**Waldo Low-level Design**

**1. Descriere sumară a funcţionalităţilor**

Aplicaţia detectează magazine de bicicletă din aproprierea utilizatorului. Pentru aceasta, trebuie să aibă datele pornite şi GPS-ul. Magazinele sunt afişate sub forma unei liste. Dacă se dă click pe un magazin, se deschide harta cu poziţia lui şi a utilizatorului. Dacă doreşte, poate primi comenzi vocale de navigaţie până la magazin prin intermediul Google Maps.

Din aplicaţie se pot şi adăuga magazine de bicicletă. Pe moment, datele introduse de utilizator vor fi trimise către un server, iar adăugarea efectivă se va face printr-un script care interacţionează cu interfaţa Google.

Cazurile de utilizare, precum şi scheme ale ecranelor se găsesc pe Dropbox.

**2. Clase şi metode**

**2.1 Pachetul waldo**

Pachetul principal ce conţine activităţile de bază - Main, Settings, Maps (cea cu toate magazinele/un singur magazin) şi SplashScreen. Înglobează şi fragmentul din activitatea principală.

**2.1.1 SplashScreen.java**

Primul ecran care se afişează utilizatorului este logoul Waldo. Clasa aferentă este SplashScreen.java, aflată în pachetul waldo.bike.waldo. Logica este următoarea: la deschiderea aplicaţiei se verifică să fie pornite datele şi activat GPS-ul. Starea reţelei este monitorizată printr-un Broadcast Receiver **declarat dinamic** în onResume() şi desfiinţat în onPause(). Iniţial, acesta era declarat în manifest, însă e o problemă cu această abordare: la fiecare schimbare a stării reţelei, aplicaţia era adusă în prim-plan, deoarece primea semnalul de la sistem. Având un receptor dinamic, acesta nu mai există când aplicaţia nu este în prim-plan, deci nu mai există riscul de aşa ceva.

De reţinut că onReceive(), metoda din BroadcastReceiver, este apelată şi la declalarea receptorului, adică de fiecare dată când activitatea revine în prim-plan. În onResume(), prima dată facem verificarea pentru GPS, apoi înregistrăm receptorul, moment în care se apelează onReceive(), iar noi facem verificările pentru Internet. Folosim două variabile de stare, *isInternetEnabled* şi *isGPSEnabled*, pentru a şti la orice moment starea în care se află cele două componente. În momentul în care ambele sunt adevărate, pornim activitatea principală, fie din onResume() (pentru cazul în care GPS-ul este pornit când se lansează aplicaţia, dar nu şi internetul), fie din onReceive() (când se intră aici, verificarea pentru GPS este deja făcută).

**2.1.2 MainActivity**

Activitatea principală din aplicaţie. La pornire, se face conectarea la GPS (fiind deja pornit din activitatea precedentă). În onCreate() se defineşte, iar în onStart() se conectează. Clasa implementează interfeţele GoogleApiClient.ConnectionCallbacks, GoogleApiClient.OnConnectionFailedListener și LocationListener pentru partea de GPS. În onConnected(), apelată de sistem după ce se iniţializează conexiunea, se stabilesc detaliile de conectare (precizia - înaltă, frecvenţa actualizării - o secundă) şi se cer actualizări ale locaţiei. Acestea sunt transmise în onLocationChanged(), unde se poate converti la String.

Am luat în calcul varianta în care utilizatorul iese din aplicaţie şi închide GPS-ul. Tratăm acest caz în onResume(), apelată când MainActivity revine pe ecran, prin apelarea lui checkGpsEnabled() din DeviceConnection.

Comportamentul GPS la schimbare ecranului

Am definit variabila previousOrientation pentru a controla comportamentul în caz că orientarea ecranului se schimbă. Este inițializată cu 0, iar în onCreate() ia valoarea orientării actuale a ecranului (landscape = 2; portrait = 1) prin apelarea metodei getScreenOrientation(). În onPause(), care se apelează când altă activitate apare pe ecran,previousOrientation ia valoarea 4, iar deconectarea se face în onStop() doar dacă previousOrientation == 4 (adică e altă activitate pe ecran). Vezi stările unei activități pentru a înțelege complet logica: http://developer.android.com/reference/android/app/Activity.html .

Starea rețelei

Ca în SplashScreen, receptorul este înregistrat dinamic în onResume() şi desfiinţat în onPause(). Nu mai avem treabă cu manifestul.

Monitorizarea stării GPS-ului şi a reţelei

Dacă utilizatorul părăseşte aplicaţia şi dezactivează Internetul, GPS-ul sau amândouă, un mesaj corespunzător este afişat. Logica este asemănătoare cu cea din SplashScreen.java - verificarea pentru GPS se face în onResume(), iar cea pentru Internet în onReceive(), care este apelată şi din onResume() când se declară receptorul, nu doar la schimbarea reţelei. De menţionat că în onReceive() afişăm şi mesajul de GPS, pentru cazul în care utilizatorul opreşte Internetul şi GPS-ul, apoi porneşte doar Internetul.

Slider Menu

Din Main Activity se încarcă şi meniul care glisează în ecran. Codul aferent este pus în onCreate() şi semnalizat ca atare. Celelalte metode folosite de meniu şi semnalizate se află spre finalul clasei. Comportamentul la click este gestionat în SlideMenuClickListener.

**2.1.3 SettingsActivity**

Clasa aferentă ecranului de setări. De aici se pot face trei lucruri de către utilizator: stabilirea razei pe care se detectează magazine de bicicletă, a unităţii de măsură şi dacă să primească notificări de la noi. În onCreate() este apelată bindPreferenceSummaryToValue pentru fiecare opţiune; aceasta modifică interfaţa în funcţie de alegerea făcută de utilizator. Tot aici se apelează loadPreferenceScreen(mFirstLoad). mFirstLoad este o booleană care indică dacă este prima încărcare a ecranului. În funcţie de aceasta, se încarcă ecranul corespunzător - pref\_metric\_general sau pref\_imperial\_general - în loadPreferenceScreen.

În ecranul de setări se poate modifica unitatea de măsură, iar în funcţie de aceasta se modifică şi unitatea pentru viteză şi rază - în mile sau km. Acest comportament este controlat în două funcţii - loadPreferenceScreen şi onPreferenceChange.

loadPreferenceScreen, pe lângă a fi apelată din onCreate cu mFirstLoad = true, este apelată şi din onPreferenceChange(). Această funcţie este apelată de sistem la fiecare modificare de preferinţă. Noi apelăm de aici loadPreferenceScreen cu mFirstLoad = false doar dacă se modifică unitatea de măsură. În onPreferenceChange, pe ramura de else din if (prefIndex >= 0), controlăm comportamentul când utilizatorul modifică unitatea de măsură, iese din ecran, apoi revine. Decidem ce ecran să modificăm în funcţie de mIsMetricLoaded - dacă ecranul cu unităţi metrice este încărcat. Acestuia îi este asignată o valoare în loadPreferenceScreen(), în funcţie de ecranul încărcat/modificat.

**2.1.4 MapsActivity**

The map used when displaying all of the shops or only one. The origin of the call to the map is identified in setUpMap(), and the map is populated accordingly. If there is only one shop, the data regarding it is transferred through a bundle. Otherwise, we extract the data from the database via the ContentProvider.

**2.1.5 ShopsFragment**

Fragmentul ataşat activităţii principale (MainActivity). În onCreateView() se populează lista. Adaptorul este populat din printr-un SimpleCursorAdapter, care interoghează baza de date printr-un URI (vezi în directorul "Tutoriale" arhitectura acestuia). updateShopList() actualizează lista de magazine printr-o sincronizare instantă (SyncAdapter.syncImmediately(getActivity());).

Momentan, avem un buton de refresh care apelează updateShopList(), declanşând sincronizarea. În viitor, vom elimina acest buton.

**2.8 Places. FetchGooglePlaces**

Clasa este folosită pentru a accesa API-ul Google. Momentan, acest lucru se face printr-un AsyncTask, însă vom migra la un serviciu pe parcurs. Apelul efectiv se face în doInBackground; getPlaceDataFromJson() este folosită pentru a prelucra datele în format JSON întoarse de Google, iar rezultatul este publicat în onPostExecute. Aici sunt scrise în adaptor (mShopsAdapter), care este folosit în ShopsFragment pentru a popula lista.

**2.2 Pachetul sync**

Pachetul de aducere a datelor de pe server în aplicaţie. Am renunţat la vechea implementare de AsyncTask în favoarea acesteia. onPerformSync este apelat din syncImmediately, care la rândul lui este apelat de fiecare dată când nevoie de reîmprospătarea listei de magazine. În onPerformSync se face apelul de Google API şi prelucrarea datelor în format JSON. Pentru a funcţiona, adaptorul are nevoie de un serviciu de autentificare şi de un "autentificator" - de faţadă în cadrul nostru, nu sunt folosite -, şi de un serviciu de sincronizare.

**2.2.1 SyncAdapter**

Este singura clasă din acest pachet care procesează ceva. Din ea se poate face sincronizarea imediat prin apelarea metodei syncImmediately() care, printr-un ContentResolver.requestSync, apelează onPerformSync. În această metodă se face, în această ordine, apelul HTTP, procesarea răspunsului în JSON şi inserarea în baza de date.

**2.3 Pachetul data**

Conține codul pentru baza de date. Aceasta este de tip SQLite. Numele coloanelor și URI-urile sunt stocate în ShopsContract, iar în ShopsDbHelper comportamentul la crearea bazei de date (comanda CREATE TABLE) și la actualizare (DROP TABLE IF EXISTS + table\_name). ShopsProvider conţine metodele apelate de ContentProvider - query, insert, update etc.

**2.4 Pachetul Utility**

**2.4.1 DeviceConnection**

Conţine metode ce verifică accesul la reţea ori locaţie. Numele metodelor sunt intuitive şi oferă o idee legat de ce fac.

**2.4.2 Constants**

Toate constantele folosite în aplicaţie. Nu include mesajele de eroare/informare, care vor fi/sunt trecute în values/strings.xml

**2.4.3 Utility**

Clasă ce conţine diverse funcţii ajutătoare - de formatare, de extragere a preferinţelor utilizatorului etc.

**2.4.4 GlobalState**

Clasă folosită pentru a stoca variabilele globale.

**2.5 Pachetul form**

Conţine clasele aferente operaţiunii de a adăuga un nou magazin. Din SliderMenu se apasă pe "Add a shop", iar de acolo se deschide harta cu locaţia utilizatorului. După ce utilizatorul stabileşte locaţia, purcede către ecranul cu formularul.

**2.5.1 AddShopMap**

Harta pe care se adaugă magazinul. Pentru a menţine codul aerisit, am ales să avem o nouă hartă pe lângă cea folosită pentru un singur magazin deja existent/toate magazinele. La iniţializare, harta conţine locaţie utilizatorului. Acesta poate indica o altă locaţie pentru magazin folosind căutarea implementată în PlacesAutoComplete/PlacesAutoCompleteAdapter. Momentan, la apăsarea lungă pe hartă se adaugă un nou balon şi se şterge cel vechi. Comportamentul va fi modificat pe parcurs.

**2.5.2 PlacesAutoComplete**

Clasă ajutătoare pentru implementarea căutării de locuri.

**2.5.3 PlacesAutoCompleteAdapter**

Folosită pentru a implementa căutarea de locuri pe hartă. În getFilter() se declanşează căutarea prin apelul HTTP către API-ul Google, însă doar dacă utilizatorul a introdus minim trei litere. În publishResults() este notificat către sistem rezultatul apelului.

**2.5.4 AddShopFormActivity**

Activitatea ce conţine formularul. În onCreate() este încărcat designul (activity\_add\_shop\_form), iar căsuţelelor (nume magazin, număr de telefon etc.) le sunt desemnate valori. Avem şi trei booleane de control - mShopNameOk, mShopWebsiteOk şi mShopPhoneNumberOk - folosite pentru a aplica constrângerile pe câmpuri. În onCreate() este definit şi comportamentul pentru cele trei câmpuri prin apelul metodei native addTextChangedListener, iar în aceasta - afterTextChanged. De aici se apelează metodele care fac efectiv verificarea: checkShopName(), checkShopWebsite() şi checkShopPhoneNumber().

În metoda addShop() se face adăugarea efectivă. Metoda este apelată prin apăsarea butonului "Add Shop" de către utilizator. Parametrii pentru apel sunt formaţi aici, iar apelul efectiv se face în clasa PostForm. Adăugarea se face doar dacă cele trei booleane de control (mShopNameOk etc.) sunt true. Altfel, un mesaj de avertizare este afişat, iar câmpurile necorespunzătoare sunt evidenţiate.

**Notă**: Doar numele este obligatoriu. Dacă, însă, utilizatorul completează site-ul ori numărul de telefon, verificăm să fie conforme.

**2.5.5 PostForm**

Aici se face apelul efectiv către API-ul Google în metoda doInBackground(). Clasa extinde AsyncTask. Datele sunt puse în format JSON în createJSONObject().

**2.6 Pachetul Places**

**2.6.1 FetchPlacesAutocomplete**

Aici se face apelul către Google Places API, funcţionalitatea AutoComplete. Metoda este *autocomplete*, care se apelează din PlacesAutoCompleteAdapter. În această metodă se face şi prelucrarea răspunsului din JSON.

**2.6.2 FetchGooglePlaces**

Vechea clasă folosită pentru a lua magazinele de biclă de pe serverele Google. Aceeași logică este implementată și în SyncAdapter, diferența fiind că parsarea JSON se face într-o metodă separată - getPlaceDataFromJson. Vom păstra clasa pe moment.

**2.7 Pachetul slidermenu**

Conține clasele aferente meniului glisant. Vom trata SliderDrawerListAdapter, clasa care procesează - restul sunt clase ajutătoare.

**2.7.1 SliderDrawerListAdapter**

În getView() sunt setate imaginile fiecărei opțiuni din meniu. De asemenea, aici este încărcată și lista de opțiuni (drawer\_list\_item).

**2.8 Pachetul test**

Conține două clase folosite în testarea bazei de date. ShopTest folosește direct baza de date, iar ShopsTestContentProvider folosește ContentProvider-ul ca intermediar. Metodele din ambele clase au nume intuitive, care indică funcționalitatea lor.

**Notă**: În Android Studio, toate metodele folosite în testare trebuie să înceapă cu "test". Altfel, programul le ignoră.