zoptymalizowanymi plikami XML z kodem układów. Inna możliwość to pominięcie orientacji poziomej i obsługiwanie tylko trybu pionowego. Ma to pewne zalety, choć użytkownicy urządzeń z wysuwanymi klawiaturami nie będą zadowoleni z aplikacji napisanej w ten sposób.

Tablety różnią się od smartfonów. Orientacja to jeden z obszarów, gdzie różnice między tymi typami urządzeń są duże. Z tabletów zwykle korzysta się w orientacji poziomej. Dlatego standardowo pliki układu dla tej orientacji umieszcza się w katalogu */res/layout*, a pliki dla trybu pionowego — w katalogu */res/layout-port*. Jeśli korzystasz z wtyczki ADT dla środowiska Eclipse, mechanizm tworzenia interfejsu użytkownika z tej wtyczki pomoże Ci w rozwijaniu aplikacji na tablety, co pokazano na rysunku 15.3.

OMÓWIENIE

Opisane tu podejście pod wieloma względami różni się od tworzenia typowych aplikacji na Android. Zwykle warto obsługiwać jak najwięcej różnych wyświetlaczy. Tu wykluczamy wszystkie wymiary oprócz jednego. Kiedy pojawiły się pierwsze tablety z wersją Honeycomb Androida, nie tylko miały podobne wymiary, ale też tę samą rozdzielczość ekranu. Było to coś nowego dla programistów aplikacji na Android, przyzwyczajonych do tworzenia rozwiązań z uwzględnieniem ekranów o różnej wielkości i rozdzielczości. Od czasów urządzeń G1 nie można było tworzyć programów dostosowanych do ekranu o konkretnych cechach (przy czym fizyczne wymiary poszczególnych modeli tabletów były zróżnicowane). Unikaj jednak stosowania przestarzałych układów `AbsoluteLayout` lub podawania wymiarów w układzie za pomocą fizycznych pikseli.

Przedstawiliśmy projekty bibliotek, najnowsze interfejsy API i układy dla dużych ekranów. Pora rozpocząć tworzenie programów na tablety z Androidem.



Rysunek 15.3. **Tworzenie interfejsu na tablety z wykorzystaniem wtyczki ADT**

Zaczynamy od podstawowych technik, które powinien znać każdy programista aplikacji na tablety. Ponadto pokazujemy, że techniki te nie są ograniczone do tabletów i że można je łatwo wykorzystać także przy tworzeniu aplikacji na smartfony.

15.2. Podstawowe informacje o tabletach

Tablety z Androidem istniały już na długo przed pojawieniem się Androida 3.0. Miały ekrany o przekątnej od pięciu do siedmiu cali, były więc mniejsze niż pierwsze urządzenia z wersją Honeycomb. Takie miniaturowe tablety były ciekawe same w sobie. Jak już wspomnieliśmy, większość aplikacji na Android działała w nich prawidłowo. Z uwagi na dodatkową przestrzeń niektóre takie programy wyglądały całkiem dobrze. Mimo to można było je opisać najwyżej jako poprawne. Jest to dowód na to, że Android potrafi dostosować aplikację do wyświetlacza bez nadmiernego utrudniania pracy użytkownikom.

Android 3.0 zaprojektowano tak, aby był więcej niż poprawny. Podejście nie polegało na dostosowaniu Androida do poprawnej pracy na większym ekranie lub na dodaniu nowych komponentów interfejsu użytkownika. Twórcy platformy wprowadzili poważne zmiany, aby pomóc programistom w skutecznym pisaniu aplikacji na urządzenia z większymi wyświetlaczami. Omawianie podstawowych technik tworzenia aplikacji na tablety zaczynamy od przyjrzenia się jednemu z najważniejszych mechanizmów wprowadzonych w wersji Honeycomb. Są nim fragmenty.

TECHNIKA 89. Fragmenty

Wspomnieliśmy już, że Android 3.0 zaprojektowano z myślą o tabletach. Opracowanie nowej wersji nie polegało na dodaniu nowych elementów do wcześniejszych odmian Androida. Jednym z najlepszych dowodów na to jest interfejs API fragmentów. Fragmenty umożliwiają porządkowanie kodu aplikacji w nowy sposób, znacznie ułatwiający radzenie sobie z tworzeniem układów dostosowanych do dużych ekranów tabletów z Androidem. Jednak mechanizm ten jest przydatny nie tylko w programach na tablety.

PROBLEM

Chcemy podzielić kod aplikacji na moduły, aby można było stosować zupełnie odmienne układy dla orientacji poziomej i pionowej bez konieczności powielania kodu oraz funkcji.

ROZWIĄZANIE

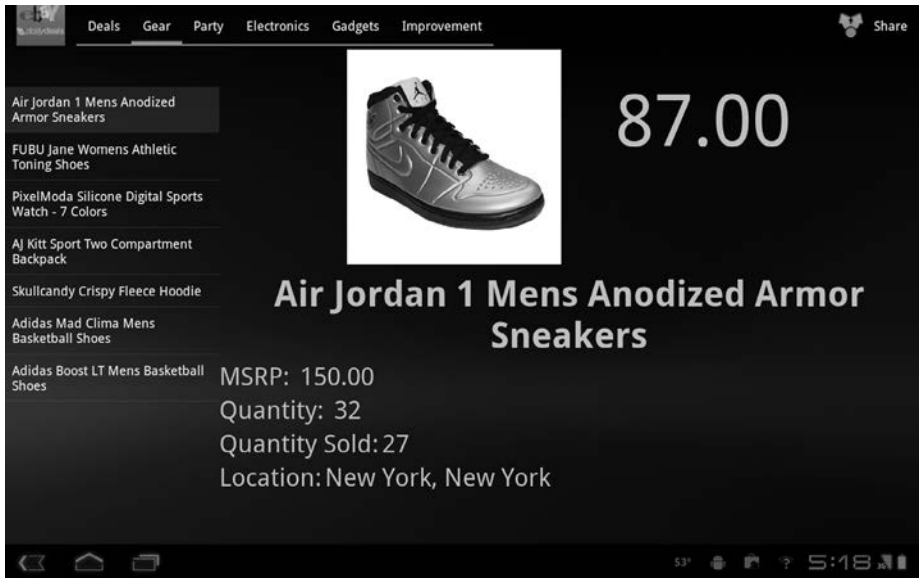
Rozwiązanie polega na użyciu fragmentów do uporządkowania kodu. Stosowanie różnych układów dla orientacji poziomej i pionowej nie jest niczym nowym. W przypadku tabletów istotna jest natomiast ilość miejsca na ekranie. W smartfonach, gdzie wyświetlacze są mniejsze, w układach poziomych i pionowych zwykle dostępne są te same informacje oraz funkcje. Zmiana orientacji prowadzi do sensownego nowego uporządkowania elementów. W tabletach nie jest niczym niezwykłym wyświetlanie na ekranie odmiennych komponentów w różnych układach. Przyjrzyjmy się konkretnemu przykładowi.

Aplikacja DealDroid (rozdział 2.) umożliwia użytkownikom wyświetlanie ofert dnia z eBaya. Jedna z aktywności aplikacji wyświetla listę ofert, a druga — szczegółowe informacje o wybranych ofertach. Na tablecie oba zadania można wykonywać w jednej aktywności, ale tylko w układzie poziomym. Na rysunku 15.4 pokazano wygląd takiej aktywności.

W aplikacjach na tablety często stosuje się pewien wzorzec. Po lewej stronie ekranu wyświetla się przewijaną listę, a po prawej — szczegółowe informacje o wybranym elemencie. Na rysunku 15.5 pokazano, że wybranie elementu z listy prowadzi do zmiany danych wyświetlanych w dużym obszarze ze szczegółami.

Wróćmy do problemu, czyli wyświetlania różnych komponentów w zależności od orientacji tabletu. Na rysunku 15.6 widać, co się dzieje po obróceniu urządzenia.

Porównaj rysunki 15.5 i 15.6. Widać, że obszar ze szczegółowymi informacjami wygląda tak samo w obu orientacjach. Różnice są podobne do tych, które znamy z aplikacji na smartfony. Jednak listy ofert wyglądają inaczej. Tego właśnie dotyczy określenie „zupełnie odmienne układy” z opisu problemu. W smartfonach takie podejście stosuje się bardzo rzadko, jednak w aplikacjach na tablety nie jest ono niczym niezwykłym.



Rysunek 15.4. **Lista ofert i szczegółowe informacje w układzie poziomym**

Rysunek 15.5. **Przeglądanie elementów z listy ofert**

Najważniejszym mechanizmem przy tworzeniu aplikacji podobnych do pokazanej są fragmenty. Umożliwiają one podział interfejsu użytkownika na moduły. Na listingu 15.2 przedstawiono kod układu z rysunku 15.4.

Rysunek 15.6. Lista ofert i szczegółowe informacje w układzie pionowym

Listing 15.2. Kod XML układu ze szczegółowymi informacjami o ofercie (/res/layout/details.xml)

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:orientation="horizontal"
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   android:id="@+id/details_container">

   <fragment
      class="com.manning.aip.tabdroid.SectionDetailsFragment"          ❶
         android:id="@+id/section_list_fragment"
         android:visibility="gone"
         android:layout_marginTop="?android:attr/actionBarSize"
         android:layout_width="300dp"
         android:layout_height="match_parent" />

   <fragment class="com.manning.aip.tabdroid.DealFragment"             ❷
         android:id="@+id/deal_fragment"
         android:layout_width="match_parent"
         android:layout_height="match_parent" />

</LinearLayout>
```

Mamy nadzieję, że miłym zaskoczeniem jest dla Ciebie to, jak prosty jest plik układu dla widoku ze szczegółowymi informacjami. Kod obejmuje dwa fragmenty. Pierwszy ❶ wyświetla listę ofert po prawej stronie ekranu. Drugi ❷ pokazuje szczegółowe informacje na temat wybranego elementu. Kod pierwszego fragmentu przedstawiono na listingu 15.3.

Listing 15.3. Fragment wyświetlający listę ofert (plik SectionDetailsFragment.java)

```
public class SectionDetailsFragment extends ListFragment {      ←❶
   Section section;
   int currentPosition = 0;
   DealsApp app;
   @Override
   public void onCreate(Bundle savedInstanceState){
      super.onCreate(savedInstanceState);
      app = (DealsApp) this.getActivity().getApplication();
      section = app.currentSection;

      if (savedInstanceState != null){      ←❷
         currentPosition = savedInstanceState.getInt("currentPosition");
         int savedSectionPos =
            savedInstanceState.getInt("currentSection", -1);
         if (savedSectionPos >= 0){
            section = app.sectionList.get(savedSectionPos);
            app.currentSection = section;
         }
      } else if (app.currentItem != null){
         for (int i=0;i<section.items.size();i++){
            if (app.currentItem.equals(section.items.get(i))){
               currentPosition = i;
               break;
            }
         }
      }
   }

   @Override
   public void onActivityCreated(Bundle savedInstanceState) {      ←❸
      super.onActivityCreated(savedInstanceState);
      buildUi();
   }
   private void buildUi(){
      ListView listView = this.getListView();      ←❹
      listView.setChoiceMode(ListView.CHOICE_MODE_SINGLE);
      String[] dealTitles = new String[section.items.size()];
      int i = 0;
      for (Item item : section.items){
         dealTitles[i++] = item.title;
      }
      setListAdapter(new ArrayAdapter<String>(getActivity(),
            R.layout.deal_title_list_entry, dealTitles));      ←❺
      listView.setSelection(currentPosition);
      showDeal(currentPosition);
   }
}
```

Aby utworzyć fragment, wystarczy utworzyć klasę pochodną od `android.app.Fragment`. Tu tworzymy klasę pochodną od `ListFragment` ❶, która sama jest klasą pochodną od klasy `Fragment`. Klasa `ListFragment` obejmuje jeden widok `ListView` i często służy do wyświetlania list elementów w układzie z podzielonym ekranem, takim jak na rysunku 15.4. Fragment ma odrębny cykl życia powiązany z cyklem życia nadrzędnej aktywności. Aktywność ta żąda widoku od fragmentu przez wywołanie metody `onCreateView` danego fragmentu. Metoda `onCreate` ❷ fragmentu jest wywoływana bezpośrednio po wywołaniu metody `onCreate` aktywności, jednak przed metodą `onCreateView` fragmentu. W metodzie `onCreate` przywracamy lub ustawiamy stan fragmentu (podobnie jak robimy z aktywnością). Pewien czas po wywołaniu metody `onCreateView` fragmentu następuje wywołanie metody `onActivityCreated` ❸. Jak wskazuje nazwa, wywołanie to ma miejsce po utworzeniu aktywności. We wspomnianej metodzie konfigurujemy widok `ListView` ❹ będący częścią fragmentu `ListFragment`. Widok ten działa jak inne widoki `ListView`, dlatego trzeba określić dla niego adapter `ListAdapter` ❺, który zapewni dane i układ elementów z widoku.

Warto zauważyć, że ostatnią operacją w ramach konfigurowania interfejsu użytkownika fragmentu `ListFragment` jest wywołanie metody `showDeal`. Metoda ta wyświetla konkretną ofertę w głównym fragmencie ze szczegółowymi informacjami. Dlatego metodę tę należy wywoływać po wybraniu elementu listy. Na listingu 15.4 przedstawiono kod do wyświetlania ofert i obsługi dotknięcia elementu.

Listing 15.4. Wyświetlanie konkretnej oferty (plik SectionDetailsFragment.java)

```
@Override
public void onListItemClick(ListView l, View v, int position, long id) {
    this.currentPosition = position;          ❶
    showDeal(position);
}

private void showDeal(int position){
   app.currentItem = app.currentSection.items.get(position);
   DealFragment fragment =
      (DealFragment) getFragmentManager().findFragmentById(
         R.id.deal_fragment);                 ❷
   fragment.showCurrentItem();                ❸
}
```

Jednym z wygodnych aspektów korzystania z klasy `ListFragment` jest to, że trzeba przesłonić metodę `onListItemClick`, aby obsługiwać dotknięcie elementów listy. Tu sprawdzamy wybrany element listy ❶. Następnie wywołujemy metodę `showDeal` z listingu 15.3. Drugi fragment ma wtedy wyświetlić inną ofertę, dlatego potrzebny jest uchwyt do tego fragmentu ❷. Do pobrania uchwytu używamy egzemplarza klasy `FragmentManager`, dostępnego w każdym fragmencie. Wróć do listingu 15.2. Zwróć uwagę, że przypisaliśmy do fragmentu identyfikator, który

można teraz wykorzystać do uzyskania uchwytu. Po jego pobraniu należy wywołać metodę `showCurrentItem` ❸, aby nakazać ponowne wyświetlenie fragmentu. Na listingu 15.5 przedstawiono tę metodę i pozostały kod klasy `DealFragment`.

Listing 15.5. Fragment do wyświetlania ofert (plik DealFragment.java)

```
public class DealFragment extends Fragment {          ←❶
   DealsApp app;
   private ProgressBar progressBar;
   @Override
   public View onCreateView(LayoutInflater inflater,
         ViewGroup container,
         Bundle savedInstanceState) {                  ←❷
      app = (DealsApp) getActivity().getApplication();
      View dealView = inflater.inflate(R.layout.deal_details,
         container,
         false);                                       ←❸
      progressBar = (ProgressBar) dealView.findViewById(R.id.progress);
      progressBar.setIndeterminate(true);
      Item item = app.currentItem;
      if (item != null) {
         populateDealView(dealView, item);
      }
      return dealView;
   }
   private void populateDealView(View dealView, Item item) {   ←❹
      ImageView icon = (ImageView) dealView.findViewById(
            R.id.details_icon);
         icon.setImageResource(R.drawable.placeholder);
         new RetrieveImageTask(icon).execute(item.picUrl);
         TextView title =
            (TextView) dealView.findViewById(R.id.details_title);
         title.setText(item.title);
         CharSequence pricePrefix =
            getText(R.string.deal_details_price_prefix);
         TextView price =
            (TextView) dealView.findViewById(R.id.details_price);
         price.setText(pricePrefix + item.convertedCurrentPrice);
         TextView msrp = (TextView) dealView.findViewById(
             R.id.details_msrp);
         msrp.setText(item.msrp);
         TextView quantity =
            (TextView) dealView.findViewById(R.id.details_quantity);
         quantity.setText(Integer.toString(item.quantity));
         TextView quantitySold = (TextView) dealView.findViewById(
            R.id.details_quantity_sold);
         quantitySold.setText(Integer.toString(item.quantitySold));
         TextView location =
            (TextView) dealView.findViewById(R.id.details_location);
         location.setText(item.location);
   }
   public void showCurrentItem(){                    ←❺
      Item item = app.currentItem;
      View dealView = getView();
      populateDealView(dealView, item);
    }
}
```

To kolejny fragment. Tym razem bezpośrednio tworzymy klasę pochodną od klasy `Fragment` ❶. Nie trzeba przejmować się zarządzaniem stanem tego fragmentu, ponieważ jest on powiązany ze stanem aktywności (i fragmentu `ListFragment` z listingu 15.3). Dlatego wystarczy przesłonić metodę `onCreateView` ❷. Zauważ, że do tej wywoływanej zwrotnie metody przekazujemy obiekt klasy `LayoutInflater`. Wykorzystujemy go do przekształcenia pliku XML układu na widok ❸. Następnie wiążemy ❹ dane wybranej oferty z kontrolkami z pliku XML układu. W końcowej części listingu znajduje się metoda `showCurrentItem`, którą mogą wywoływać inne fragmenty. Metoda ta sprawdza, który element jest wybrany, i przekazuje go do używanej już wcześniej metody `populateDealView` ❺.

Po zmianie orientacji tabletu na pionową należy wyświetlić inny układ, widoczny na rysunku 15.6. Najprościej uzyskać ten efekt przez zastosowanie odrębnego pliku XML układu. Kod układu dla orientacji pionowej przedstawiono na listingu 15.6.

Listing 15.6. Układ dla orientacji pionowej (/res/layout-port/details.xml)

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:orientation="vertical"
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   android:id="@+id/details_container"
   android:gravity="bottom">

   <fragment class="com.manning.aip.tabdroid.DealFragment"          ←❶
      android:id="@+id/deal_fragment"
      android:layout_marginTop="?android:attr/actionBarSize"
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
   />

   <fragment class="com.manning.aip.tabdroid.FilmstripFragment"     ←❷
      android:id="@+id/section_filmstrip_fragment"
      android:layout_width="match_parent"
      android:layout_height="300dp"
      android:layout_gravity="bottom"
   />
</LinearLayout>
```

Kod z listingu 15.6 jest podobny do kodu z listingu 15.2. Ponownie wykorzystujemy tu ❶ opisany już fragment `DealFragment`. Przeznaczeniem fragmentów jest właśnie umożliwianie powtórnego wykorzystania kodu. Zauważ, że nie pokazujemy kodu aktywności obejmującej fragmenty. Nie ma takiej potrzeby. Fragmenty są niezależne. W orientacji pionowej zastępujemy klasę `SectionDetailsFragment` klasą `FilmstripFragment` ❷. Kod tej ostatniej znajduje się na listingu 15.7.

Klasa `FilmstripFragment` jest nieco podobna do klasy `SectionDetailsFragment` z listingu 15.3. Obie klasy wyświetlają wszystkie oferty z danej kategorii i umożliwiają dotknięcie oferty w celu wyświetlenia szczegółowych informacji na jej

Listing 15.7. Pozioma lista rysunków używana do wybierania ofert (plik FilmstripFragment.java)

```
public class FilmstripFragment extends Fragment {
   @Override
   public void onCreate(Bundle savedInstanceState){          ❶
      super.onCreate(savedInstanceState);
      // Kod do zarządzania stanem pominięto.
   }
   @Override
   public View onCreateView(LayoutInflater inflater,
         ViewGroup container,
         Bundle savedInstanceState){          ❷
      HorizontalScrollView strip =
         (HorizontalScrollView) inflater.inflate(R.layout.filmstrip,
            container,
            false);
      fillWithPics(strip);
      return strip;
   }
   private void fillWithPics(HorizontalScrollView strip) {
      ViewGroup pics = (ViewGroup) strip.findViewById(R.id.pics);
      if (pics.getChildCount() > 0){
         pics.removeAllViews();
      }
      int i =0;
      for (Item item : section.items){                    ❸
         ImageView imgView = new ImageView(getActivity());
         // Kod do pobierania bitmapy.
         imgView.setOnClickListener(new OnClickListener(){
            @Override
            public void onClick(View img) {
               currentPosition = pos;
               showDeal(pos);          ❹
            }
         });
      }
      showDeal(currentPosition);
   }
    private void showDeal(int position){
      app.currentItem = app.currentSection.items.get(position);
      DealFragment fragment = (DealFragment) getFragmentManager()
         .findFragmentById(R.id.deal_fragment);
      fragment.showCurrentItem();          ❺
   }
}
```

temat w komponencie `DealFragment`. Podobny mechanizm stosujemy w innych fragmentach. Najpierw w metodzie `onCreate` ❶ przywracamy stan. Dalej znajduje się implementacja metody `onCreateView` ❷, która przekształca plik XML układu w zwracany widok. Tym razem nie stosujemy widoku `ListView`, ale „rolkę filmu" — przewijany w poziomie zbiór rysunków (jest to widok `HorizontalScrollView`; zobacz plik */res/layout-port/filmstrip.xml*). Wystarczy zapełnić ten widok ❸ obiektami klasy `ImageView`, obejmującymi bitmapy z rysunkami ofert. Dla każdego widoku `ImageView` trzeba ustawić metodę obsługi zdarzeń ❹ wywoływaną w reakcji

na dotknięcie grafiki przez użytkownika. Zdarzenie to prowadzi do pobrania za pomocą klasy `FragmentManager` uchwytu do fragmentu `DealFragment` i wywołania metody `showCurrentItem` tego fragmentu ❺.

OMÓWIENIE

Fragmenty umożliwiają porządkowanie kodu aplikacji w nowy sposób. Fragment pod wieloma względami może być „samowystarczalny". Samodzielnie zarządza wyświetlanymi danymi i obsługuje swój stan. Fragmenty z przykładowej aplikacji zależą od globalnego stanu aplikacji (obiektu typu `Application`). Można z nich korzystać w dowolnym miejscu omawianego programu, ale już nie poza nim. Jest to celowe. Inna możliwość to utworzenie fragmentów zależnych od usług lub całkowicie niezależnych.

Programiści stosowali ten wzorzec na długo przed wprowadzeniem fragmentów, choć framework Androida tego nie ułatwiał. Jedno z często używanych podejść polegało na tworzeniu komponentów interfejsu użytkownika, które potrafią zarządzać stanem, pobierać dane i wykonywać podobne operacje. Odmianą tego wzorca jest rozwijanie komponentów interfejsu użytkownika bezpośrednio komunikujących się z usługą wykonującą wszystkie skomplikowane operacje. Choć zdaniem niektórych programistów należy unikać takich rozwiązań, ponieważ naruszają paradygmat model-widok-kontroler, opisane wzorce często okazują się przydatne. Załóżmy, że w aplikacji znajduje się nagłówek z informacjami o stanie, na przykład o liczbie nieprzeczytanych wiadomości lub nowych ofert dnia. Przed wprowadzeniem fragmentów często (oprócz łączenia kodu modelu z kodem komponentu interfejsu użytkownika) tworzono klasę aktywności do zarządzania stanem. Następnie aktywność tę stosowano jako klasę bazową dla wszystkich pozostałych aktywności aplikacji. Rozwiązania te mają wady i zalety, przy czym fragmenty pozwalają pisać dużo bardziej przejrzysty kod, niż jest to w przypadku innych podejść.

Technika oparta na bazowej aktywności często służy też do obsługi menu. Taka aktywność może tworzyć menu wyświetlane we wszystkich aktywnościach. Jednym z powodów tworzenia menu na poziomie aplikacji jest to, że zazwyczaj ma ono dla programisty niewielkie znaczenie. Jeśli w menu znajduje się jakaś ważna opcja, i tak trzeba umieścić ją także w innym miejscu ekranu, ponieważ bywa, że użytkownicy nie korzystają z menu. W menu nierzadko znajdują się też typowe opcje, takie jak *O programie*, *Pomoc techniczna*, *Wyrejestruj się* itd. W wersji Honeycomb wprowadzono rozwiązanie znacznie wygodniejsze od menu — pasek akcji (ang. *Action Bar*). W następnej technice opisujemy ten mechanizm i wyjaśniamy, kiedy warto go stosować.

TECHNIKA 90. Pasek akcji

Warto stosować menu w Androidzie. Można umieścić w nim wiele skrótów i przydatnych opcji. Można też udostępniać w nim operacje kontekstowe. Poważnym problemem jest jednak to, że użytkownicy rzadko zaglądają do menu.

Wskutek tego zaczęto tworzyć paski akcji. Często stosuje się je w tym samym celu co samo menu, są jednak skuteczniejsze z uwagi na większą widoczność dla użytkownika.

PROBLEM

Chcemy wyświetlać dodatkowe, ale użyteczne funkcje dostępne w kontekście używanej akurat aktywności. Nie zamierzamy jednak stosować standardowego menu Androida, ponieważ użytkownicy często nie korzystają z niego.

ROZWIĄZANIE

Rozwiązanie polega na zastosowaniu paska akcji. Znajduje się on w górnej części ekranu i jest dobrze widoczny dla użytkowników. Eliminuje to największy kłopot związany z menu. Na rysunku 15.7 pokazano przykładowy pasek akcji w aplikacji na tablety.

Rysunek 15.7. Pasek akcji w akcji

Jak widać, pasek akcji znajduje się w górnej części ekranu. W przykładowym programie na pasku są ikona aplikacji, kilka zakładek i przycisk *Podziel się*. Ikona aplikacji pozwala użytkownikom przejść do głównego ekranu, a zakładki służą do przechodzenia do różnych kategorii ofert dnia z eBaya. Na rysunku 15.8 pokazano, że przycisk *Podziel się* pozwala „podzielić się" ofertą z innymi osobami za pomocą aplikacji zainstalowanych w urządzeniu.

Jak może pamiętasz, w pierwszej wersji aplikacji DealDroid funkcja „dzielenia się" była ukryta w menu. W wersji dla tabletów nawigacja jest wygodniejsza. Pasek akcji nie tylko pozwala rozwiązać problem z menu, ale ma też inne funkcje. Zakładki nawigacyjne omawiamy dalej. Teraz skupimy się na ikonach aplikacji i funkcji „dzielenia się". Na listingu 15.8 przedstawiono kod tych elementów.

Listing 15.8. Ikona aplikacji i funkcji „dzielenia się" z paska akcji (plik DetailsActivity.java)

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
   MenuInflater inflater = getMenuInflater();
   inflater.inflate(R.menu.details_menu, menu);  ←1
   return true;
}
```

Rysunek 15.8. „Dzielenie się" ofertą na tablecie

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
   switch (item.getItemId()) {
      case android.R.id.home:                                    ❷
         Intent intent = new Intent(this, DealsMain.class);      ❸
         intent.addFlags(Intent.FLAG_ACTIVITY_CLEAR_TOP);
         startActivity(intent);
         return true;
      case R.id.share_action:                                    ❹
        shareDealUsingChooser("text/*");
        return true;
     default:
        return super.onOptionsItemSelected(item);
   }
}

private void shareDealUsingChooser(final String type) {
// Z uwagi na zwięzłość pominięto. Kod jest taki sam jak w rozdziale 2.
 }

private String createDealMessage() {
// Z uwagi na zwięzłość pominięto.
 }
```

Na listingu 15.8 widać, że przodkiem paska akcji jest menu. Aby utworzyć pasek akcji, należy zaimplementować wywoływaną zwrotnie metodę `onCreateOptionsMenu` aktywności ❶. Elementy paska akcji można tworzyć programowo. Jest to przydatne zwłaszcza wtedy, gdy wyświetlanie elementów zależy od stanu aktywności. Inna możliwość to określenie zawartości paska akcji w XML-u. Oto kod w XML-u tworzący pasek akcji z rozdziału 15.7:

```
<menu xmlns:android="http://schemas.android.com/apk/res/android">
   <item android:id="@+id/share_action"
      android:title="@string/deal_details_share_menu"
      android:icon="@drawable/ic_menu_share"
      android:showAsAction="ifRoom|withText" />
</menu>
```

Jak widać, określony jest tu jeden element z tytułem i ikoną. W przykładzie tytuł i ikona to zewnętrzne zasoby, dlatego można utworzyć ich wersje dla innych języków. Zwróć też uwagę na atrybut `showAsAction`. Pojawił się on w Androidzie 3.0 i służy do określenia, kiedy dana opcja menu ma być dostępna jako akcja i jak ma wyglądać. Można ustawić ten atrybut na `always`, jednak jeśli na ekranie brakuje miejsca, pasek wygląda nieelegancko.

Wróćmy do listingu 15.8. Aby zdefiniować działanie paska akcji (reakcję na dotknięcie opcji przez użytkownika), należy zaimplementować metodę `onOptionsItemSelected` aktywności. W ten sam sposób określana jest reakcja na dotknięcie ikony aplikacji, widocznej po lewej stronie paska ❷. Ikona ta jest identyfikowana na podstawie predefiniowanego identyfikatora zasobu (`home`). Aplikacja w reakcji na wybranie tej ikony opróżnia stos aktywności i kieruje użytkownika do głównego ekranu ❸. Dotknięcie przycisku *Podziel się* można wykryć przez dopasowanie identyfikatora zdefiniowanego w XML-owym kodzie menu do identyfikatora wybranego elementu `MenuItem` ❹. Wybranie wspomnianego przycisku prowadzi do wywołania metody `shareDealUsingChooser` z aplikacji DealDroid z rozdziału 2. Aplikacja wyświetla wtedy interfejs użytkownika widoczny na rysunku 15.8.

Wiesz już, jak tworzyć ikony i określać ich działanie. Przyjrzyjmy się teraz, jak tworzyć zakładki widoczne na rysunku 15.7. Potrzebny kod pokazano na listingu 15.9.

Listing 15.9. Tworzenie zakładek paska akcji i zarządzanie nimi (plik DetailsActivity.java)

```
public void onCreate(Bundle savedInstanceState) {
   super.onCreate(savedInstanceState);
   setContentView(R.layout.details);
   app = (DealsApp) getApplication();
   ActionBar bar = this.getActionBar();          ❶
   TabListener listener = new TabListener(){
      @Override
      public void onTabReselected(Tab t, FragmentTransaction txn) {}
      @Override
      public void onTabSelected(Tab t, FragmentTransaction txn) {
         if (active){
            changeTab(t.getPosition());          ❷
         }
      }
      @Override
      public void onTabUnselected(Tab t, FragmentTransaction txn) {}
   };
   for (int i=0;i<Math.min(6, app.sectionList.size());i++){
```

```
        final Section section = app.sectionList.get(i);
        Tab tab = bar.newTab();                                    ❸
        tab.setText(chomp(section.title));
        tab.setTabListener(listener);                              ❹
        if (app.currentSection != null &&
              app.currentSection.equals(section)){
           bar.addTab(tab, true);
        } else {
           bar.addTab(tab);
        }
     }
     bar.setDisplayShowTitleEnabled(false);
     bar.setNavigationMode(ActionBar.NAVIGATION_MODE_TABS);
     active = true;
  }
  private void changeTab(int position){
     FragmentManager fm = getFragmentManager();
     int orientation = getResources().getConfiguration().orientation;
     if (orientation == ORIENTATION_LANDSCAPE){                    ❺
        SectionDetailsFragment fragment =
           (SectionDetailsFragment) fm.findFragmentById(
               R.id.section_list_fragment);
        fragment.setSection(position);                             ❻
     } else {
        FilmstripFragment fragment =
          (FilmstripFragment) fm.findFragmentById(
              R.id.section_filmstrip_fragment);
        fragment.setSection(position);
     }
  }
```

Od wersji Android 3.0 każda aktywność może mieć pasek akcji. Jest on dostępny poprzez metodę `getActionBar` aktywności ❶. Pomysł polega na programowym tworzeniu zakładek i dodawaniu ich do paska akcji. Każda zakładka wymaga odbiornika `TabListener`, który będzie reagował na dotknięcie, dlatego tworzymy jeden taki odbiornik do obsługi wszystkich zakładek. Dalej znajduje się implementacja metody `onTabSelected` ❷ i wywołanie metody `changeTab` na podstawie pozycji wybranej zakładki. Działanie metody `changeTab` omawiamy dalej.

Kiedy egzemplarz odbiornika `TabListener` jest już gotowy, można utworzyć zakładki i dodać je do paska akcji. Programowo tworzymy zakładkę ❸, ustawiamy jej tytuł i odbiornik `TabListener` ❹. Zauważ, że na podstawie kategorii wybranej przez użytkownika aplikacja określa obecnie zaznaczoną zakładkę. Nakazujemy też paskowi akcji, aby nie wyświetlał nazwy aktywności, a w zamian pokazywał zakładki nawigacyjne.

Przyjrzyjmy się teraz metodzie `changeTab` wywoływanej przez zwrotną metodę `onTabSelected` odbiornika `TabListener`. Metoda `changeTab` najpierw sprawdza orientację urządzenia ❺. Jest to potrzebne, ponieważ układ tabletu wpływa na zawartość aktywności. Metoda wykorzystuje informację o orientacji i obiekt klasy `FragmentManager` z aktywności do uzyskania uchwytu do wyświetlanego fragmentu. Następnie ustawiamy kategorię dla fragmentu ❻, co pozwala określić, jakiego rodzaju oferty dnia mają być widoczne.

OMÓWIENIE

Nawigacja z wykorzystaniem zakładek nie jest niczym nowym ani specjalnym dla tabletów. Od lat jest powszechnie używana w aplikacjach sieciowych i występuje w Androidzie od wersji 1.0. Do tworzenia zakładek zawsze służyły klasy `TabHost` i `TabWidget`. Pierwsza z nich umożliwia tworzenie zestawu zakładek, z których każda powiązana jest z wyświetlaną aktywnością. Zakładki paska akcji to rozwinięcie tej techniki, podobnie jak inne aspekty tego paska są rozwinięciem menu.

Aby zbudować nawigację opartą na zakładkach z paska akcji, należy utworzyć zakładki w podobny sposób jak w klasie `TabHost`. Jednak zamiast łączyć z każdą zakładką odrębną aktywność, można pracować w ramach jednej aktywności i stosować fragmenty. W przykładzie zmieniamy zawartość fragmentu. Zwróć jednak uwagę na to, że do metody `onTabSelected` przekazywany jest obiekt klasy `FragmentTransaction`. Dlatego w aktywności można wykonywać różne operacje na fragmentach, na przykład usuwać je lub zastępować innymi. Pasek akcji nie tylko jest ulepszeniem dawnego systemu menu, ale w połączeniu z fragmentami sprawia też, że porządkowanie kodu aplikacji jest prostsze, i daje przy tym więcej możliwości.

Ostatnia z podstawowych technik związanych z tabletami, przeciąganie, pozwala usprawnić interakcję użytkowników z aplikacją.

TECHNIKA 91. Przeciąganie

Odkąd Douglas Engelbart wymyślił mysz komputerową, menedżerowie produktu żądają od programistów dodawania funkcji przeciągania. W rozbudowanych frameworkach do tworzenia aplikacji desktopowych mechanizm przeciągania jest dostępny od wielu lat. W aplikacjach sieciowych przez długi czas występowały znaczne problemy z jego obsługą. Pracę programistom ułatwiały frameworki JavaScriptu, aż w końcu mechanizm przeciągania stał się częścią specyfikacji języka HTML5. W świecie urządzeń mobilnych do momentu pojawienia się Androida 3.0 przeciąganie było w frameworkach pomijane. Oczywiście, można było dodać jego obsługę za pomocą interfejsów API do obsługi dotknięć, jednak technikę tę wykorzystywano głównie w grach. Od wersji Honeycomb przeciąganie można stosunkowo łatwo dodać do aplikacji dowolnego rodzaju.

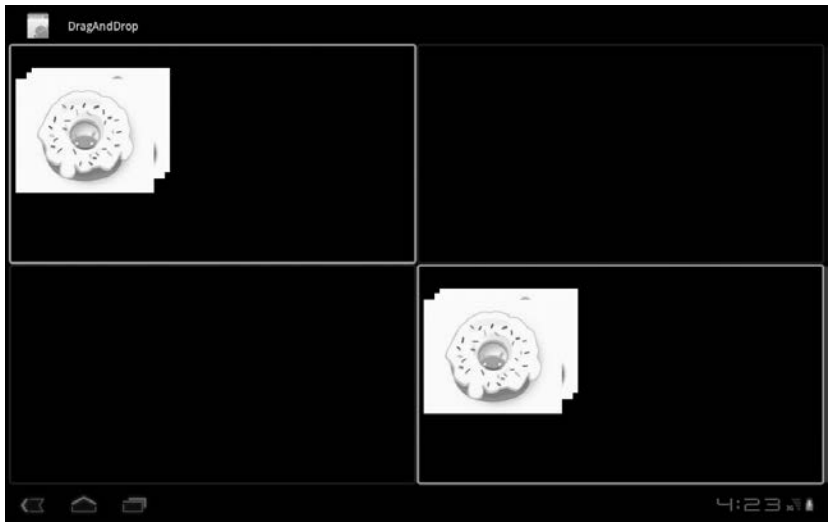
PROBLEM

Chcemy umożliwić użytkownikom bardziej intuicyjną interakcję z aplikacją przez udostępnienie przeciągania różnych elementów.

ROZWIĄZANIE

Aby umożliwić przeciąganie w aplikacji, wystarczy zastosować kilka interfejsów API wprowadzonych w Androidzie 3.0. W ramach przykładu przedstawiamy prostą aplikację z funkcją przeciągania. Program wyświetla na ekranie widoki

`StackView` (jest to nowa kontrolka wprowadzona w wersji Honeycomb) i umożliwia użytkownikom zmianę uporządkowania tych kontrolek przez ich przeciągnięcie. Wygląd aplikacji przedstawiono na rysunku 15.9.



Rysunek 15.9. Aplikacja z funkcją przeciągania

Jak widać, aplikacja wyświetla prostą siatkę z kilkoma kontrolkami `StackView`. Na listingu 15.10 znajduje się kod tego układu.

Listing 15.10. Plik XML z układem z siatką, używanym przez mechanizm przeciągania

```
<?xml version="1.0" encoding="utf-8"?>
<TableLayout                                          ❶
   xmlns:android="http://schemas.android.com/apk/res/android"
   android:layout_width="match_parent"
   android:layout_height="match_parent">
   <TableRow>
      <LinearLayout android:layout_width="640dp"
         android:layout_height="345dp"
         android:id="@+id/topLeft">
            <StackView android:id="@+id/stack"          ❷
               android:layout_width="250dp"
               android:layout_height="250dp"
               android:clickable="true"
               android:loopViews="true"
               android:longClickable="true"
      />                                              ❸
      </LinearLayout>
      <LinearLayout android:layout_width="640dp"
         android:layout_height="345dp"
         android:id="@+id/topRight"
      />
   </TableRow>
   <TableRow>
```

```
      <LinearLayout android:layout_width="640dp"
         android:layout_height="345dp"
         android:id="@+id/bottomLeft"
      />
      <LinearLayout android:layout_width="640dp"
         android:layout_height="345dp"
         android:id="@+id/bottomRight">
         <StackView android:id="@+id/stack2"
            android:layout_width="250dp"
            android:layout_height="250dp"
            android:clickable="true"
            android:loopViews="true"
            android:longClickable="true"
         />
      </LinearLayout>
   </TableRow>
</TableLayout>
```

W kodzie z listingu 15.10 używamy układu `TableLayout` ❶. Nie ma w nim nic wyjątkowego. Tu pozwala łatwo zidentyfikować różne części ekranu na potrzeby pokazu przeciągania. Każda komórka tabeli obejmuje układ `LinearLayout` ❷. Także on nie ma żadnych specjalnych cech. Potrzebujemy tylko kontenera, do którego można przeciągać widoki `StackView` ❸. Również te widoki nie mają żadnych specjalnych cech w kontekście przeciągania. Co więcej, prościej byłoby ich nie używać, jednak są ciekawe wizualnie, dlatego warto z nich korzystać w aplikacjach na tablety.

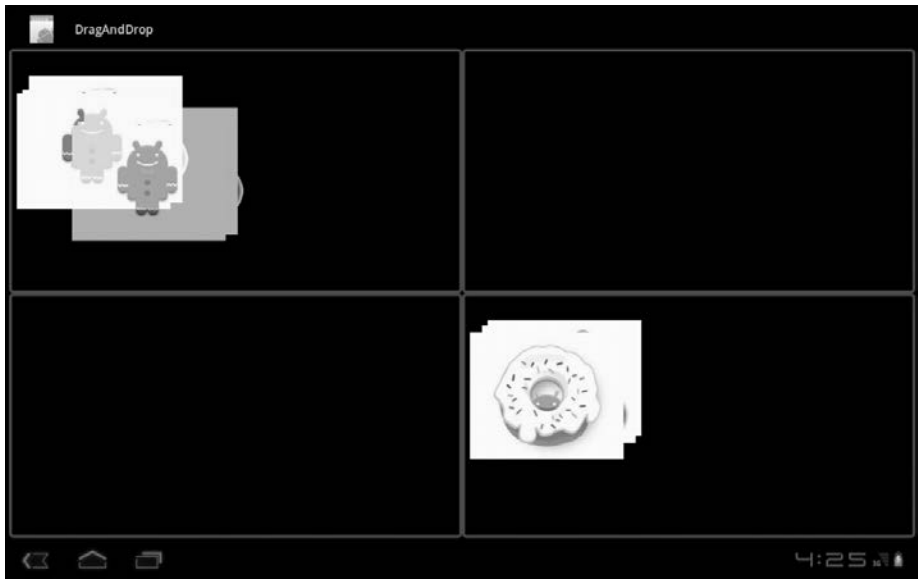
Najważniejszą cechą aplikacji jest umożliwianie użytkownikom przeciągania widoków `StackView` do różnych kontenerów (układów `LinearLayout`) na ekranie. Aby rozpocząć przeciąganie, użytkownik musi dotknąć i przytrzymać stos (długie kliknięcie), co powoduje podświetlenie miejsc, w których można upuścić elementy. Na rysunku 15.10 pokazano takie miejsca.

Wszystkie pozostałe operacje w aplikacji wykonujemy programowo. Na listingu 15.11 znajduje się kod tworzący interfejs użytkownika.

Listing 15.11. Tworzenie interfejsu użytkownika

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

      StackView stack = (StackView) findViewById(R.id.stack);  ←❶
      Bitmap[] bmps = new Bitmap[5];
      Resources res = getResources();          ←❷
      bmps[0] = BitmapFactory.decodeResource(res, R.drawable.donut);
      bmps[1] = BitmapFactory.decodeResource(res, R.drawable.eclair);
      bmps[2] = BitmapFactory.decodeResource(res, R.drawable.froyo);
      bmps[3] =
         BitmapFactory.decodeResource(res, R.drawable.gingerbread);
      bmps[4] =
         BitmapFactory.decodeResource(res, R.drawable.honeycomb);
```



Rysunek 15.10. Aktywne obszary, w których można upuścić elementy

```
        ImgAdapter adapter = new ImgAdapter(bmps, stack);          ←❸
        stack.setAdapter(adapter);

        StackView stack2 = (StackView) findViewById(R.id.stack2);
        stack2.setAdapter(new ImgAdapter(bmps, stack2));

        findViewById(R.id.topLeft).setOnDragListener(
           new BoxDragListener());                                 ←❹
        findViewById(R.id.bottomLeft).setOnDragListener(
           new BoxDragListener());
        findViewById(R.id.topRight).setOnDragListener(
           new BoxDragListener());
        findViewById(R.id.bottomRight).setOnDragListener(
           new BoxDragListener());
    }
}
```

Pierwszą rzeczą, jaką trzeba zrobić w celu utworzenia interfejsu użytkownika, jest uzyskanie referencji do jednego z widoków `StackView` ❶. Następnie aplikacja wczytuje zasoby graficzne będące częścią aplikacji ❷. Na podstawie rysunków 15.9 i 15.10, a także kodu z listingu 15.11 można stwierdzić, że rysunki używane w widokach `StackView` to ikony reprezentujące różne wersje Androida (aż do wersji Honeycomb). Rysunki te są przekształcane na obiekty klasy `Bitmap` i przekazywane do niestandardowego adaptera widoku `StackView` ❸ (kod tego adaptera przedstawiono na listingu 15.12). `StackView` to oparta na adapterze kontrolka podobna do widoków `ListView` i `GridView`. Pobieramy uchwyty do kontenerów `LinearLayout` z listingu 15.10 i przekazujemy im obiekt typu `OnDragListener` ❹.

OnDragListener to nowy interfejs wprowadzony w Androidzie 3.0. Jego implementację przedstawiono na listingu 15.13. Teraz przyjrzyjmy się niestandardowemu adapterowi widoków StackView.

Listing 15.12. Adapter używany dla widoków StackView w aplikacji

```
class ImgAdapter extends BaseAdapter{ ←❶
   private Bitmap[] bmps;
   private Context ctx = DndActivity.this;
   private ViewGroup owner;
   ImgAdapter(Bitmap[] bmps, ViewGroup owner){
      this.bmps = bmps;
      this.owner = owner;
   }
   @Override
   public int getCount() {
      return bmps.length;
   }
   @Override
   public Object getItem(int index) {
      return bmps[index];
   }
   @Override
   public long getItemId(int index) {
      return index;
   }
   @Override
   public View getView(int index, View recycledView, ViewGroup parent) {
      if (recycledView == null){
         recycledView = new ImageView(ctx);
      }
      ImageView imgView = (ImageView) recycledView;
      imgView.setOnLongClickListener(new OnLongClickListener(){
         @Override
         public boolean onLongClick(View view) { ←❷
            ClipData data =
               ClipData.newPlainText("foo","bar");
            DragShadowBuilder sBuilder =
               new DragShadowBuilder(owner);
            owner.startDrag(data, sBuilder, owner, 0); ←❸
            return true;
         }
      });
      imgView.setImageBitmap(bmps[index]);
      return imgView;
   }
}
```

Wspomnieliśmy już, że widok StackView przypomina widoki ListView i GridView. Wszystkie te widoki korzystają z adaptera tego samego rodzaju, dlatego tu tworzymy klasę pochodną od BaseAdapter ❶. Jej kod jest podobny do kodu innych adapterów. Adapter ImgAdapter z listingu 15.12 można zastosować także dla widoku ListView lub GridView. Jest on przeznaczony nie tylko dla widoków StackView. Jedyny wyjątkowy kod służy do obsługi przeciągania. Najpierw z widokiem

StackView wiążemy odbiornik OnLongClickListener ❷. Przeciąganie chcemy inicjować w momencie długiego kliknięcia, wykrywanego właśnie przez ten odbiornik. Nie chcemy przeciągać poszczególnych widoków ImageView z widoku StackView — interesuje nas przeciąganie całego widoku StackView. To dlatego przechowujemy referencję owner do tego widoku. Dalej wywołujemy metodę startDrag ❸, dodaną do klasy android.view.View w wersji Honeycomb Androida. Metoda ta początkuje przeciąganie widoku StackView. Zauważ, że jednym z parametrów metody startDrag jest egzemplarz klasy DragShadowBuilder. Klasa ta odpowiada za wyświetlanie cienia widoku przeciąganego po ekranie. Tu używamy domyślnej wersji klasy DragShadowBuilder. Wyświetla ona widok przekazany do konstruktora tej klasy. Wyświetlanie takiego widoku może wymagać dużo zasobów. Jeśli tak jest, warto utworzyć klasę pochodną od DragShadowBuilder i wyświetlać w niej niestandardowy cień. Istnieje też inne rozwiązanie — można zastosować domyślną wersję klasy DragShadowBuilder, jednak nie przekazywać widoku do jej konstruktora. Wtedy klasa nie wyświetla żadnego cienia.

Do czego służy klasa ClipData?

Może zauważyłeś, że na listingu 15.13 przekazaliśmy do metody startDrag obiekt klasy ClipData. Jeśli tak, prawdopodobnie zwróciłeś uwagę, że umieściliśmy w tym obiekcie fikcyjne dane. Klasa ClipData jest przydatna, kiedy widok reprezentuje skomplikowane dane przeciągane w aplikacji. Widok może na przykład wyświetlać obiekt z danymi kontaktowymi, które aplikacja pomaga porządkować. Klasa ClipData pozwala w wygodny sposób powiązać z przeciąganym widokiem wskaźnik do złożonego obiektu. Ponadto klasa ta jest potrzebna do udostępniania niestandardowego mechanizmu kopiowania i wklejania.

Po zainicjowaniu przeciągania widoku StackView w aplikacji trzeba dodać obsługę upuszczania go w jednym z kontenerów LinearLayout. Na listingu 15.13 pokazano, jak to zrobić za pomocą ustawionego wcześniej odbiornika OnDragListener.

Listing 15.13. Implementacja interfejsu OnDragListener dla kontenerów

```
class BoxDragListener implements OnDragListener{
   boolean insideOfMe = false;          ←❶
   Drawable border = null;
   Drawable redBorder = getResources().getDrawable(R.drawable.border3);
   @Override
   public boolean onDrag(View self, DragEvent event) {
      if (event.getAction() == DragEvent.ACTION_DRAG_STARTED){
         border = self.getBackground();
         self.setBackgroundDrawable(redBorder);          ←❷
      } else if (event.getAction() == DragEvent.ACTION_DRAG_ENTERED){
         insideOfMe = true;
      } else if (event.getAction() == DragEvent.ACTION_DRAG_EXITED){
         insideOfMe = false;
      } else if (event.getAction() == DragEvent.ACTION_DROP){
         if (insideOfMe){
            View view = (View) event.getLocalState();
```

```
            ViewGroup owner = (ViewGroup) view.getParent();
            owner.removeView(view);                              ❸
            LinearLayout container = (LinearLayout) self;
            if (container.getChildCount() > 0){
               container.addView(view,
                  container.getChildCount());                    ❹
            } else {
               container.addView(view);
            }
         }
      } else if (event.getAction() == DragEvent.ACTION_DRAG_ENDED){
         self.setBackgroundDrawable(border);                     ❺
      }
      return true;
   }
}
```

W każdym kontenerze w aplikacji używamy egzemplarza klasy z implementacją interfejsu `OnDragListener` (listing 15.13). Kontener musi śledzić, czy przeciągany widok się w nim znajduje. Informacja ta jest przechowywana w zmiennej logicznej ❶. W momencie rozpoczęcia przeciągania obramowanie kontenera ustawiamy na kolor czerwony ❷. Jest to informacja, że w danym kontenerze można upuścić przeciągany widok. Następnie śledzimy zdarzenia `ENTERED` i `EXIT` za pomocą zmiennych logicznych. Po zgłoszeniu upuszczenia trzeba zrobić dwie rzeczy. Najpierw należy usunąć przeciągany widok z poprzedniego kontenera ❸. Później trzeba dodać przeciągany widok do nowego kontenera ❹. Po wykryciu zakończenia przeciągania należy przywrócić domyślny wygląd obramowania ❺. Warto zauważyć, że nie ma tu znaczenia, jakiego rodzaju widok jest przeciągany do kontenera. Ponadto jeśli dany kontener nie ma obsługiwać mechanizmu przeciągania, wystarczy nie ustawiać odbiornika `OnDragListener`. Kontener bez tego odbiornika nie ma czerwonego obramowania i nie przyjmuje upuszczonych widoków.

OMÓWIENIE

Nie bez powodu programiści nieustannie są proszeni o dodawanie obsługi przeciągania do aplikacji. W programach dowolnego rodzaju przeciąganie pozwala w bardziej intuicyjny sposób komunikować się z aplikacją. W przeszłości przeciąganie odbywało się za pomocą myszy w aplikacjach desktopowych oraz sieciowych — i nawet przy korzystaniu z tego urządzenia operacja ta była intuicyjna.

Obecnie nastała era urządzeń dotykowych, z których jako pierwsze pojawiły się smartfony. Jednak przeciąganie rzadko było w nich stosowane, choć nie z powodu braku potrzebnych mechanizmów w Androidzie. Po prostu operacja ta na małych ekranach jest niewygodna. Miejsca, w których można dotknąć ekran, aby rozpocząć przeciąganie, są zwykle małe. Jednak na tabletach sytuacja wygląda zupełnie inaczej. Przeciąganie jest w nich dużo wygodniejsze niż w smartfonach. Tablety mają duże wyświetlacze, dlatego można precyzyjnie wskazać przeciągany element. Ponadto przeciąganie za pomocą interfejsów dotykowych jest

znacznie wygodniejsze niż posługiwanie się myszą. Przy przeciąganiu z wykorzystaniem myszy pojawia się ikona dłoni, co ma imitować przenoszenie elementów w rzeczywistym świecie. Przeciąganie na tabletach nie wymaga żadnego imitowania, ponieważ to prawdziwa, ludzka dłoń służy do interakcji z obiektami na ekranie.

Możliwości tworzenia intuicyjnych interfejsów opartych na przeciąganiu w aplikacjach na tablety są atrakcyjne i niemal nieskończone. Trzeba jednak pokonać kilka przeszkód. Ponieważ przeciąganie nie było popularne w urządzeniach dotykowych (smartfonach), nie istnieją powszechnie przyjęte sposoby informowania użytkowników o tym, że aplikacja obsługuje tę technikę. W przykładowej aplikacji początkiem przeciągania jest dotknięcie ekranu i przytrzymanie palca (czyli *długie kliknięcie*, jak określają to programiści aplikacji na Android). Nie chcemy rozpoczynać przeciągania w reakcji na każde dotknięcie widoku `StackView`. To uniemożliwiłoby użytkownikom przeglądanie rysunków z widoku `StackView`. Ponadto irytująca byłaby sytuacja, gdyby każde dotknięcie ekranu skutkowało wyświetlaniem czerwonego obramowania wokół kontenerów. Nie twierdzimy jednak, że długie kliknięcie powinno być powszechnie przyjętym sposobem początkowania przeciągania w aplikacjach na tablety. Inna możliwość to dodanie przycisku *Edycja* (na przykład na pasku akcji) do włączania trybu przeciągania, w którym dotknięcie początkuje przeciąganie obiektu.

15.3. Podsumowanie

Z pewnością wiesz, że Android nie był pierwszą platformą na smartfony, którą przeniesiono na tablety. Sukces Androida w dotykowych smartfonach sprawił, że zastosowanie go w tabletach było oczywiste. Jednak, jak zobaczyłeś w tym rozdziale, twórcy tej platformy nie spoczęli na laurach i wprowadzili szereg istotnych usprawnień z myślą o tabletach. Najważniejszą nowinką było zastosowanie fragmentów. Wspomnieliśmy już, że potrzeba tworzenia bardziej niezależnych komponentów w Androidzie nie jest warunkowana tylko tym, że pojawiły się tablety. Programiści odczuwali ją już wcześniej i wymyślali rozmaite rozwiązania problemu. Jednak na tabletach chciano tworzyć interfejsy użytkownika lepiej wykorzystujące dużą powierzchnię ekranu. Fragmenty to umożliwiły. Obecnie fragmenty (a także inne ważne mechanizmy programowania aplikacji na tablety — pasek akcji i przeciąganie) są dostępne także we wcześniejszych wersjach Androida poprzez pakiet Android Compatibility.

Warto zauważyć, że rozdział ten nie jest wyczerpującym omówieniem wszystkich zmian wprowadzonych w wersji Honeycomb Androida. Większość modyfikacji ma pomagać programistom pisać aplikacje na tablety, dlatego pasuje do tego rozdziału. Zdecydowaliśmy się jednak skoncentrować na najważniejszych technikach, które powinni znać autorzy wszystkich aplikacji na Android. Nie chcemy przy tym umniejszać znaczenia innych funkcji. Pominęliśmy tu na przykład kilka nowych rozwiązań (m.in. środowisko Renderscript) bardzo przydat-

nych dla twórców gier. Nie omówiliśmy też usprawnień klasy `RemoteView`, ulepszających obsługę kontrolek i powiadomień na ekranie głównym. Wszystkie te funkcje są istotne, a w niektórych sytuacjach niemal niezastąpione. Szczegółowy opis ulepszeń w interfejsach API i działaniu mechanizmów znajdziesz, jak zawsze, w dokumentacji Androida. W ten sposób doszliśmy do końca książki! Zacząłeś od utworzenia aplikacji HelloAndroid w rozdziale 1., przebrnąłeś przez wiele zaawansowanych zagadnień i ponad 80 technik, a na końcu zapoznałeś się z tabletami. W tym momencie masz bardzo solidne podstawy do tworzenia aplikacji na Android. Możesz sobie pogratulować!

Skorowidz

A

B

C

D

E

F

G

H

I

J

K

L

Ł

M

N

O

P

R

S

Ś

T

U

W

X

Z

Ż

Nietrudno jest znaleźć informacje potrzebne do stworzenia pierwszej aplikacji na Androida. Ale co dalej? Im głębiej zanurzymy się w świat urządzeń mobilnych, tym częściej będziemy trafiać na pułapki i ślepe uliczki. Na ratunek przychodzi książka *Android w praktyce*. Nie znajdziesz tu jednak żadnych banalnych informacji. Ta książka to bogate źródło wskazówek, sztuczek i najlepszych praktyk z obszaru tworzenia aplikacji na Androida, obejmująca ponad dziewięćdziesiąt pomysłowych i przydatnych technik, dzięki którym staniesz się lepszym programistą.

W trakcie lektury zobaczysz, jak tworzyć precyzyjne (co do piksela) elementy graficzne, zarządzać zadaniami wykonywanymi w tle oraz równoległymi wątkami. Ponadto sprawdzisz, jak współużytkować dane między aplikacjami oraz komunikować się z usługami sieciowymi. To tylko niektóre z tematów poruszonych w tej wyjątkowej książce, poświęconej platformie Android. Przeznaczona zarówno dla początkujących, jak i zaawansowanych użytkowników, pomoże Ci ona zrozumieć, jak budować doskonałe aplikacje, które przez lata będą odnosić sukcesy na platformie Android.

Sięgnij po tę książkę i...

- poznaj najlepsze praktyki tworzenia aplikacji na platformę Android
- wykorzystaj potencjał wielordzeniowych procesorów
- twórz grafikę dwu- i trzywymiarową
- znajdź rozwiązania Twoich problemów

Obowiązkowa pozycja na półce programisty aplikacji mobilnych!

Charlie Collins jest programistą aplikacji mobilnych i sieciowych w firmie MOVL, współtwórcą kilku projektów o otwartym dostępie do kodu źródłowego, a także współautorem książek *GWT in Practice* i *Unlocking Android*.

Michael Galpin jest programistą w firmie Bump Technologies, gdzie pracował nad dwiema spośród najbardziej popularnych aplikacji ze sklepu Android Market (Bump i eBay Mobile).

Matthias Kaeppler jest inżynierem odpowiedzialnym za Androida i interfejsy API w firmie Qype.

helion.pl
księgarnia internetowa

Nr katalogowy: 10480

Księgarnia internetowa:
http://helion.pl

Zamówienia telefoniczne:
0 801 339900
0 601 339900

Helion

Sprawdź najnowsze promocje:
http://helion.pl/promocje
Książki najchętniej czytane:
http://helion.pl/bestsellery
Zamów informacje o nowościach:
http://helion.pl/nowosci

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
http://helion.pl

Informatyka w najlepszym wydaniu

ISBN 978-83-246-4810-8

9 788324 648108

Cena: 99,00 zł