Złożenie pracy online:
2011-05-29 18:37:52
Kod pracy:
5267
Kod załącznika:
5259

Magdalena Wiercioch
(nr albumu: 17070*INF/LIC)

Praca inżynierska

Aplikacja dla systemu Android z archiwizacją danych na komputerze z systemem Windows.

Application for Android System with Data Archiving on PC with Microsoft Windows.

Wydział: Informatyki

Kierunek: Informatyka

Specjalność: inżynieria oprogramowania

Promotor: dr Henryk Telega

Nowy Sącz, 2011

SPIS TREŚCI

1. Wstęp.

Pojawienie się na rynku nowego systemu operacyjnego dla urządzeń mobilnych w postaci platformy Android spowodowało wzrost zainteresowania tego typu technologiami, zwłaszcza wśród programistów. Od tego czasu powstało mnóstwo aplikacji wzorowanych na tych, dotychczas dostępnych w komputerach stacjonarnych.
Program opracowany w ramach pracy inżynierskiej jest szeroko rozumianym notatnikiem uwzględniającym mobilność jego przyszłego użytkownika.

1.1. Cel i zakres pracy.

Celem pracy było zaprojektowanie i utworzenie aplikacji dedykowanej systemowi Android polegającej na generowaniu notatek o różnej formie skojarzonych z mapą.
Dodatkowo wprowadzono moduł archiwizacji danych zapisanych na urządzeniu mobilnym poprzez ich synchronizację z komputerem z systemem Windows. Jako rozszerzenie zbudowano aplikację dla platformy Windows, która gromadzi przesłane dane i zajmuje się ich przetwarzaniem.

Praca swym zakresem obejmowała:

- zgromadzenie literatury tematu,
- zapoznanie się ze sposobem działania i charakterystyką systemu Android,
- rozpoznanie możliwości tworzenia połączeń bluetooth na platformie .NET,
- opracowanie założeń funkcjonalnych,
- zaprojektowanie struktury logicznej i interfejsu aplikacji,
- testy, wprowadzanie poprawek,
- opracowanie dokumentu w formie pisemnej.

Utworzona w ramach pracy dyplomowej aplikacja jest przeznaczona dla urządzeń mobilnych wyposażonych w system Android w wersji 2.2 oraz wyższej.

1.2. Struktura.

Struktura pracy pisemnej jest następująca.
W rozdziale 2 przedstawiono informacje dotyczące systemu Android, jego architekturę oraz sposób działania aplikacji mu dedykowanych. Zamieszczono tam również wiadomości na temat bazy danych SQLite.
Rozdział 3 jest poświęcony opisowi tworzenia aplikacji MapNote, stanowiącej część praktyczną pracy. Ujęto w nim wszystkie etapy konstrukcji programu począwszy od wyodrębnienia założeń, doboru narzędzi, a skończywszy na sprawozdaniu z kolejnych etapów implementacji.
Rozdział 4 stanowi opis realizacji aplikacji MapNote Synch umożliwiającej archiwizację

danych pochodzących z programu MapNote. Zamieszczono w nim informacje na temat przebiegu tworzenia aplikacji wraz z wiadomościami dotyczącymi użytych dodatkowych bibliotek dla platformy .NET.
Rozdział 5 zawiera podsumowanie pracy.

2. Wprowadzenie do systemu Android.

Android jest systemem operacyjnym dla urządzeń mobilnych. Platformę tą określa się jako wspólny projekt firmy Google oraz organizacji OHA (Open Handset Alliance) powstałej w 2007 roku i zrzeszającej największych światowych udziałowców rynku takich jak: Samsung, Motorola, Intel, Qualcomm itd.
Porównując Androida z bogatą ofertą konkurencji można dostrzec cechy, które stawiają go jako nową propozycję wśród dotychczasowych systemów. Oto one:

- architektura oparta na komponentach
  Taka struktura umożliwia wykorzystywanie elementów należących do jednej aplikacji w innych. Dzięki temu można dowolnie rozwijać i poszerzać funkcjonalności już istniejących komponentów.
- automatyczne zarządzanie cyklem życia aplikacji
  Zapewnia to przede wszystkim stabilność systemu. Aplikacje działają zupełnie niezależnie od siebie i są oddzielone wieloma poziomami zabezpieczeń.
  W konsekwencji użytkownik jest zupełnie nieświadomy kiedy na przykład dana aplikacja przestaje być aktywną.
- optymalizacja pod względem oszczędzania energii i zasobów pamięciowych.

Jednocześnie należy zauważyć, że Android jest zupełnie bezpłatną platformą (open source). Licencja na framework opiera się na Apache Software License (ASL/Apache2). Ponadto dla developerów udostępnionych jest szereg narzędzi oraz obszerna dokumentacja.

2.1. Konfiguracja środowiska.

Aplikacje dedykowane Androidowi mogą być tworzone w następujących systemach operacyjnych:

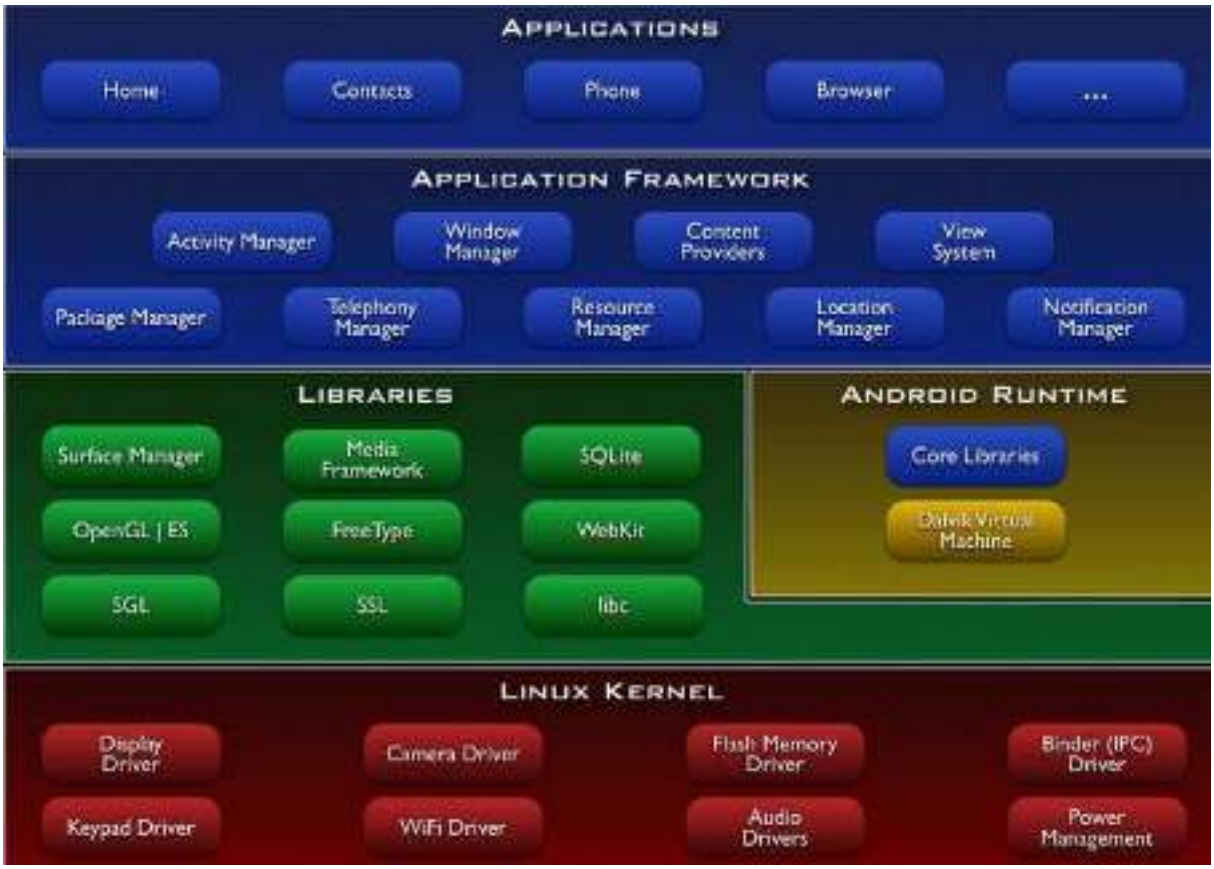
- Windows XP/Vista/7,
- Linux,
- Mac OS X 10.5.8 i późniejsze.

W celu rozpoczęcia pracy z Androidem konieczne jest skonfigurowanie odpowiedniego narzędzia. Przede wszystkim z uwagi na fakt, że aplikacje dla Androida tworzy się w języku Java, należy zainstalować bieżącą wersję JDK (*Java Development Kit*). Następnie potrzebne jest środowisko wykonawcze dla tego języka. Najbardziej rozpowszechniony i określany jako godny uwagi jest Eclipse IDE. Co więcej, jest on bezpłatny i polecany nawet przez twórców Androida. Kolejny etap stanowi instalacja Android SDK (*Android Software Development Kit*). Zawiera on pakiet klas tworzących Framework oraz przydatne programiście narzędzia i dokumentację. Ostatnim elementem fazy przygotowawczej jest konfiguracja zainstalowanego środowiska wykonawczego.
Jeżeli jest to Eclipse, bardzo przydatna jest wydana przez Google wtyczka Android

Development Toolkit (*ADT*). Pozwala ona na integrację środowiska z systemem.

2.2. Architektura.

Android zbudowany jest w oparciu o architekturę warstwową. Każda z niżej wymienionych warstw stanowi kluczowe znaczenie dla całego systemu.



Rys. 2.1. Warstwowa architektura systemu Android.

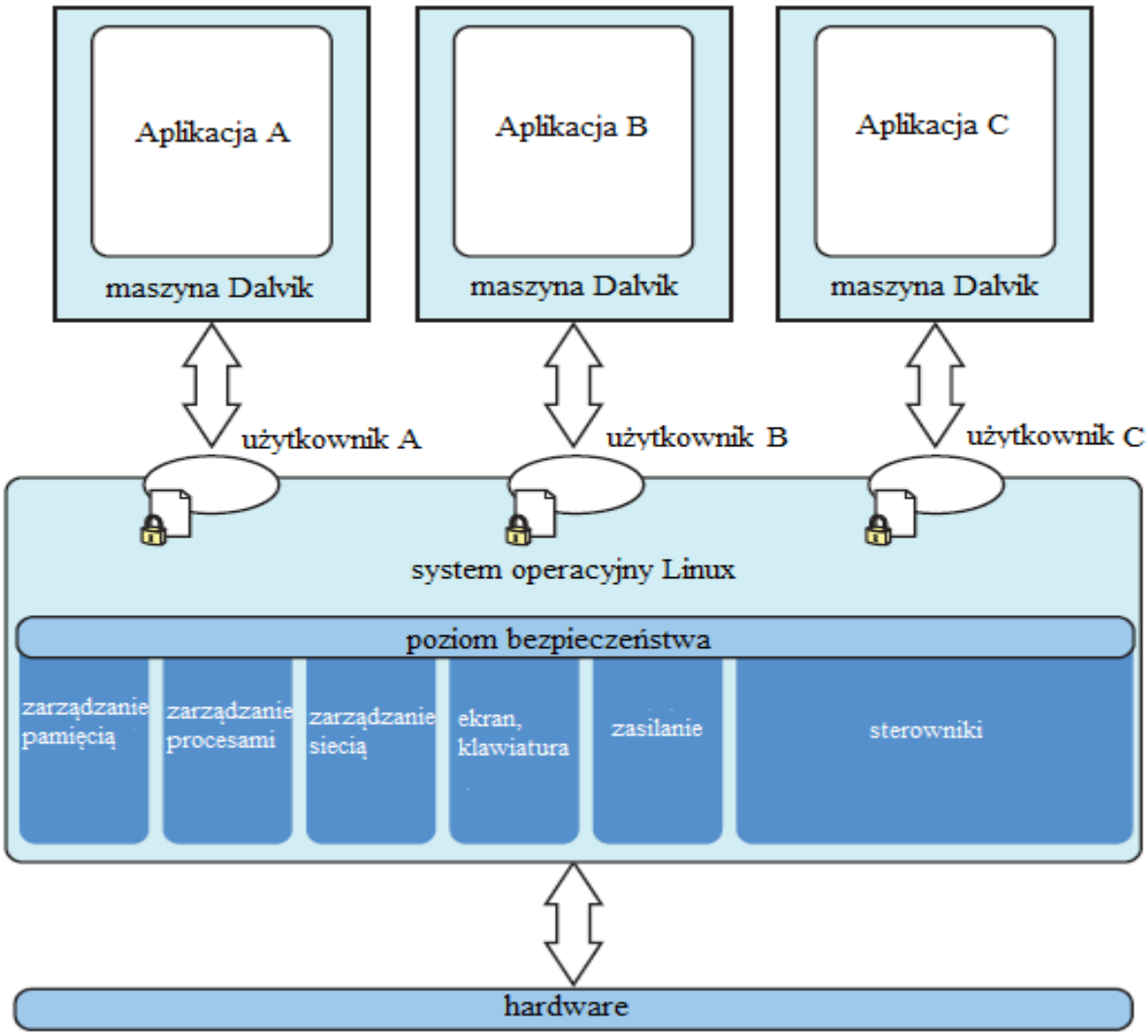
- *Linux Kernel* – jądro Linuksa
  Na tym poziomie zapewnione jest zarządzanie pamięcią, siecią, procesami, bezpieczeństwem, obsługą ekranu, klawiatury, wbudowanymi do danego urządzenia sensorami oraz sterownikami aparatu i bluetooth. Należy jednak podkreślić, że docelowy użytkownik aplikacji nie ma bezpośrednio kontaktu z technicznym aspektem Linuksa.
- *Libraries*
  Kolejna warstwa to biblioteki. Są one napisane w języku C/C++, a kompilowane z uwzględnieniem architektury urządzenia. Następnie są one przeinstalowywane przez dystrybutora urządzenia. Odpowiadają za takie funkcje jak obsługa bazy danych lub wyświetlanie grafiki. Warto zauważyć tutaj, że Android dostarcza

również bibliotekę dla grafiki 3D zwaną OpenGL ES (*Open GL for Embedded Systems*).
Wszystkie biblioteki są wywoływane przez aplikacje wyższego poziomu. Dodatkowo istnieje możliwość samodzielnego ich tworzenia poprzez przeznaczone do tego narzędzie – Native Development Toolkit (*NDK*).

- *Android Runtime*
  Warstwa zawiera biblioteki Java oraz wirtualną maszynę Dalvik. Biblioteki te, mimo dostrzegalnych podobieństw do tych standardowych dla języka Java, zawierają istotne różnice. Mechanizm wygląda w ten sposób, że kody źródłowe są najpierw standardowo kompilowane do kodu bajtowego, a następnie wykonywane w lżejszej formie jako pliki *.dex przez wspomnianą maszynę Dalvik. W rzeczywistości Dalvik jako produkt Google jest wersją maszyny wirtualnej Java zaprojektowanej dla celów oszczędności pamięci.
- *Application Framework*
  Znajdują się tutaj bloki wspomagające budowę aplikacji i rozszerzające jej funkcjonalność. Przykładem jest Notification Manager służący do tworzenia alertów bądź Location Manager dający możliwość ustalenia lokalizacji. Pozostałe istotne bloke, tzn. Activity Manager, Content Provider i Resource Manager zostaną omówione w dalszej części rozdziału.
- *Applications and Widgets*
  Jest ostatnią warstwą w strukturze. W praktyce to z nią bezpośrednio wchodzi w interakcję użytkownik.

2.2.1. Bezpieczeństwo.

Kwestia bezpieczeństwa oraz integralność stanowi ważny aspekt w systemie Android. Jednym z mechanizmów, które mają to zapewnić jest tworzenie odrębnego profilu użytkownika dla każdej zainstalowanej aplikacji. Wiąże się to z nadaniem identyfikatora i skojarzeniem przestrzeni dla osobistych plików.
Ponadto stosuje się dwa sposoby przyznawania praw dostępu do określonych zasobów. Należy do nich np. możliwość wykonywania połączeń telefonicznych, dostępu do sieci, korzystania z aparatu. W celu użycia tego typu funkcji aplikacja musi o tym poinformować system. Realizuje się to poprzez odpowiedni zapis w tzw. pliku manifestu. Równocześnie aplikacja może także sama definiować zakres w jakim oferuje swoje usługi dla innych programów. Istnieje również możliwość nadawania uprawnień dostępu ad-hoc poprzez użycie specjalnych identyfikatorów – Uniform Resource Identifiers (*URI*), odnoszących się np. do plików graficznych.



Rys. 2.2. Architektura systemu Android z punktu widzenia bezpieczeństwa.

2.3. Cykl życia aplikacji.

Pracując z Androidem należy mieć świadomość, że filozofia działania dedykowanych mu aplikacji odbiega od standardów. Przede wszystkim istotną rolę odgrywa w tej kwestii wspomniany w podrozdziale 2.2 manager aktywności (*Activity Manager*). Zarządza on wszystkimi uruchomionymi programami umieszczając je na stosie wywołań. Dzięki temu, z punktu widzenia użytkownika wywoływanie aplikacji przypomina pracę z przeglądarką internetową.

Specyficzny tryb działania programów dla Androida wiąże się z istnieniem czterech bloków, które je tworzą. Poniższe podrozdziały mają na celu wyjaśnienie ich własności.

2.3.1. Aktywności.

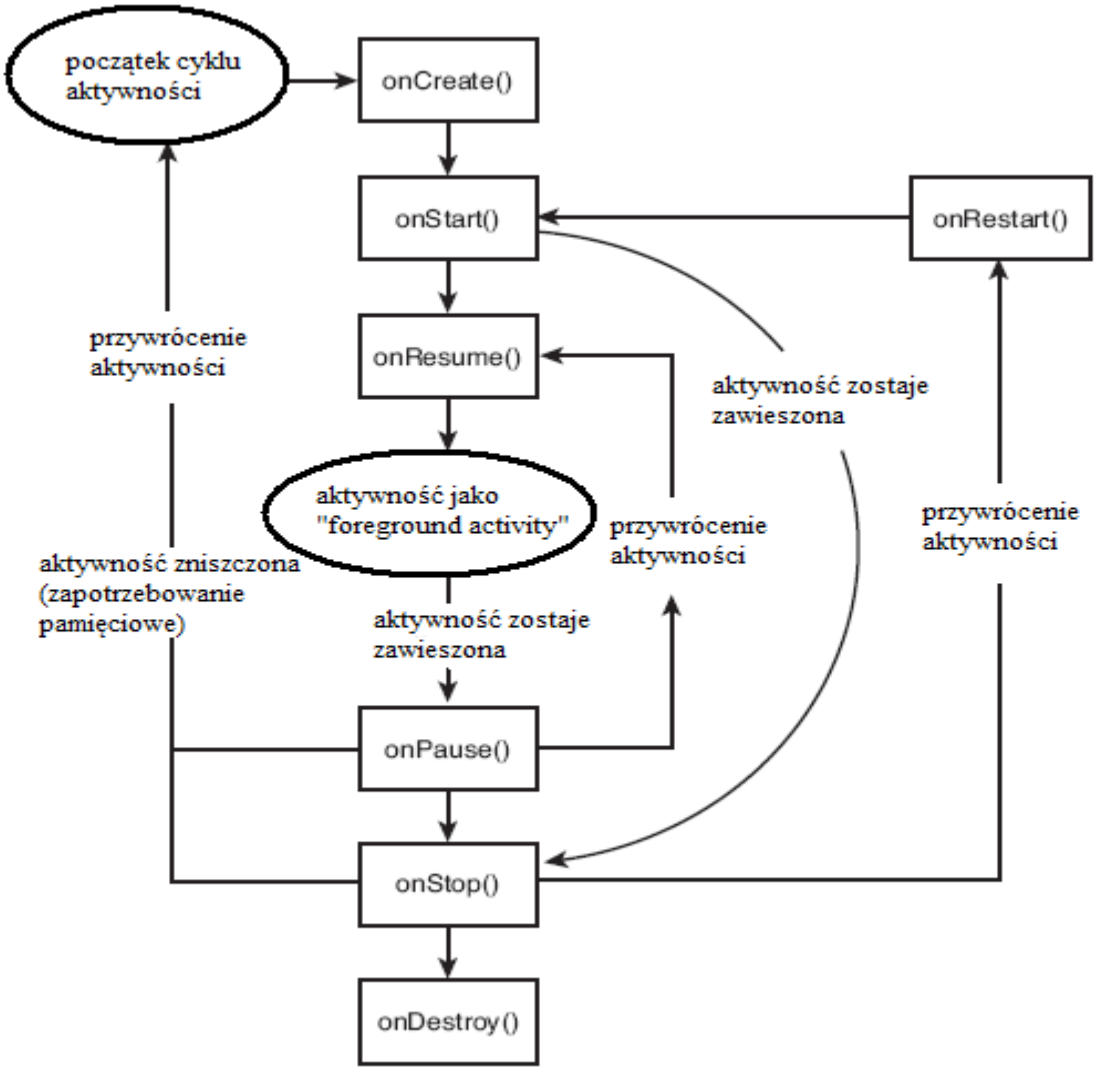
Aktywność (*Activity*) reprezentuje pojedynczy ekran aplikacji. Jest to jedno z fundamentalnych pojęć dla systemu. W rzeczywistości aplikację stanowi co najmniej jedna aktywność wraz z procesem Linuksa pełniącym rolę kontenera dla tych aktywności. Ich istota

jest złożona. Przede wszystkim aktywność charakteryzuje się posiadaniem własnego cyklu życia. Dlatego też sama aplikacja w systemie może nadal być aktywną mimo usunięcia procesu, który jest z nią związany. Aktywność działa od niego niezależnie, według własnych reguł. Związane jest z tym istnienie stanów, w których aktywność może się znajdować.

- Resumed
  Oznacza to, że aktywność zajmuje w bieżącym momencie ekran użytkownika. Określa się ją jako *foreground activity*.
- Paused
  Użytkownik widzi ekran interfejsu dostarczonego przez inną aktywność będącą w stanie Resumed, a ta jest dla niego niewidoczna bądź zajmuje niewielki fragment ekranu. Należy dodać, że poza tym faktem działa zupełnie tak samo jakby była bieżąca i jej obiekt znajduje się na stosie wywołań zaraz pod obiektem tej, która jest aktualnie w stanie Resumed. Jeśli aktywność bieżąca zostanie zniszczona, wówczas ta stanie się główną. Niemniej jednak może zostać też usunięta w sytuacji zapotrzebowania zasobów pamięciowych.
- Stopped
  Aktywność wciąż działa i jest na stosie wywołań, ale nie ma już uchwytu do okna. W razie zapotrzebowań pamięciowych zostanie natychmiast usunięta.

Istotnym faktem jest to, że osoba pisząca aplikację dla Androida nie wpływu na stan, w którym aktywność powinna się znajdować, ponieważ to kontroluje system. Jednak klasa Activity udostępnia metody, które można nadpisywać i są one wywoływane w chwili przejścia pomiędzy stanami. Należą do nich poniżej wymienione.

- OnCreate() – wywoływana jest przy rozpoczęciu aktywności.
- OnStart() – wskazuje, że użytkownikowi zostanie pokazany ekran.
- OnResume() – aktywność stanie się bieżącą i rozpocznie interakcję z użytkownikiem.
- OnPause() – oznacza, że aktywność przechodzi do stanu zawieszenia.
- OnStop() – aktywność przechodzi do stanu zatrzymania.
- OnDestroy() – wskazuje, że aktywność zostanie usunięta.
- OnRestart() - zatrzymana aktywność powraca do stanu bieżącego.
- OnSaveInstanceState() – wskazuje na przejście aktywności ze stanu Resumed do Paused lub ze stanu Paused do Stopped.
- OnRestoreInstanceState() – aktywność powraca do stanu bieżącego po okresie wstrzymania i przywracany jest jej zapamiętany stan sprzed wstrzymania.



Rys. 2.3. Cykl życia aktywności.

2.3.2. Intencje.

Intencje są asynchronicznymi wiadomościami wykorzystywanymi do uruchamiania pewnych zadań, wywoływania aktywności lub do wysyłania informacji rozgłoszeniowych. System używa intencji np. do powiadamiania o zmianach na poziomie hardware lub nowych danych, takich jak sms bądź e-mail. Ponadto można tworzyć własne intencje. Z punktu widzenia programisty są one obiektami klasy Intent. W ten sposób przekazuje się informacje, które wykorzystuje system lub odbiorca intencji. Zaliczają się do nich: nazwa klasy odbiorcy, jego typ, rodzaj akcji do wykonania, specjalny identyfikator wskazujący na dane, będące przedmiotem pracy, dodatkowe informacje dla odbiorcy oraz sposób, w jaki system powinien traktować wywoływaną aktywność.

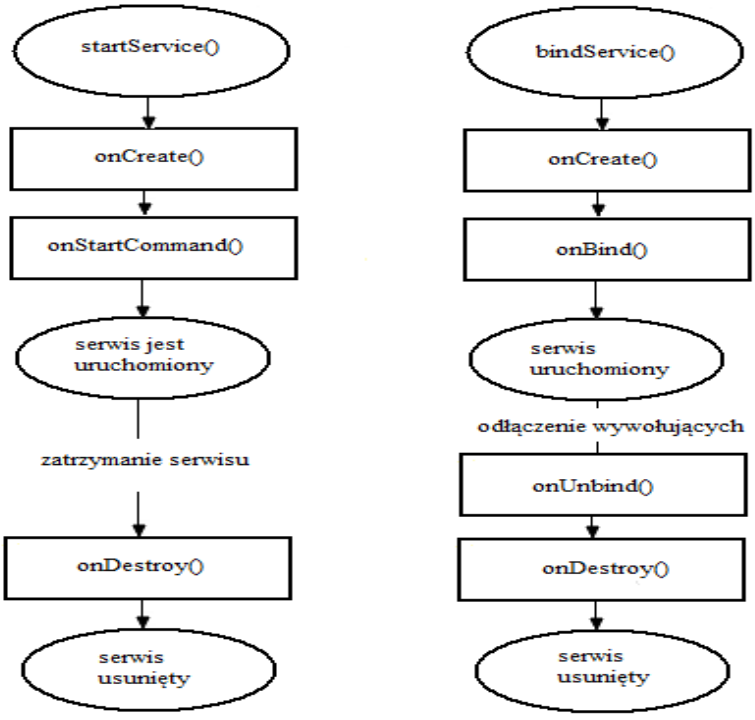
Należy dodać, że istnieją dwa rodzaje intencji: *explicit* i *implicit*. W praktyce różnica między nimi związana jest z typem ich odbiorcy. Tak więc, intencje określane jako implicit są przeznaczone zwykle do wywoływania aktywności należących do innych aplikacji.

2.3.3. Serwisy.

Serwis (*Service*) jest strukturą pozwalającą wykonywać operacje w tle bez dostarczania interfejsu dla użytkownika. Znajduje zastosowanie przy procesach wymagających dłuższego czasu trwania. Zazwyczaj jest więc przydatny np. w aplikacjach ściągających pliki o większym rozmiarze lub dostarczających pewnych danych, jak informacje o lokalizacji bądź wiadomości z kanałów RSS. Istnieją dwa rodzaje serwisów.

- Started
  Wywoływany jest przez inną strukturę np. aktywność w celu wykonania określonego zadania. Działa niezależnie od bloku, przez który go utworzył. Po zakończeniu czynności zostaje zniszczony.
- Bound
  Charakteryzuje się udostępnianiem komunikacji w postaci interakcji klient – serwer ze strukturą, która go przywołała.

Typowo serwis działa w głównym wątku procesu, ale można to zmienić, a nawet należy dla złożonych operacji, poprzez utworzenie dla niego osobnego wątku. Ponadto serwisy, podobnie jak aktywności posiadają własny cykl życia, którego forma uzależniona jest od rodzaju serwisu.



Rys. 2.4. Cykl życia odpowiednio serwisów Started oraz Bound.

W zależności od typu serwisu, w celu jego rozpoczęcia wywoływana jest metoda startService() lub bindService(). Ta ostatnia powoduje dodatkowo nawiązanie połączenia

z serwisem. Serwis określany jako Started działa dopóki nie zostanie wywołana na jego rzecz metoda stopService() lub wcześniej sam zakończy swoje działanie. Natomiast w przypadku serwisu Bound jest on aktywny do czasu, gdy co najmniej jeden klient jest z nim połączony. Niezależnie od tego sam system może wymusić zakończenie serwisu dla odzyskiwania zasobów pamięciowych. Wówczas najmniej prawdopodobnym do zatrzymania serwisem jest ten połączony z bieżącą aktywnością.

2.3.4. Dostawcy danych.

Dostawca danych (*Content provider*) stanowi strukturę pozwalającą na dostęp do danych należących do innych aplikacji znajdujących się w systemie. Jest to istotny element systemu, ponieważ jak zostało to przedstawione w podrozdziale 2.2.1. dostęp do danych każdego programu jest chroniony m.in. poprzez przyznawanie odpowiedniej przestrzeni dla plików. Należy przy tym dodać, że Android udostępnia szereg danych przez wbudowane w system aplikacje i można ich używać właśnie poprzez element content provider.

- MediaStore – dostęp do plików multimedialnych.
- Browser – dostęp do historii przeglądanych stron internetowych i zakładek.
- CallLog – uzyskanie informacji dotyczących połączeń.
- ContactsContract – dostęp do książki adresowej.
- Settings – uzyskanie informacji o bieżących ustawieniach i konfiguracjach.
- UserDictionary – dostęp do słownika.

Content provider udostępnia dane w postaci tabel, w których każdy rekord jest jednoznacznie zidentyfikowany poprzez pole _id. Natomiast odniesienie do tabel stanowią identyfikatory URI zbudowane według następującego, ściśle określonego schematu.

`content://authority/path/id`

gdzie:

content – standardowy prefiks, oznacza że dane są skojarzone z content provider-em,
authority – nazwa dostawcy, zazwyczaj jest to nazwa pakietu,
path – wirtualny katalog, który identyfikuje typ oczekiwanych danych,
id – identyfikator rekordu w tabeli.

Wydobycie danych wiąże się z podaniem wspomnianego URI, nazwy pola oraz typu danych, które ono zawiera. Ponadto jeśli aplikacja ma udostępniać swoje dane, musi posiadać zaimplementowany interfejs ContentProvider oraz zapis w pliku ustawieniowym, zwanym manifestem.

Istotną kwestią jest fakt, że standardowo dane w systemie Android są przechowywane w bazie danych SQLite. Szczegółowe omówienie tej bazy znajduje się w podrozdziale 2.5.

2.4. Struktura aplikacji.

Każdy projekt aplikacji dedykowanej dla Androida zawiera określoną strukturę plików i katalogów. Najważniejsze spośród nich to poniższe.

- *src* – katalog zawiera kody źródłowe klas,
- *res* – umieszczone są w nim zasoby danego programu,
- plik *AndroidManifest.xml* – zawiera szczegółowe informacje dotyczące ustawień aplikacji.

2.4.1. Zarządzanie zasobami.

Zasobami są wszystkie dane używane przez aplikację takie jak: ikony, ciągi tekstowe, pliki multimedialne, graficzne. Większość z nich przechowana jest w formacie XML. Wszystkie zasoby znajdują się w katalogu *res*. W zależności od typu, umieszczone są w odpowiednich jego podkatalogach. Przykładowo, pliki graficzne znajdują się w katalogu */res/drawable/*, tworzące menu w */res/menu/*, a opisujące wygląd danego ekranu użytkownika w */res/drawable/*. Do zasobów można odnosić się w czasie wykonania aplikacji. Jest to możliwe, ponieważ należy zauważyć, że każdy budowany program wyposażony zostaje automatycznie w klasę o nazwie *R*, umieszczonej w pliku *R.java*. Składa się ona z identyfikatorów odnoszących się do zasobów. Po każdorazowym dodaniu nowego zasobu, kompilator zasobów kompresuje je i uzupełnia zawartość klasy o odpowiedni kod. Dla przykładu zakładając, że w pliku *string.xml* opisującym wyświetlane teksty znajduje się definicja etykiety o nazwie *powitanie*, programista może się do niej odnieść poprzez instrukcję *R.string.powitanie*, otrzymując w efekcie jej identyfikator. W celu otrzymania tego napisu należy wywołać:

```
getResouces().getString(R.string.powitanie)
```

gdzie:

getResources() – metoda zwraca obiekt klasy Resources, odnoszącej się do zdefiniowanych zasobów.

2.4.2. Konfiguracja.

Do definiowania ustawień każdej aplikacji system Android udostępnia plik o nazwie *AndroidManifest.xml*. Zawiera on informacje o wersji programu, nadanych prawach i wszelkich regułach niezbędnych do jego działania. Ponadto muszą w nim zostać zarejestrowane wszystkie aktywności użyte w aplikacji, gdyż w przeciwnym razie dochodzi do błędu wykonania. Jest to również miejsce do definiowania odpowiednich uprawnień do korzystania z dostępnych zasobów i funkcji, np. odbierania wiadomości sms. Poniżej natomiast znajduje się przykład zezwolenia na wykonywanie połączeń telefonicznych.

```
<uses-permission android:name="android.permission.CALL_PHONE" />
```

2.5. Baza danych SQLite.

Do przechowywania większej ilości danych Android używa relacyjnej bazy SQLite. Jest dostępna na licencji public domain. W rzeczywistości zajmuje ona jeden plik. Na dysku utrzymywana jest w postaci B-drzew, a każde z nich reprezentuje inną tabelę. Ponadto oparta jest na silniku SQL i nie wymaga uruchamiania osobnego procesu RDBMS. Baza w dużej mierze nawiązuje do standardu SQL-92. Różnice występują np. w określaniu typów danych. Istnieje m.in. osobliwość, którą dokumentacja nazywa *manifest typing*. Polega na tym, że mimo podania typu danych dla kolumny, przy tworzeniu tabel dopuszcza się umieszczanie w niej wartości innego typu.
W kwestiach bezpieczeństwa należy polegać na zabezpieczeniach zapewnianych przez określony system plików.
W systemie Android plik bazodanowy mieści się w katalogu */data/data/<nazwa pakietu>/databases/*. Bezpośredni dostęp do bazy realizowany jest poprzez narzędzie linii komend lub programistycznie, tworząc odpowiednie klasy.

W dalszej części tego podrozdziału omówiona zostanie obsługa bazy danych SQLite w systemie Android z punktu widzenia programisty.

W celu utworzenia bazy należy powołać do istnienia obiekt klasy *SQLiteDatabase*. Następnie, aby uzyskać tabelę trzeba wywołać na rzecz stworzonej wcześniej instancji metodę *execSQL()* w postaci na przykład zaprezentowanej poniżej.

```
execSQL("CREATE TABLE Ksiazki "+
"(ID INTEGER PRIMARY KEY AUTOINCREMENT, "+
   "Tytul TEXT, rok_wydania INTEGER)")
```

Tabelę można usunąć poprzez instrukcję *drop table*.

```
execSQL("DROP TABLE Ksiazki")
```

Modyfikacja utworzonej tabeli może przebiegać na dwa sposoby. Pierwszym z nich jest kolejne wywołanie metody execSQL() z odpowiednią instrukcją.

```
execSQL("INSERT INTO Ksiazki (Tytul, rok_wydania)"+
            "VALUES ('Potop', 2000)")
```

Drugą możliwość stanowią wywołania metod *insert()*, *delete()* oraz *update()*. W tym celu należy utworzyć obiekt klasy *ContentValues*. Jest to mapa, w której kluczowi odpowiadającemu kolumnie przyporządkowuje się daną wartość.

```
ContentValues contentValues = New ContentValues();
contentValues.put(“Tytul”,”Potop”);
contentValues.put(“rok_wydania”,2000);
```

Następnie na rzecz obiektu klasy SQLiteDatabase trzeba wywołać odpowiednią z metod: *insert()*, *update()* bądź *delete()*.
Kolejną kwestią jest tworzenie zapytań do bazy danych. Realizuje się to poprzez metodę *rawQuery()* lub *query()*. W pierwszym przypadku może to wyglądać w następujący sposób.

```
String[] tab={"Potop"};
Cursor cursor =
sqliteDatabase.rawQuery("SELECT ID,rok_wydania FROM Ksiazki "+
"WHERE Tytul=?",tab);
```

gdzie:

tab – tablica parametrów zapytania,
sqliteDatabase – obiekt klasy SQLiteDatabase.

Natomiast posługując się metodą query() wywołanie przybiera postać.

```
String[] kolumny={"ID", "rok_wydania"};
String[] tab={"Potop"};
Cursor cursor=sqliteDatabase.query("Ksiazki", columns, "name=?",
tab, null, null, null);
```

gdzie:

kolumny – tablica zawierająca pola wybierane w zapytaniu,
tab – tablica parametrów zapytania,
sqliteDatabase – obiekt klasy SQLiteDatabase.

W obu powyższych przypadkach zapytania zwracały obiekt klasy *Cursor*. W efekcie pozwala on przechodzić iteracyjnie przez otrzymane rekordy.

3. Aplikacja MapNote.

Aplikacja MapNote jest programem przeznaczonym na urządzenie mobilne z systemem Android.

3.1. Założenia.

Założenia funkcjonalne aplikacji:

- dodawanie notatek tekstowych,
- edycja notatek tekstowych,
- wysyłanie notatek tekstowych w formie wiadomości sms,
- wyświetlanie notatek tekstowych względem kryteriów sortowania,
- dodawanie notatek rysunkowych,
- wyświetlanie notatek rysunkowych względem kryteriów sortowania,
- dodawanie notatek w formie zdjęć,
- edycja notatek zdjęciowych,
- wyświetlanie notatek zdjęciowych względem kryteriów sortowania,
- rejestrowanie aktualnej lokalizacji,
- powiązanie notatek tekstowych, rysunkowych oraz zdjęciowych z położeniem,
- wysyłanie informacji o bieżącej lokalizacji w formie wiadomości sms,
- archiwizacja notatek tekstowych, rysunkowych, zdjęciowych oraz rejestrów pobytu,
- wyświetlanie notatek na mapie,
- wyszukiwanie notatek w oparciu o odległość od aktualnego położenia użytkownika.

3.2. Dobór narzędzi.

Do wykonania aplikacji użyto następujących narzędzi:

- środowisko programistyczne Eclipse – Eclipse for Java Developers wraz z zainstalowanym Android SDK oraz Eclipse ADT Plug-In,
- system zarządzania bazą danych SQLite.

Prace zostały wykonane w języku programowania Java.

3.3. Realizacja.

Proces realizacji aplikacji podzielony został na etapy. Każdy z nich wprowadzał istotną funkcjonalność do programu.

3.3.1. Obsługa bazy danych.

Aspektem, który odgrywał ogromną rolę w kwestii funkcjonowania aplikacji był projekt bazy danych. Z uwagi na uwarunkowania bazy SQLite omówione w podrozdziale 2.5, charakter danych oraz ich przeznaczenie, opracowano następującą jej strukturę.

Rys. 3.1. Diagram bazy danych.

Zgodnie z diagramem z rysunku 3.1:

- tabela Notes zawiera dane potrzebne dla notatek tekstowych,
- tabela Art przeznaczona jest dla notatek rysunkowych i zdjęciowych,
- tabela Visited_places zawiera dane miejsca tylko oznaczonego przez użytkownika.

Kolejnym istotnym zagadnieniem okazała się koncepcja sposobu dostępu do danych zapisanych w bazie biorąc pod uwagę częstą konieczność wykorzystania ich w kontrolkach. Ze względu na tak wszechstronne i wieloaspektowe ich użycie została w tym celu zdefiniowana klasa DataAdapter. Zawiera ona klasę wewnętrzną *DataAdapterHelper* dziedziczącą po klasie *SQLiteOpenHelper*. Zaimplementowano w niej metody *onCreate()* oraz *onUpgrade()* odpowiedzialne za utworzenie tabel dla aplikacji oraz za obsługę ewentualnych przyszłych zmian.
W następnej kolejności w klasie zostały zdefiniowane metody odpowiedzialne za wykonywanie zapytań do bazy. Ponadto utworzono też takie, które dodają, usuwają bądź uaktualniają określone rekordy. Dodatkowo klasę wzbogacono o metody implementujące umożliwienie dostępu do bazy danych do odczytu lub do zapisu. Odpowiadają za to metody: *openForReading()* oraz *open()*.

3.3.2. Tworzenie notatek.

Jedną z czynności było utworzenie modułów przeznaczonych do tworzenia notatek tekstowych, rysunkowych oraz zdjęciowych. Zważywszy na charakter aplikacji dedykowanych Androidowi wiązało się to w dużej mierze z utworzeniem odpowiednich aktywności i zadbaniem o ich właściwy przebieg działania.
Zgodnie z założeniami funkcjonalnymi omówionymi w podrozdziale 3.1, obsługa notatek tekstowych miała obejmować zarządzanie nimi od chwili powstania do usunięcia.
W programie odpowiadają za to następujące klasy.

- *Reporter*
  Jako aktywność dostarcza m.in. interfejsu użytkownikowi. Elementy widoczne na ekranie zostały zdefiniowane w odpowiednich plikach XML i z nią skojarzone. Głównym celem klasy jest wyświetlanie listy zebranych notatek. Obejmuje to również możliwość ich prezentacji uwzględniając różne rodzaje sortowania: malejąco i rosnąco według daty dodania, malejąco i rosnąco według priorytetu. Stworzenie tych funkcjonalności wiązało się z organizacją kwestii dostępu do bazy danych poprzez zdefiniowanie odpowiedniego adaptera, co zostało omówione w podrozdziale 3.3.1. Co więcej, opcje sortowania można niezależnie wzbogacać, gdyż znajdują się one w osobnym pliku XML powiązanym z kolejnym rodzajem adaptera gromadzącym te funkcje i udostępniającym je do komponentu *Spinner* przypominającego klasyczny element DropDownList. Ponadto z poziomu tej klasy istnieje możliwość usunięcia wybranej notatki, jej edycji oraz dodania kolejnej. W wymienionych dwóch ostatnich funkcjach odpowiadają za to zdefiniowane intencje, mające na celu wywołanie stosownych klas. Należy dodać, że po wprowadzeniu notatki, użytkownik otrzymuje odświeżony ich wykaz wraz z komunikatem o nowości. Odpowiada za to definicja metody *onActivityResult()* wywoływana bezpośrednio w tej klasie po zakończeniu działania aktywności dodającej wpis.
- *DataSaver*
  Odpowiedzialna jest za dodawanie nowej notatki. Będąc aktywnością stanowi równocześnie interfejs służący do pobierania od użytkownika następujących danych: tytuł notatki, treść, priorytet. Wywołana jest przez intencję w klasie Reporter, a po udanym zapisie notatki do bazy danych zwraca do klasy informację o rezultacie działania poprzez następujące instrukcje:

```
Intent intent = new Intent();
setResult(RESULT_OK, intent);
```

  gdzie:

  setResult() – powoduje wywołanie metody onActivityResult() wraz z przekazanymi argumentami w klasie, w której poprzez odpowiednią intencję ta aktywność została powołana do istnienia.
- *DataViewer*
  Jest aktywnością wywoływaną poprzez intencję w klasie Reporter.

Dostarcza użytkownikowi interfejsu w formie wyświetlania szczegółów konkretnej notatki pobierając stosowne dane z bazy danych. Klasa posiada także metody implementujące wywoływanie kolejnych aktywności umożliwiając w ten sposób edycję notatki oraz wysłanie jej poprzez wiadomość sms.

- *DataEditor*
  Odpowiedzialna za dostarczenie interfejsu do edycji notatki. Dodatkowo na wypadek nieprzewidywanego przerwania jej bieżącej aktywności, została zaimplementowana metoda onSaveInstanceState() omówiona w podrozdziale 2.3.1. W ten sposób w przypadku nieoczekiwanej zmiany stanu z Resumed na inny, nowo wprowadzone, ale nie zapisane przez użytkownika dane nie zostają utracone.
- *Sender*
  Stanowi aktywność wywoływaną poprzez intencję zdefiniowaną w klasie DataViewer. Jej celem jest udostępnienie możliwości wysłania treści notatki sms-em. Zrealizowane jest to poprzez wywołanie intencji z akcją *send* jako argumentem. Ponadto dostarcza użytkownikowi interfejsu w postaci komponentu zawierającego dane kontaktowe potencjalnych odbiorców wiadomości oraz treść notatki, którą również można poddać edycji. Element gromadzący informacje o osobach, do których można wysłać sms jest budowany w oparciu o opisane poniżej klasy ContactsAdapter oraz ExtendedContactsAdapter.
- *ContactsAdapter*
  Jest to klasa abstrakcyjna, będąca podstawą do tworzenia adapterów, służących do pobierania i formatowania danych przeznaczonych do prezentacji w komponencie Spinner.
- *ExtendedContactsAdapter*
  Stanowi klasę odpowiedzialną za utworzenie konkretnego adaptera dla komponentu Spinner. Wykorzystując możliwości dostarczane przez mechanizm Content provider omówiony w podrozdziale 2.3.4, zrealizowane zostało odbieranie danych zapisanych w książce adresowej urządzenia.

Kolejny moduł stanowiły notatki rysunkowe. Pod względem funkcjonalnym miały one pełnić mobilną wersję nawiązania do dostępnych na rynku edytorów graficznych, np. programu Paint dostępnego dla systemu Windows. Cel został zrealizowany poprzez budowę następujących klas.

- *DataViewerPaint*
  Aktywność dostarcza użytkownikowi interfejs w postaci listy dostępnych notatek rysunkowych. Ponadto dostępna jest możliwość ich sortowania względem daty dodania: malejąco i rosnąco. Klasa implementuje również funkcję usuwania notatek rysunkowych oraz dodawania nowych. W tym celu poprzez intencję wywoływana jest aktywność zdefiniowana w klasie PaintService, a po jej pomyślnym ukończeniu utworzony rysunek jest zwracany łącznie z intencją i zapisywany do bazy danych z nazwą

w formacie:

rys_RMD

gdzie R, M, D to odpowiednio rok, miesiąc i dzień zapisu.

- *PaintService*
  Stanowi aktywność udostępniającą użytkownikowi interfejs w postaci arkusza do rysowania. Zostało to zrealizowane poprzez utworzenie w tej klasie klasy wewnętrznej *DrawingView*. Oprócz możliwości rysowania, dla użytkownika dostępny jest wybór koloru pisaka, grubości linii, opcja mazania, rysowanie figur w postaci prostokątów. Zapis związany jest z przekształceniem przestrzeni do rysowania do bitmapy.
- *DrawingView*
  Jest klasą wewnętrzną klasy PaintService i jednocześnie będącą potomkiem standardowo zdefiniowanej klasy *View*. Jako dane składowe zawiera m.in. obiekty klas zdefiniowanych w pakiecie *android.graphics*. Do użytych w tym przypadku należą obiekty klas: *Bitmap*, *Canvas*, *Path* i *Paint*. Klasa Canvas została wykorzystana do utworzenia powierzchni do rysowania, Path oraz Paint do tworzenia konkretnych rysunków, w tym ustawiania właściwości pisaka, a Paint do konfiguracji koloru. W kwestii rysowania kluczową rolę odegrała implementacja obsługi zdarzenia *onTouchEvent*. Zadanie zostało rozwiązane w ten sposób, że w zależności od fazy ruchu rozpoznawanej przez klasę *MotionEvent* wykonuje się odpowiednie operacje oparte na współrzędnych położenia obiektu dotykającego ekran.
- *PaintReporter*
  Stanowi aktywność implementującą interfejs użytkownika prezentując zapisaną notatkę rysunkową. Wywoływana jest przez intencję zdefiniowaną w klasie DataViewerPaint. Na podstawie dostarczonego poprzez nią identyfikatora oraz za pomocą zdefiniowanego adaptera odbiera określony rysunek z bazy danych i zamienia go na bitmapę. Następnie zostaje on osadzony w stosownie zdefiniowanym do tego pliku XML w komponencie *ImageView*.
- *ColoursService*
  Klasa jest potomkiem klasy *Dialog* zdefiniowanej w pakiecie *android.app*. Należy dodać, że od strony praktycznej Dialog stanowi strukturę, która dostarcza interfejs użytkownika w postaci okna, ale zajmuje tylko część ekranu. Co więcej, może istnieć jedynie w obrębie aktywności, która go tworzy i równocześnie ma kontrolę nad jego cyklem życia. W związku z tym obiekt klasy ColoursService jest tworzony w klasie PaintService. W aplikacji odpowiada za dostarczenie użytkownikowi palety kolorów pisaka.
- *StrokeServiceDialog*
  Klasa dziedziczy po klasie Dialog omówionej przy opisie klasy ColoursService. Jej obiekt tworzony jest w klasie PaintService.

Odpowiada za dostarczenie użytkownikowi menu wyboru grubości linii rysowanej przez pisak.

Ostatnim spośród wyodrębnionych modułów był ten, który miał realizować tworzenie notatek zdjęciowych. Efektem prac nad nim są poniższe klasy.

- *DataViewerCamera*
  Klasa ta jako aktywność dostarcza interfejs w formie listy notatek zdjęciowych. Udostępnia także możliwość wyświetlenia ich posortowanych według daty: rosnąco i malejąco. Ponadto zawiera implementacje metod odpowiedzialnych za usuwanie, prezentację oraz dodawanie notatek w formie zdjęć.
- *PhotoReporter*
  Stanowi aktywność, a dostarczanym przez nią interfejsem jest notatka zdjęciowa. Korzystając ze specjalnie zdefiniowanego adaptera, który zostanie omówiony w dalszej części rozdziału, sięga do bazy danych po zdjęcie, zamienia go na bitmapę i prezentuje na ekranie.
- *EditionPaintService*
  Jest aktywnością wywoływaną przez intencję zdefiniowaną w klasie PhotoReporter. Zawiera utworzoną dla dostarczenia interfejsu użytkownika klasę *EditionView*.
- *EditionView*
  Stanowi klasę wewnętrzną klasy EditionPaintService i dziedziczy po klasie *View*. Używając obiektu klasy *Canvas* jako swojej danej składowej zapewnia powierzchnię do nanoszenia rysunków w postaci bitmapy utworzonej z wydobytego z bazy danych zdjęcia.

3.3.3. Zarządzanie mapą.

Etap tworzenia aplikacji, który stanowił kluczową rolę w spełnieniu jej funkcjonalności stanowiła obsługa mapy. Efektem pracy nad tą częścią są poniższe klasy.

- *MapNote*
  Jest potomkiem klasy *MapActivity*, która daje możliwość umieszczenia na ekranie mapy. Sama mapa została osadzona w komponencie *MapView* co wymagało odpowiedniej konstrukcji kolejnego pliku XML.
  Należy dodać, że aby móc wykorzystywać mapę dostarczaną przez Google aplikacja, którą się implementuje musi być zbudowana w oparciu o dodatkowe Google API. Następnie wymagane jest uzyskanie tzw. *Google Maps API Key*. Taki klucz wpisuje się do wspomnianego wcześniej pliku XML. Natomiast jego wygenerowanie opiera się na *odcisku palca* określanym jako *MD5 fingerprint* certyfikatu, który służy do podpisywania danej aplikacji.
  Omawiana klasa implementuje również interfejs *LocationListener*. Taka konstrukcja umożliwia wykorzystywanie wszystkich mechanizmów wykrywania lokalizacji, które udostępnia urządzenie. Określane jest to jako LBS (*Location – Based Services*).

W związku z tym, że zgodnie z wymaganiami funkcjonalnymi aplikacji, ma ona rejestrować bieżące położenie użytkownika, działanie w tym obszarze oparte jest na definicji metody *onLocationChanged()*. Odpowiada ona m.in. za odczytanie aktualnej lokalizacji, a ta zgodnie z odpowiednią instrukcją jest ustalana w odstępach 60000ms lub po przemieszczeniu się w odległości 1m. Dalsze działanie metody onLocationChanged() zostanie omówione przy okazji opisu klasy MapLayout, gdyż wykorzystuje ona jej usługi.

- *MapLayout*
  Jest potomkiem klasy *ItemizedOverlay*. Taka budowa umożliwia nałożenie na mapę warstwy i zarządzanie nią. W związku z tym jedną z danych składowych klasy jest lista potencjalnych warstw. W aplikacji używane są dwie. Pierwsza zawiera element wskazujący bieżącą lokalizację. Natomiast druga gromadzi elementy pokazujące miejsca, w których zapisane zostały notatki. Kontrolę nad warstwami pełni omówiona wcześniej klasa MapNote, ponieważ zawiera ona obiekt klasy MapLayout. Zrealizowane jest to w ten sposób, że w przypadku każdorazowego uaktualnienia bieżącego położenia, w metodzie onLocationChanged() klasy MapNote na mapie umieszczane są stosowne warstwy. Należy jednocześnie dodać, że warstwa zawierająca punkty, dla których wpisano notatki zbiera te dane korzystając z adaptera omówionego w podrozdziale 3.3.1.

Dodatkową wykonaną czynnością w kwestii korzystania z usług lokalizacji urządzenia stanowiło nadanie odpowiednich zezwoleń w pliku manifest.

3.3.4. Organizacja archiwizacji danych.

Kolejnym etapem implementacji aplikacji, a jednocześnie częścią, która miała stanowić rozszerzenie do dotychczas utworzonych funkcjonalności było opracowanie mechanizmu archiwizacji danych. Efektem prac jest utworzenie modułu, o którym mowa w tym podrozdziale oraz aplikacja przeznaczona dla komputera z systemem operacyjnym Windows, której sposób realizacji został opisany w rozdziale 4.

W celu udostępnienia możliwości gromadzenia wszystkich zapisanych notatek należało wzbogacić aplikację o funkcję połączenia z innym urządzeniem oraz zarządzanie procesem przesyłania danych. Zrealizowano to wykorzystując Bluetooth. Poniższe klasy złożyły się na wykonanie opisanych funkcjonalności.

- *SynchronizeManager*
  Stanowi aktywność dostarczając użytkownikowi interfejsu w postaci okna wyboru czynności związanych z archiwizacją. Ponadto jest ona źródłem wywołań określonych zadań i odbiorcą rezultatów.
  Przede wszystkim jako jedną z danych składowych zawiera obiekt klasy *BluetoothAdapter* znajdującej się w pakiecie *android.bluetooth*. Jeśli urządzenie obsługuje moduł bluetooth, wówczas jest on inicjalizowany

domyślnym adapterem. W efekcie przy starcie aktywności sprawdzany jest stan połączenia bluetooth. W przypadku, gdy jest on aktualnie wyłączony, definiuje się intencję z akcją włączenia jako parametrem. Inną składową jest obiekt klasy SynchService, która zostanie omówiona w dalszej części tego podrozdziału. W klasie SynchronizeManager odpowiada on za przebieg połączenia z innym urządzeniem, w tym oczekiwanie na jego rozpoczęcie i obsługę zakończenia. Dodatkowo w klasie wykorzystuje się obiekt klasy *Handler* z pakietu *android.os* umożliwiający komunikowanie się wątków poprzez zamieszczanie informacji w obiekcie klasy Message pełniącej rolę opakowania dla nich (*Bundle*). Przy jego tworzeniu metodę *handleMessage()* zdefiniowano w ten sposób, aby odbierała ona informacje o stanie połączenia. Taka struktura pozwala na bieżące przekazywanie użytkownikowi wiadomości o ewentualnym połączeniu oraz umożliwia zwalnianie niektórych niepotrzebnych zasobów, gdy np. dojdzie do zerwania połączenia. Oprócz tego metoda handleMessage() obsługuje przebieg przesyłania danych do innego urządzenia.
Należy przy tym dodać, że transfer danych realizowany jest etapami. Najpierw urządzenie odbierające otrzymuje informacje o ich typie, później o rozmiarze, a następnie te właściwe. Każda faza jest potwierdzana przez odbiorcę, co powoduje ciągłość i spójność komunikacji.

- *SynchService*
  Jest klasą, która zarządza przebiegiem połączenia bluetooth. Jej obiekt posiada klasa SynchronizeManager. Klasa SynchService zawiera trzy klasy wewnętrzne dziedziczące po klasie *Thread*, których implementacja służy podzieleniu sesji bluetooth na trzy odrębne stany. Są to: *WaitingForConnectionThread*, *AttemptToConnectThread* oraz *ConnectedWithDeviceThread*. Zostaną one omówione w dalszej części. Dla klasy SynchService istotne są metody startWaiting(), connectWithDevice() i manageConnectedDevice(). Odpowiadają one za uruchomienie odpowiednich wątków, ustawienie stanu połączenia i przesłanie tej informacji do klasy SynchronizeManager.
- *WaitingForConnectionThread*
  Implementuje wątek, który działa oczekując na próbę połączenia z innym urządzeniem. Zrealizowane to zostało wykorzystując zasoby pakietu android.bluetooth poprzez utworzenie w konstruktorze obiektu typu *BluetoothServerSocket* reprezentującego gniazdo nasłuchujące. Użyto w tym celu metody *listenUsingRfcommWithServiceRecord()* dostarczanej przez Google API. W ten sposób zapewnione jest także szyfrowanie komunikacji poprzez zwrócone gniazdo.
  W konsekwencji po nadejściu zgłoszenia do połączenia wątek przestaje działać.
- *AttemptToConnectThread*
  Jest implementacją wątku próbującego nawiązać połączenie z urządzeniem. Działa dość krótko, ponieważ zarówno w razie udanej jak

i nieudanej próby w efekcie przestaje działać. W przypadku powodzenia uruchamiana jest metoda, która ma na celu rozpocząć działanie wątku zaimplementowanego w klasie ConnectedWithDeviceThread.

- *ConnectedWithDeviceThread*
  Klasa stanowi implementację wątku odpowiedzialnego za połączenie z innym urządzeniem. Poprzez nią zrealizowany jest przesył danych pomiędzy urządzeniem a komputerem.
- *DeviceSeacher*
  Jako aktywność dostarcza użytkownikowi interfejsu w postaci listy urządzeń sparowanych. Wywoływana jest w klasie SynchronizeManager poprzez zdefiniowaną intencję. W przypadku powodzenia, tzn. wyboru danego urządzenia spośród zaproponowanych, obsługiwany zostaje rezultat tej intencji w formie metody inicjalizującej połączenie.
- *DatabaseManager*
  Jest klasą abstrakcyjną stanowiącą podstawę do tworzenia klas obsługujących kwestię rodzaju danych, które są przesyłane. Operuje ona na adapterze, który został opisany w podrozdziale 3.3.1. Klasy będące jej potomkami to: NoteClient, PhotoClient, DrawingClient i PlaceClient.
- *NoteClient*
  Klasa dziedziczy po klasie DatabaseManager. Realizuje dostęp do przesyłanych notatek tekstowych.
- *PhotoClient*
  Jest klasą dziedziczącą po klasie DatabaseManager. Realizuje dostęp do notatek zdjęciowych.
- *DrawingClient*
  Klasa dziedziczy po klasie DatabaseManager, ale nie bezpośrednio. Jej rodzicem jest klasa PhotoClient. Realizuje dostęp do przesyłanych notatek rysunkowych.
- *PlaceClient*
  Klasa dziedziczy po klasie DatabaseManager. Realizuje dostęp do przesyłanych informacji dotyczących miejsc jedynie oznaczonych przez użytkownika.

3.3.5. Implementacja innych funkcji.

Pozostałe wymagania funkcjonalne wymienione z podrozdziale 3.1 zrealizowano poprzez definicje następujących klas.

- *DistanceService*
  Jako aktywność dostarcza użytkownikowi interfejs umożliwiający podanie odległości, w jakiej mają być wskazywane notatki od bieżącej lokalizacji. Wywoływana jest przez intencję zdefiniowaną w klasie MapNote i w efekcie zwracającą wynik w postaci liczby odpowiadającej odległości.

- *LocalizationInformer*
  Jest aktywnością zapewniającą wyświetlanie elementu pozwalającego na wybór odbiorcy wiadomości sms. Doprowadzenie danych do użytego komponentu zapewnia adapter zdefiniowany w klasie ExtendedContactsAdapter, a został on opisany w podrozdziale 3.3.2.
- *TasksManager*
  Dziedziczy po klasie Dialog, którą omówiono w podrozdziale 3.3.2 przy opisie klasy ColoursService. Reprezentuje okno z możliwością wyboru typu notatek do wyświetlenia oraz oznaczenia miejsca, dla którego to okno wywołano. Dodatkowo wykorzystując adapter opisany w podrozdziale 3.3.1, użytkownik informowany jest o ilości notatek każdego z typów dla wybranej lokalizacji.

4. Aplikacja MapNote Synch.

Aplikacja MapNote Synch jest przeznaczona dla komputera z systemem Windows. Służy do zarządzania danymi pochodzącymi z programu MapNote.

4.1. Założenia.

Założenia funkcjonalne aplikacji:

- komunikacja z programem MapNote,
- synchronizacja danych z aplikacji MapNote,
- archiwizacja notatek rysunkowych, tekstowych, zdjęciowych oraz zaznaczonych lokalizacji,
- prezentacja notatek na mapie,
- wyznaczanie trasy odtwarzającej miejsca powstawania notatek,
- wyznaczanie trasy łączącej zarejestrowane miejsca pobytu,
- wyświetlanie treści notatek tekstowych,
- wyświetlanie notatek zdjęciowych oraz rysunkowych,
- zapis notatek na dysku,
- usuwanie notatek.

4.2. Dobór narzędzi.

4.2.1. Wykaz.

Wykonanie aplikacji związane było z wykorzystaniem następujących narzędzi:

- środowisko programistyczne Microsoft Visual Studio 2008,
- system zarządzania bazą danych SQLite,
- biblioteka 32feet.NET.

Prace zostały wykonane w języku programowania C#, framework 2.0.

4.2.2. Instalacja bazy danych.

Mimo tego, że baza danych SQLite nie wymaga żadnych instalacji, to aby móc z niej korzystać w środowisku .NET należy wykonać kilka operacji.
Przede wszystkim udostępniany jest do tego celu specjalny provider, którego trzeba dodatkowo pobrać. Po jego instalacji zostaje oddana do użycia biblioteka *System.Data.SQLite*. Umożliwia on wykonywanie wszelkich poleceń związanych z bazą.

4.2.3. Biblioteka 32feet.NET.

Biblioteka daje możliwość wykonywania połączeń bluetooth. Dostarczana jest w postaci pliku InTheHand.NET.Personal.dll. Ponadto można jej używać na urządzeniach obsługujących zarówno stos Microsoft jak i Widcomm/Broadcomm.

W programie wykorzystanie biblioteki ujawnia się poprzez pracę z poniższymi typami.

- *BluetoothRadio* - dostarcza informacji o radiu bluetooth, łącznie z danymi o jego dostępności w danym urządzeniu.
- *BluetoothClient* - umożliwia odbieranie danych pochodzących ze strumienia.
- *BluetoothListener* - służy do nasłuchiwania na ewentualne zgłoszenie się urządzenia do połączenia.
- *BluetoothService* - klasa funkcji usługowych użyta do nadania numeru UUID.

4.3. Implementacja.

Proces tworzenia aplikacji podzielono na moduły wyodrębnione w oparciu o rodzaj działań, za które miały być odpowiedzialne.

4.3.1. Zarządzanie bazą danych.

Jednym z celów, które postawiono w kwestii realizacji w programie dostępu do bazy danych było utworzenie mechanizmu charakteryzującego się względnie łatwą możliwością ewentualnego wprowadzania zmian w przyszłości.
W związku z tym w aplikacji zdefiniowano następujący interfejs.

```
public interface DatabaseManager
{
   void insert(String command);
   void insert(String command, byte[] data);
   void update(String command);
   DataTable select(String command);
   void delete(String command);
}
```

Konstrukcja ta pozwoliła uzyskać sposobność dostępu do bazy danych niezależnie od jej typu, ponieważ polecenia mogą przyjmować różne formy.
Takie rozwiązanie oznaczało konieczność utworzenia klasy implementującej podany interfejs. Zgodnie z tym, w następnym etapie powstała klasa *DeviceDatabase*. Posiada ona prywatny obiekt klasy *SQLiteConnection*, który reprezentuje połączenie z bazą danych. Definicja klasy DeviceDatabase w większości obejmuje implementację metod opisanego wcześniej interfejsu. Tak więc, na podstawie przekazanego w ramach argumentu polecenia wykonuje się określone instrukcje.
Niemniej jednak należy uwzględnić jej konstruktor. Jako argument przyjmuje on nazwę pliku bazy. Jeżeli taka baza nie istnieje, wówczas jest ona tworzona wraz z odpowiednimi tabelami. W przeciwnym razie następuje bezpośrednia inicjalizacja obiektu typu SQLiteConnection.
W aplikacji ujawnia się to w ten sposób, że w chwili pierwszego sięgnięcia do bazy w razie, gdy nie istnieje, na dysku powstaje określony plik z jej początkową zawartością, czyli pustymi tabelami.

4.3.2. Synchronizacja.

Kwestia synchronizacji danych stanowiła kluczowy etap budowy aplikacji. W tym aspekcie wykorzystano w dużym stopniu ideę wzorca Obserwator na co wskazują poniżej omówione, a zdefiniowane w programie interfejsy oraz klasy.

- *Manager*
  Jest to interfejs z metodami: manage(), informClientDevices(), registerClientDevice(), deleteClientDevice(). W kontekście aplikacji klasy implementujące go miały z założenia zarządzać rejestracją połączeń z innymi urządzeniami.
- *BluetoothDeviceManager*
  Klasa implementuje interfejs Manager. Jej działanie polega na uruchomieniu wątku z obiektem klasy BluetoothListener omówionej w podrozdziale 4.2.3, a następnie w przypadku udanego nawiązania ewentualnego połączenia, tworzy obiekt dziedziczący po klasie ClientDevice (opisanej w dalszej części) i uruchamia jego metodę startActivity().
- *ClientDevice*
  Jest interfejsem z metodami startActivity(), update(), clear(). W aplikacji klasy implementujące go mają organizować komunikację z połączonym urządzeniem.
- *BluetoothClientDevice*
  Klasa implementuje interfejs ClientDevice. Jej obiekt zostaje tworzony w klasie BluetoothClientDevice, co oznacza nawiązanie połączenia z innym urządzeniem. Należy dodać przy tym, że do jej konstruktora zostaje przekazany obiekt klasy BluetoothClient opisanej w podrozdziale 4.2.3. Wywołanie na rzecz obiektu klasy BluetoothClientDevice metody startActivity() powoduje uruchomienie wątku, którego zadaniem jest oczekiwanie na nadejście danych ze strumienia, a następnie odczytanie i wykonanie dalszych operacji.
  W związku z faktem, że aplikacja przeznaczona jest do współpracy z programem MapNote, to klasa realizuje we wspomnianym wątku drugie ogniwo komunikacji opisanej w podrozdziale 3.3.4 przy okazji omawiania klasy SynchronizeManager. Nawiązując do tego co zostało tam przedstawione należy uzupełnić, że to w klasie BluetoothClientDevice zdefiniowana jest metoda sendInfo(), której zadaniem jest przesłanie do urządzenia będącego źródłem danych informacji potwierdzającej ich otrzymanie.

Ponadto, istotną kwestią jest fakt, że w aplikacji użytkownik jest na bieżąco informowany o przebiegu synchronizacji danych. Umożliwione to zostało poprzez użycie utworzonych delegacji, których deklaracje zamieszczono poniżej:

```
private delegate void StateConnection(String info, String advice)
private delegate void ButtonConnectionState(bool visibility, bool
                                                        visibility2)
```

4.3.3. Obsługa prezentacji danych.

Zakładając posiadanie danych otrzymanych z aplikacji MapNote, należało zorganizować ich prezentację i przetwarzanie.
Zgodnie z tym w pierwszej kolejności został rozpatrzony problem obsługi interfejsu użytkownika. W efekcie do utworzenia mechanizmu zarządzania wyświetlaniem poszczególnych okien użyto wzorca Fabryka Abstrakcyjna. Zdefiniowano w tym celu następujące typy.

- *FormsManager*
  Jest to interfejs z deklaracją metody create() zwracającej okno użytkownika.
- *NotesFormsManager*
  Implementuje interfejs FormsManager definiując metodę create(). W efekcie zwraca ona obiekt typu NotesForm.
- *PhotosFormsManager*
  Implementuje interfejs FormsManager definiując metodę create(). W efekcie zwraca ona obiekt typu PhotosForm.
- *DrawingsFormsManager*
  Implementuje interfejs FormsManager definiując metodę create(). W efekcie zwraca ona obiekt typu DrawingsForm.
- *VisitedPlacesFormsManager*
  Implementuje interfejs FormsManager definiując metodę create(). W efekcie zwraca ona obiekt typu VisitedPlacesForm.
- *NotesForm*
  Klasa reprezentuje okno użytkownika służące do wyświetlania informacji dotyczących notatek tekstowych.
- *PhotosForm*
  Reprezentuje okno użytkownika służące do wyświetlania informacji dotyczących notatek zdjęciowych.
- *DrawingsForm*
  Klasa reprezentuje okno użytkownika służące do wyświetlania informacji dotyczących notatek rysunkowych.
- *VisitedPlacesForm*
  Klasa reprezentuje okno użytkownika służące do wyświetlania informacji dotyczących zarejestrowanych w aplikacji MapNote miejsc pobytu.

Klasy NotesForm, DrawingsForm, PhotosForm oraz VisitedPlacesForm mają na celu dostarczać użytkownikowi interfejs w postaci okna prezentującego informacje dotyczące określonego typu danych. Co więcej, każda z nich daje możliwość otrzymania posortowanych wyników. Do formatek obsługujących poszczególne okna wbudowano również mapę, na którą nanoszone są określone punkty lokalizacji. W związku z tym, że taka realizacja wymagała odnoszenia się do bazy danych, wprowadzono mechanizm ujednolicający polegający na tym, że każda z wymienionych czterech klas posiada swój obiekt, który dostarcza jej danych, jakie potrzebuje. Klasy, które zdefiniowano dla spełnienia opisanej

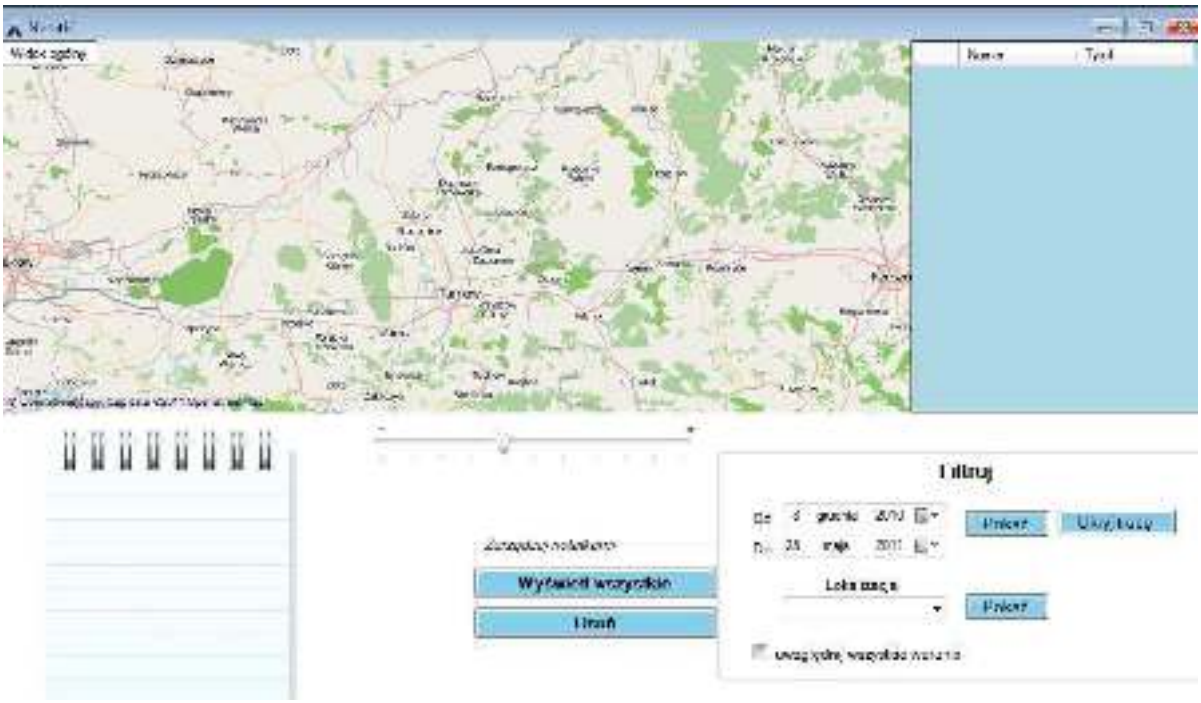
zależności to wymienione poniżej.

- *FormUser*
  Jest klasą nadrzędną w stosunku do klas NotesFormUser, PhotosFormUser, DrawingsFormUser oraz VisitedPlacesFormUser.
- *NotesFormUser*
  Jej obiekt realizuje dostęp do bazy danych dla okna przeznaczonego notatkom tekstowym.
- *PhotosFormUser*
  Jej obiekt realizuje dostęp do bazy danych dla okna przeznaczonego notatkom zdjęciowym.
- *DrawingsFormUser*
  Jej obiekt realizuje dostęp do bazy danych dla okna przeznaczonego notatkom rysunkowym.
- *VisitedPlacesFormUser*
  Jej obiekt realizuje dostęp do bazy danych dla okna przeznaczonego zarejestrowanym w czasie pobytu lokalizacjom.

5. Podsumowanie.

Celem niniejszego dokumentu było przedstawienie przebiegu realizacji części praktycznej pracy dyplomowej, czyli aplikacji dla platformy Android zapewniającej archiwizację danych na komputerze z systemem Windows. Ponadto uwzględniono kwestie teoretyczne ściśle związane z wykonaniem zadania, a odnoszące się do nowych technologii. Należy podkreślić, że utworzona aplikacja wypełnia wszystkie postawione wymagania funkcjonalne. W związku z tym daje możliwość zarządzania w szerokim zakresie różnego rodzaju notatkami skojarzonymi z mapą i umieszczonymi na urządzeniu mobilnym.
Co więcej, dane te mogą zostać przeniesione do komputera i także przetwarzane, ponieważ dodatkowo został utworzony to tego celu odrębny program komunikujący się z tym, przeznaczonym dla systemu Android.

Rys. 5.1. Przykładowy ekran aplikacji MapNote.



Rys. 5.2. Przykładowy ekran aplikacji MapNote Synch.

Należy zauważyć, że ciągle istnieje możliwość rozbudowywania aplikacji o nowe moduły. Oto przykładowe kierunki ewentualnego rozszerzenia programu:

- stowarzyszenie notatek z kalendarzem,
- synchronizacja danych pomiędzy innymi urządzeniami mobilnymi.

6. Bibliografia.

[1] Conder S., Darcey L., *Android Wireless Application Development Second Edition*, Addison – Wesley, 2010
[2] Conder S., Darcey L., *Android Application Development in 24 Hours*, SAMS, 2010
[3] Burnette E., *Hello, Android*, The Pragmatic Bookshelf, 2009
[4] Murphy Mark L., *The Busy Coder's Guide to Android Development*, CommonsWare, 2008
[5] http://developer.android.com
[6] http://sqlite.org
[7] http://inthehand.com
[8] http://greatmaps.codeplex.com