Uniwersytet Marii Curie-Skłodowskiej w Lublinie

Wydział Matematyki, Fizyki i Informatyki

Instytut Informatyki

Marcin Gawski

Nr indeksu: 218220

System monitorujący AutoAndroid

The monitoring system AutoAndroid

Praca licencjacka

przygotowana w Instytucie Informatyki

promotor: dr Barbara Gocłowska

Lublin 2011

Spis treści

Wstęp

Obecnie rynek urządzeń mobilnych rozwija się w bardzo szybkim tempie. Dzisiaj prawie każdy posiada swój własny telefon. Jednak nie są to już zwykłe telefony a zaawansowane technologicznie urządzenia, naszpikowane różnego rodzaju czujnikami. W ostatnim czasie na rynek mobilny wkroczył prawdziwy światowy gigant – firma Google. Wydała ona rewelacyjny system na platformę mobilną o nazwie Android, który podbija serca użytkowników.

W pierwszym rozdziale pracy została przedstawiona krótka historia i kierunek rozwoju palmtopów i telefonów komórkowych. Opisane są również nowoczesne urządzenia mobilne i ich wbudowane czujniki. Omówiony został także system Andorid. Drugi rozdział skupia się na metodach geolokalizacji. Przedstawione zostały trzy obecnie dostępne metody uzyskiwania lokalizacji pozycji, ich wady oraz zalety. W trzecim rozdziale jest dokładny opis implementacji programu, który stanowi część praktyczną tej pracy. Aplikacja została napisana na platformę Android. Jest ona przykładem wykorzystania możliwości dzisiejszych, zaawansowanych technologicznie telefonów komórkowych. Umożliwia dokładne monitorowanie pojazdu, udostępniając funkcjonalności takie jak: ewidencja wydatków, raporty spalania, przypomnienia o rutynowych badaniach, pomiar przyśpieszenia, licznik samochodowy. Na podstawie danych wprowadzanych przez użytkownika, generowane są ciekawe statystyki i wykresy. Najciekawszymi elementami programu są pomiar przyśpieszenia oraz funkcja licznika samochodowego. Do pomiaru osiągów samochodu został wykorzystany system GPS, zaś prędkościomierz oprócz tego modułu, wykorzystuje również akcelerometr.

Rozdział 1. Rozwój urządzeń mobilnych

Rynek urządzeń mobilnych istnieje już od wielu lat. Wraz z rozwojem techniki i informatyki, powstawały coraz to nowsze przenośne komputery o różnych zastosowaniach. Pierwsze były proste kalkulatory, które zyskały dużą popularność. Później pojawiły się telefony komórkowe oraz laptopy. Pierwszy mobilny telefon komórkowy został wyprodukowany w 1973 roku[1], zaś pierwszy komercyjny laptop został zaprojektowany w 1979 roku[2]. W tym samym czasie pracowano nad urządzeniami do nawigacji satelitarnej oraz urządzeniami PDA (ang. *Personal Digital Assistant* lub *Personal Data Assistant*), inaczej nazywanymi palmtopami. Pierwszy palmtop został wyprodukowany w 1984 roku[3]. Na początku rozwój urządzeń mobilnych był stosunkowo powolny, lecz ciągle nabierał tempa wraz z postępem technologicznym i miniaturyzacją układów elektronicznych.

Początkowo wszystkie te urządzenia były tworzone głównie dla wojska oraz dużych firm i ich pracowników. Zmieniło się to na przestrzeni ostatnich kilkunastu lat. Urządzenia mobilne stały się ogólnie dostępne dla zwykłych ludzi, dzięki czemu dziś ich rynek przeżywa prawdziwy rozkwit. Obecnie najprężniej rozwija się gałąź telefonów komórkowych, które połączyły w sobie funkcjonalność kilku urządzeń: telefonu, palmtopa, laptopa, nawigacji satelitarnej, bankomatu, telewizora i innych.

1.1 Ewolucja urządzeń PDA

Pierwsze urządzenia PDA przypominały zaawansowany kalkulator. Posiadały one klawiaturę oraz monochromatyczny wyświetlacz LCD (ang. *Liquid Crystal Display*). Podstawowymi funkcjami były: kalkulator, kalendarz, książka adresowa oraz notatnik. Szybko zastąpiono zwykły wyświetlacz, ekranem dotykowym, redukując przy tym ilość sprzętowych przycisków. Klawiatura fizyczna, została zamieniona na klawiaturę ekranową, zaś sam ekran był obsługiwany rysikiem. Dodano także kolejne moduły do komunikacji takie jak: złącze USB, bluetooth oraz IrDA (ang. *Infrared Data Association*). Pierwsze urządzenia PDA dysponowały mocą obliczeniową rzędu 4,91 MHz, 128 KB pamięci RAM oraz 256 KB pamięci ROM. [4]

W latach od 1999 do 2003 roku główną docelową grupą użytkowników palmtopów byli

biznesmeni. Liderami na rynku były firmy RIM[5] (linia urządzeń BlackBerry), Palm[6] (linia urządzeń Palm Treo), Compaq/HP[7] (linia urządzeń iPAQ). Dźwignią rynku urządzeń mobilnych była wtedy jeszcze mało znana firma HTC (ang. *High Tech Computer Corporation*), która na początku produkowała urządzenia na zlecenia innych dużych firm takich jak: Compaq/HP (produkcja urządzeń serii iPAQ), Fujitsu-Siemens, Palm[8]. Urządzenia BlackBerry pracowały pod systemem Blackberry OS, Palm Treo pod systemem Palm OS zaś iPAQ i HTC pod systemem Windows Mobile. Powstawały także inne palmtopy pracujące pod popularnym systemem Symbian. Funkcjonalność urządzeń została znacząco rozszerzona poprzez dodanie modułu GSM (ang. *Global System for Mobile Communications*). Możliwość prowadzenia rozmów w każdym miejscu i o każdej porze była bardzo pożądana w świecie biznesu, lecz nie była to jedyna funkcjonalność jaką dawał moduł GSM. Umożliwił on rozwój mobilnego internetu, a co za tym idzie stały dostęp do poczty internetowej. Początkowo transmisja danych odbywała się za pomocą technologii CSD (ang. *Circuit Switched Data*) w której płacono tak samo jak za zwykłe rozmowy telefoniczne, czyli za czas połączenia. Podczas takiego połączenia linia telefoniczna była zajęta. Później operatorzy GSM wprowadzili pakietową transmisję danych – GPRS (ang. *General Packet Radio Service*) w której nie płacono za czas, lecz za ilość przesłanych i odebranych danych. Takie połączenie umożliwiało jednoczesny dostęp do sieci oraz prowadzenie rozmów telefonicznych. Ceny połączeń internetowych były bardzo wysokie, dlatego mogli sobie na nie pozwolić jedynie ludzie zamożni i biznesmeni. Dzięki pakietowej transmisji danych użytkownik mógł być stale podpięty do sieci a wykorzystując technologię „e-mail push" był automatycznie powiadamiany o nowych wiadomościach, praktycznie w tym samym momencie gdy pojawiły się one na serwerze poczty. Powstały także pierwsze przeglądarki internetowe. W późniejszym okresie powstawały coraz to nowsze i szybsze technologie przesyłu danych, oparte na transmisji GPRS, takie jak: EDGE (ang. *Enhanced Data rates for GSM Evolution*) i HSDPA (ang. *High Speed Downlink Packet Access*). Palmtop z wbudowanym modułem GSM został nazwany smartphone'em. W 2003 roku powstał także jeden z pierwszych palmtopów z wbudowaną bezprzewodową kartą sieciową Wi-Fi, a był nim HP iPAQ H435x (wyprodukowany przez HTC)[8].

Po 2003 roku wykształciła się druga grupa docelowych odbiorców, którymi byli kierowcy. Na początku tego okresu powstał Garmin iQue 3600, czyli pierwszy palmtop z wbudowanym odbiornikiem GPS pracujący pod systemem Palm OS[9]. Dedykowane urządzenia do

nawigacji powstawały już wcześniej, lecz iQue był pierwszym który łączył w sobie pełną funkcjonalność palmtopa i nawigacji satelitarnej. W ślad za firmą Garmin poszli inni producenci palmtopów, wypuszczając na rynek cały wachlarz różnych urządzeń posiadających wbudowany moduł GPS. Stopniowo zaczęły pojawiać się programy do nawigacji samochodowej, dedykowane na różne systemy operacyjne. Najpopularniejsze były aplikacje na system Symbian oraz Windows Mobile. Najbardziej uznanym w Europie programem do nawigacji samochodowej był i nadal jest TomTom Navigator. Oprogramowanie to posiada najdokładniejsze mapy całej europy oraz działa pod systemami: Palm, Symbian oraz Windows Mobile. Do europejskiej czołówki należy także polski program AutoMapa, który posiada bardzo dobrze odwzorowanie sieci dróg nie tylko Polski, ale także i innych krajów europejskich. AutoMapa działa jedynie pod systemem Windows Mobile.

Rynek palmtopów rozwijał się a producenci palmtopów stopniowo zaczęli kierować się ku platformie Windows Mobile, odchodząc przy tym od systemów Symbian, Palm OS. Na popularności zaczęła także tracić firma BlackBerry. Do czołówki producentów dołączyła tajwańska firma HTC, która od początku produkowała urządzenia obsługujące wyłącznie system Microsoftu. Dzięki swoim innowacyjnym pomysłom, firma HTC w znacznym stopniu przyczyniła się do rozwoju urządzeń mobilnych. Jako pierwsza na świecie, wyprodukowała urządzenie pracujące pod systemem Windows Mobile, oraz jako jedna z pierwszych wprowadziła na rynek dotykowy wyświetlacz[10]. HTC produkowało nie tylko sprzęt, ale także cenione przez użytkowników oprogramowanie.

Ceny urządzeń PDA stopniowo zaczęły maleć. Już w 2006 roku można było kupić nowe wielofunkcyjne urządzenie posiadające procesor o częstotliwości 400 MHz, 64MB RAM i 128MB ROM, aparat fotograficzny, Wi-Fi, wbudowany moduł GSM i moduł GPS. Cena takiego urządzenia w Polsce oscylowała w granicach 1600zł. Jeżeli porównać tę kwotę z cenami ówczesnych lepszych telefonów komórkowych marki Nokia, można by dojść do wniosku, że zakup zwykłego biznesowego telefonu komórkowego był nieopłacalny. Co więcej, urządzenie PDA było pod każdym względem lepsze od telefonu komórkowego. Nic więc dziwnego, że powoli palmtopy zaczęły trafiać do zwykłych konsumentów, którzy wykorzystywali je jako telefon komórkowy, dodatkowo dysponując przy tym szeregiem nowych funkcji takich jak np. nawigacja samochodowa. Od tego momentu rozpoczął się prawdziwy wzrost popularności palmtopów. Producenci przestawili się z klienta biznesowego

na użytkowników zwykłych telefonów komórkowych. Lawinowo powstawały nowe aplikacje ułatwiające korzystanie z systemu Windows Mobile i wbudowanego telefonu. W wielu urządzeniach dodano aparat fotograficzny i położono większy nacisk na stronę multimedialną – słuchanie muzyki, oglądanie filmów oraz gry.

Postęp technologiczny i miniaturyzacja sprawiły, że dzisiaj na rynku mamy dostęp do urządzeń z procesorami dwurdzeniowymi o częstotliwości taktowania nawet 1.2GHz oraz praktycznie nie ograniczonej pamięci. Połączenie funkcjonalności palmtopa i telefonu komórkowego sprawiło, że częściej na urządzenia PDA mówimy smartphone. Z urządzeń PDA powstały też inne wyspecjalizowane urządzenia, oferujące tylko część funkcjonalności zwykłego palmtopa, takie jak np. E-book Reader, czy też rozszerzające funkcjonalność takie jak Tablet PC.

1.2 Ewolucja telefonów komórkowych

Pierwszy telefon który można uznać za mobilny zaprezentowała firma Motorola w 1973 roku. Telefon był rozmiaru cegły i ważył 0.8 kg. Posiadał klawiaturę numeryczną i kilka dodatkowych przycisków. Zawierał książkę telefoniczną w której można było zapisać do trzydziestu kontaktów. Jego podstawową funkcją była możliwość prowadzenia rozmów[11]. Początkowo rozwój przebiegał bardzo powoli, głównie z braku niezbędnych do funkcjonowania sieci komórkowych. Pierwsza sieć została uruchomiona w Japonii w 1979 roku i pokrywała obszar Tokyo. Jedną z pierwszych funkcji jaką dodano, była możliwość przesyłania wiadomości tekstowych – SMS (ang. *Short Message Service*). Pierwszy prawdziwy SMS został wysłany w 1993 roku w Finlandii [12].

W 1997 roku nastąpił gwałtowny wzrost popularności telefonii komórkowej. Pojawiali się nowi użytkownicy a zasięg sieci zaczął się bardzo szybko rozszerzać. Ten trend trwa do dziś. Powstało wiele firm zajmujących się produkcją telefonów komórkowych, w tym między innymi: Nokia, LG, Sagem, Nec, Samsung, Siemens, Sony Ericsson. Bardzo popularnym wśród komórek stał się system operacyjny Symbian. Każdy producent udostępniał urządzenia ze zmodyfikowaną przez siebie wersją systemu operacyjnego. Wprowadzono funkcje budzika, kalendarza, prostego notatnika oraz możliwość zmiany sygnału połączeń przychodzących. Rozwój funkcjonalności był nastawiony na stronę multimedialną. Producenci prześcigali się w tworzeniu nowych funkcji. Każdy chciał mieć coś, czego nie

miała konkurencja. Już w 2001 roku powstał pierwszy na świecie telefon, który potrafił odtwarzać muzykę w formacie MP3. Był to Samsung SGH-M100, który posiadał także przeglądarkę internetową WAP (ang. *Wireless Application Protocol*) [13]. Następną istotną rewolucją było dodanie gniazda na wymienne karty pamięci flash. Kolejne nowsze modele mogły pochwalić się polifonicznymi dzwonkami. Powoli zaczęły pojawiać się pierwsze telefony z kolorowymi wyświetlaczami, obsługującymi początkowo jedynie cztery kolory. Później głębia kolorów została rozszerzona. Następnie przyszła kolej na wbudowanie aparatu fotograficznego z możliwością kręcenia filmów. Zdjęcia z pierwszych tego typu urządzeń były bardzo słabej jakości. Wykonywane były w rozdzielczości QVGA a później VGA. Mimo to ciągle trwały prace nad poprawą tych parametrów. Po kilku latach jakość wbudowanej optyki znacząco się polepszyła. W 2008 roku Sony Ericsson stworzył linię telefonów nazwaną Cyber-shoot. Były to pierwsze telefony, które potrafiły robić zadowalające zdjęcia (jak na telefon) przy dobrym oświetleniu, oraz posiadały „mocną" lampę błyskową.

Stopniowo telefony stawały się coraz bardziej zaawansowanymi technologicznie urządzeniami. Jednak w pewnym momencie osiągnięto stan, w którym dalsza ewolucja telefonów została ograniczona przez niewielką moc obliczeniową, system operacyjny oraz słabe możliwości interakcji użytkownika z urządzeniem. Wprowadzono więc duże, dotykowe wyświetlacze, procesory o większej mocy obliczeniowej, oraz zrezygnowano z systemu Symbian.

1.3 Urządzenia typu smartphone

Kierunek ewolucji telefonów komórkowych podążał w stronę urządzeń PDA, zaś ewolucja palmtopów podążała w stronę telefonów komórkowych. Ostatecznie oba urządzenia zostały połączone w jeden wielofunkcyjny sprzęt o nazwie smartphone. Rewolucja nastąpiła w 2007 roku wraz z pojawieniem się na rynku pierwszego iPhone'a, produktu firmy Apple. Urządzenie posiadało dotykowy ekran o przekątnej 3.5 cala, procesor o mocy obliczeniowej 620 MHz oraz pracowało pod systemem iPhone OS. Produkt Apple doskonale łączył w sobie funkcjonalność zwykłego telefonu, oraz zaawansowanego urządzenia PDA. Posiadał doskonały interfejs użytkownika, który swoją responsywnością szybko zyskał sobie duże grono zwolenników. Był on także dostosowany do obsługi palcem i gestami, co wyróżniało

iPhone'a od zwykłych palmtopów które były obsługiwane rysikami. Apple w swoim urządzeniu wprowadziło wiele innowacyjnych rozwiązań. Wyposażyli je we wbudowane czujniki: przyśpieszenia, światła oraz odległości. Dzięki nim wachlarz możliwości urządzenia został znacząco rozszerzony. Kolejną innowacją było wprowadzenie sklepu aplikacji, z którego mógł korzystać każdy użytkownik iPhone'a. W markecie znajdowały się zarówno programy płatne, jak i darmowe. Market był genialnym pomysłem a zarazem okazją na zarobek dla programistów, którzy szybko zabrali się za pisanie aplikacji na tę platformę. Produkt Apple pojawił się w ofercie operatorów sieci komórkowych i przejął sporą część użytkowników zwykłych telefonów komórkowych. [14]

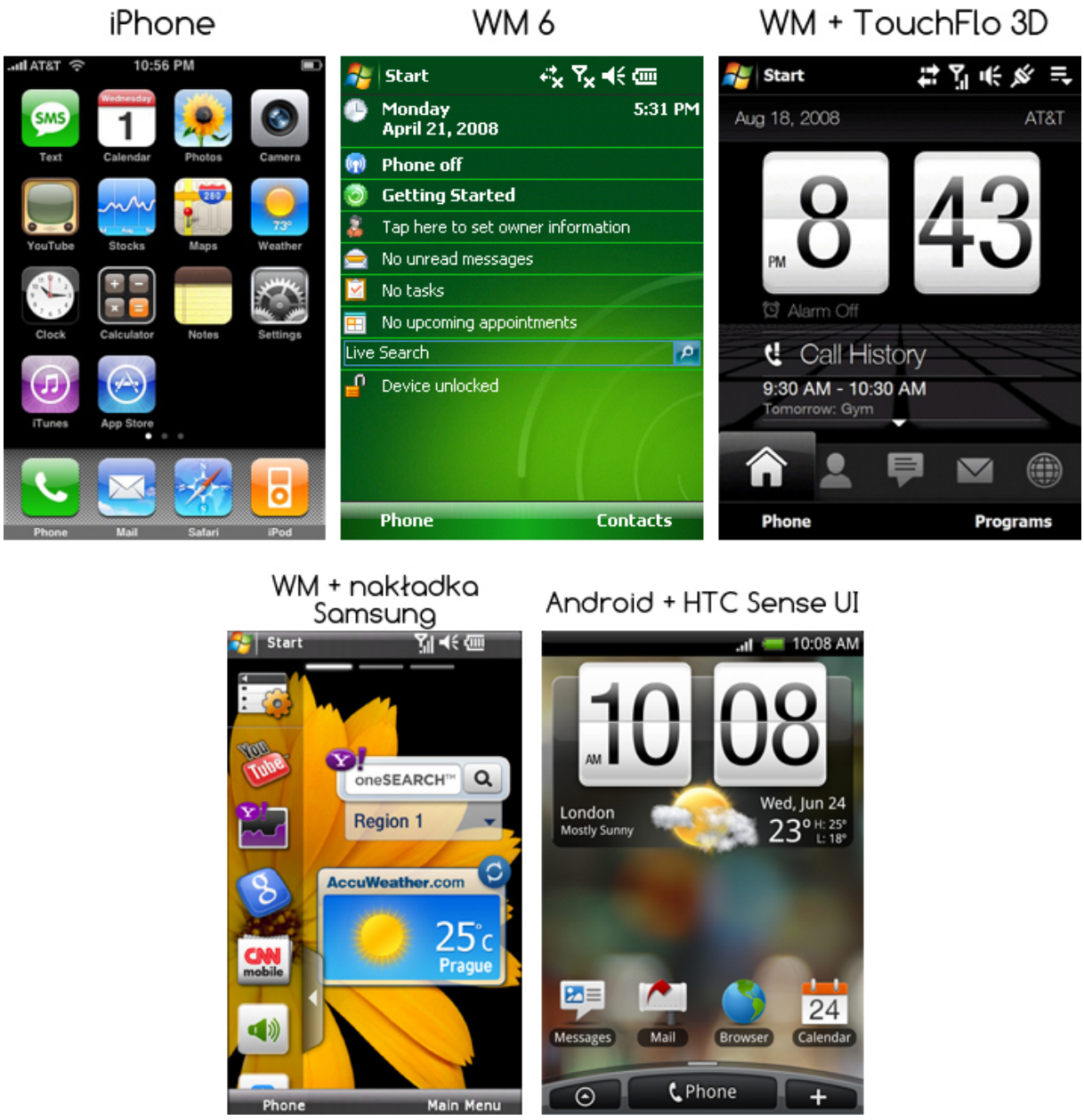
Podczas gdy iPhone robił furorę na rynku, konkurencja pracowała nad odpowiedzią na najnowszy produkt Apple. Zadanie to nie było jednak łatwe. Problem polegał na miażdżącej przewadze systemu iPhone OS nad systemem Windows Mobile, pod którym pracowała większość urządzeń konkurencji. System Microsoft'u w porównaniu do systemu Apple był bardzo powolny, nieresponsywny, miał potężne wycieki pamięci które wymuszały częste resetowanie urządzenia. Stworzenie nowego systemu nie wchodziło w grę, ponieważ trwałoby to zbyt długo. Dlatego najbardziej ceniony producent smartphone'ów, czyli HTC, podjęło się próby zmiany interfejsu użytkownika oraz możliwości standardowego systemu Windows Mobile. Nowe telefony HTC zawierały nakładę graficzną TouchFlo 3D, która całkowicie zmieniała standardowy wygląd systemu. Interfejs graficzny opracowany przez HTC był dostosowany do obsługi palcem, wprowadził obsługę gestów takich jak np. wygodne przewijanie list. Dodatkowo, nowe urządzenia HTC posiadały czujnik przyspieszenia, mocniejszy procesor oraz więcej pamięci, dzięki czemu system zyskał na szybkości a wycieki pamięci stały się mniej dokuczliwe. Tym sposobem HTC zdołało utrzymać a nawet pozyskać nowych zwolenników swoich produktów. Do rywalizacji przystąpił także Samsung, który na początku 2008 roku wprowadził na rynek model Omnia. Smartphone posiadał akcelerometr, procesor 624 MHz i pracował pod systemem Windows Mobile. Podobnie do HTC, Samsung opracował własną nakładkę graficzną na standardowy interfejs użytkownika, obsługującą tzw. widżety ekranowe. Omnia cieszyła się bardzo dobrą opinią wśród użytkowników i sporą popularnością.

System Windows Mobile wciąż nie był w stanie dorównać systemowi iPhone OS. Nowy interfejs użytkownika zmienił jedynie zewnętrzną warstwę systemu, serce nadał pozostawało

to samo a wraz z nim problemy bardzo powolnej pracy i wycieki pamięci. Tak naprawdę niewiele dało się już zrobić z tym systemem. Oczywiście Microsoft co jakiś czas próbował ratować się wypuszczając nowe wydania systemu Windows Mobile 6, oznaczone numerami 6.1 oraz 6.5 . W rzeczywistości, te nowe wersje nic szczególnego nie wnosiły, zawierały głównie drobne poprawki zwiększające wydajność. Pojawiła się jednak alternatywa w postaci nowatorskiego, dobrze rokującego i dynamicznie rozwijającego się systemu o nazwie Android.

1.3.1 System Android

Początkowo system Android był rozwijany przez niewielką firmę w Kalifornii. W 2005 roku firma Google dostrzegła potencjał w tym projekcie i wykupiła całą firmę Android. System był oparty na jądrze Linux a główne biblioteki były napisane w języku C.



Ilustracja 1: Porównanie ekranów głównych w różnych systemach operacyjnych

Wyższe warstwy systemu zostały napisane w języku Java. W tym języku pisane są aplikacje przez programistów. Kod całego systemu był otwarty [15]. Docelową platformą dla systemu Android nie były urządzenia mobilne. System był tworzony z myślą o możliwości instalacji na każdym urządzeniu np. komputerze, telefonie czy nawet telewizorze. W 2007 roku powstał pierwszy zestaw narzędzi dla programistów SDK (ang. *Software Development Kit*). Dynamiczny rozwój projektu sprawił, że pod koniec 2008 roku Google wprowadziło na rynek nowy telefon z systemem Android. Urządzenie to zostało wyprodukowane przez firmę HTC i nosiło nazwę HTC Dream [16]. W tym momencie HTC było już dobrze znane i cenione. Jej bardzo mocną stroną była rozwinięta grupa użytkowników i programistów zrzeszonych na międzynarodowym forum *http://www.forum.xda-developers.com* . Z własnej inicjatywy, podjęli oni nieoficjalną próbę przeniesienia systemu Android na inne, starsze smartphone'y. W końcu to się udało. Wiele urządzeń dotychczasowo pracujących pod systemem Windows Mobile otrzymało nieoficjalne wersje systemu Android. Użytkownicy takich telefonów, szybko skorzystali z okazji i na stałe przenieśli się na system firmy Google. Zmiana ta jeszcze bardziej podkreśliła nieudolność systemu Windows Mobile. Za przykład może posłużyć telefon HTC Kaiser wyprodukowany w 2007 roku z domyślnym systemem Microsoft'u. Po zmianie systemu na Android, użytkownik mógł się cieszyć prawie dwukrotnie szybszą pracą urządzenia!

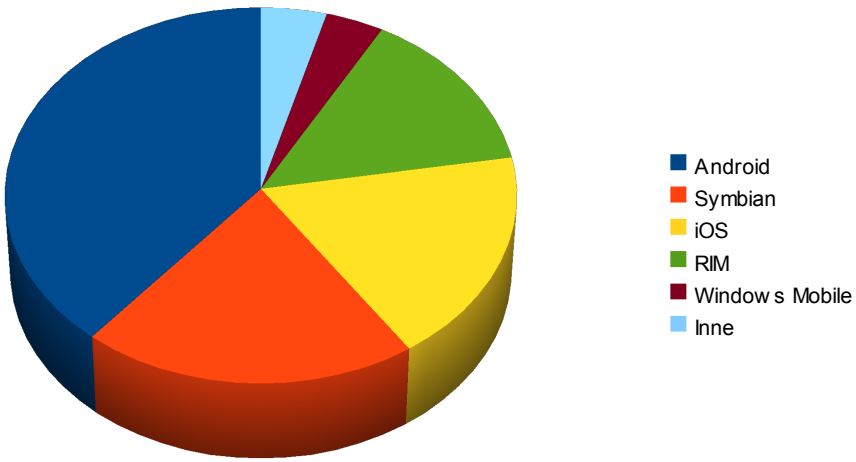
Firma HTC bardzo szybko przeszła na platformę Android wypuszczając szereg nowych urządzeń dostępnych w ofertach operatorów sieci komórkowych. Nie odeszli oni całkowicie od systemu Windows Mobile, wciąż produkują telefony pracujące pod tym systemem, jednak za sprawą programistów z xda-developers każdy telefon HTC posiada swoją wersję systemu Android. Takie smarphone'y mogą jednocześnie mieć zainstalowane oba systemy, a użytkownik może wybrać, którego chce używać w danym momencie.

W Androidzie podobnie jak w iPhone jest market aplikacji, zarówno płatnych jak i darmowych. Nie był on jednak dostępny od początku, został dodany w jednej z kolejnych wersji systemu. Android jest bardzo silnie związany z usługami oferowanymi przez Google. Aby korzystać z systemu, trzeba mieć założone konto w Google. System domyślnie udostępnia zintegrowane z kontem programy takie jak: poczta gmail, kontakty, kalendarz, mapy, google talk, youtube, oraz inne. Wszystkie dane przechowywane są na serwerach

Google i są na stałe powiązane z kontem. Dzięki temu, przy reinstalacji systemu, wszystkie dotychczasowe informacje takie jak kontakty, zostaną automatycznie pobrane. Android jest systemem, który do jego pełnej funkcjonalności wymaga stałego dostępu do internetu. Na szczęście ceny połączeń internetowych są coraz niższe, dzięki czemu już teraz możliwe jest przeglądanie stron internetowych i oglądanie filmików, za stosunkowo nieduże pieniądze.

Android stał się poważną konkurencją dla telefonów iPhone. Google i Apple rywalizują ze sobą w ilości dostępnych w markecie aplikacji. Jak na razie pod tym względem wygrywa iPhone, lecz Android jest już bardzo blisko. Należy jednak mieć na uwadze, że sklep Apple istnieje już od czterech lat, zaś sklep Google połowę krócej. Należy się więc spodziewać, że w najbliższych miesiącach Android wyjdzie na prowadzenie. Oba systemy posiadają wielu zwolenników i przeciwników, walczących między sobą w internecie.

Według ostatnich badań przeprowadzonych przez firmę IDC, system Android stał się najczęściej wybieranym przez użytkowników systemem, przejmując 38.9% rynku. Na drugim miejscu Symbian 20.6% a zaraz za nim iPhone OS z wynikiem 18.2%. Na pozostałych miejscach są RIM 14.2%, Windows Mobile 3.8% i inni [17].



Ilustracja 2: Wyniki badań popularności mobilnych systemów operacyjnych, przeprowadzonych przez firmę IDC w 2011 roku. [17]

System Android został zaprojektowany w sposób niezależny od docelowej platformy. Oznacza to, że jest on gotowy do pracy pod każdym rodzajem urządzeń. To właśnie jest przewagą tego systemu. Już niedługo w życie wejdzie kolejny projekt – Google TV. Powstaną pierwsze telewizory pracujące pod systemem Android. Otworzy to nowy rozdział w rozwoju

telewizji internetowej. Możliwe, że z rynkiem telewizyjnym stanie się to samo co z rynkiem telefonów komórkowych, czyli nastąpi jego nagły rozwój. Być może, za kilka lat, Google TV i telewizja strumieniowa VOD (ang. *Video on Demand*) na dobrze wyprą tradycyjną telewizję kablową. Google stopniowo stara się dotrzeć do każdego rodzaju odbiorcy. Trzeba przyznać, że jak dotąd, świetnie się to udaje.

1.3.2 Wbudowane czujniki pomiarowe otoczenia

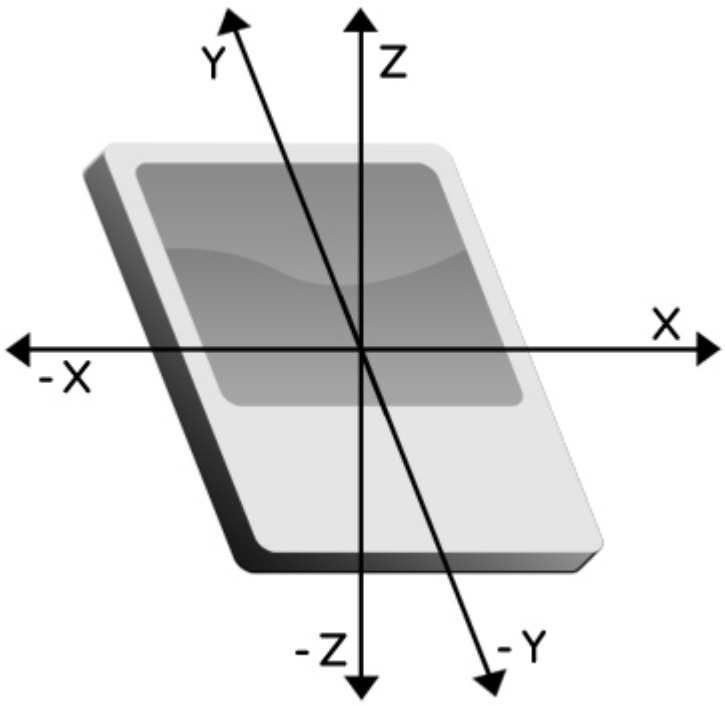
Obecne urządzenia mobilne są bardzo zaawansowane technologicznie. Są wyposażone w szereg różnych urządzeń pomiarowych, które poszerzają wachlarz funkcjonalności oraz możliwości interakcji z telefonem. Pierwszymi wbudowanymi czujnikami były mikrofon oraz kamera. Na dzień dzisiejszy, większość nowych smartphone'ów posiada dodatkowo wbudowane czujniki: przyspieszenia, odległości, światła, orientacji, temperatury, ciśnienia, pola magnetycznego. Programiści mogą z nich korzystać w dowolny sposób, implementując nowe, ciekawe aplikacje. System Android udostępnia jednolite, proste API do obsługi każdego z nich. Dane z czujników są powszechnie wykorzystywane w programach codziennego użytku, oraz w grach. [18]

1.3.2.1 Akcelerometr

Akcelerometr jest czujnikiem, który mierzy przyspieszenia, jakie działają na urządzenie. Wartości te mierzone są w trzech wymiarach i wyrażane są w jednostkach m/s^2 dla każdej z trzech osi X, Y, Z. Gdy urządzenie leży ekranem do góry na płaskim stole, działa na nie tylko siła grawitacji (ok. 9.8 m/s^2). Tak więc na osi Z akcelerometr pokaże wartość -9.8 . Gdyby położyć telefon zwrócony ekranem do blatu stołu, na osi Z pojawiła by się wartość 9.8 . Na każde urządzanie mobilne nie poddane działaniu dodatkowej zewnętrznej siły, znajdujące się w dowolnej pozycji w przestrzeni trójwymiarowej, działa siła wypadkowa równa wartości przyspieszenia ziemskiego. Gdyby urządzenie znajdowało się w stanie spadku swobodnego, akcelerometr pokazałby na każdej z osi wartość 0. Dane z tego czujnika, mogą być odczytywane z bardzo dużą częstotliwością.

Czujnik przyspieszenia jest urządzeniem dosyć dokładnym. Niestety nie w każdym telefonie działa on tak, jak powinien, przez co wskazania na poszczególnych osiach, mogą odbiegać od stanu faktycznego. Za przykład może posłużyć seria tabletów Samsung Galaxy, w której spora część urządzeń posiadała wadę w postaci niedokładnego akcelerometru. Czujnik w tym

urządzeniu zwracał dane obarczone dużym błędem – siła wypadkowa działająca na urządzenie swobodnie leżące na stole, ekranem do góry, wynosiła ok. 11 m/s^2. Co więcej, wartości te różniły się między sobą, w zależności od tego, na której stronie położony był telefon. Takich urządzeń jest więcej. Wielu producentów smartphone'ów wprowadziło możliwość kalibracji akcelerometru, która pozwala skorygować te wskazania, tak aby były one jak najdokładniejsze. Jednak nie we wszystkich telefonach znajdziemy taką opcję. Programiści muszą sobie zdawać sprawę, że taki problem istnieje. Można to częściowo rozwiązać, poprzez zastosowanie własnej kalibracji w swoich programach, poprzez obliczenie wartości siły wypadkowej działającej na urządzenie w stanie spoczynku, oraz wykluczenie jej w późniejszych obliczeniach.



Ilustracja 3: Położenie układu współrzędnych akcelerometru względem urządzenia

Akcelerometr jest jednym z najpopularniejszych czujników. Najlepiej nadaje się do wykrywania gwałtownych zmian w przyśpieszeniu działającym na telefon. Przykładem takiego wykorzystania jest wykrywanie zdarzenia polegającego na potrząśnięciu

urządzeniem. Za pomocą akcelerometru można wykryć także czy telefon jest ułożony w orientacji poziomej czy pionowej oraz czy został rzucony lub upuszczony. Programista może w dowolny sposób zaimplementować obsługę takich zdarzeń, dzięki czemu aplikacja zyskuje na interakcji z użytkownikiem. Standardowo system Android wykrywa położenie w jakim znajduje się telefon, automatycznie dostosowując wyświetlany widok do orientacji poziomej lub pionowej. Często wykorzystuje się akcelerometr w grach jako sterownie. Najlepiej sprawdza się to w grach samochodowych, gdzie możemy prowadzić samochód używając telefonu jak kierownicy.

1.3.2.2 Pozostałe czujniki

- **Czujnik odległości:** wykrywa obecność obiektów w pobliżu telefonu. Zwraca odległość wyrażoną w metrach. Wykorzystywany do wygaszenia i zablokowania ekranu urządzenia w momencie gdy użytkownik rozmawia przez telefon.
- **Czujnik pola magnetycznego:** wykrywa pole magnetyczne, zwracając odpowiednie wartości w przestrzeni trójwymiarowej.
- **Czujnik światła:** wykrywa natężenie światła otoczenia i zwraca wartość wyrażoną w luksach. Wykorzystywane do automatycznej zmiany intensywności podświetlenia ekranu urządzenia.
- **Czujnik orientacji:** jest połączeniem akcelerometru i czujnika pola magnetycznego. Za pomocą akcelerometru obliczany jest kąt nachylenia i obrotu urządzenia. Czujnik pola magnetycznego wyznacza azymut. Czujnik zwraca te trzy wartości wyrażone w stopniach.

Rozdział 2. Geolokalizacja

Geolokalizacja to informacja o położeniu dowolnego obiektu na ziemi. Położenie określane jest za pomocą współrzędnych szerokości i wysokości geograficznej. W kontekście urządzeń mobilnych, geolokalizacja odnosi się do miejsca, w którym znajduje się użytkownik telefonu. Taka informacja wykorzystywana jest w indywidualnym dostosowywaniu treści do klienta. W ostatnich latach informacja o położeniu użytkownika stała się niezwykle cenna. Jednym z najpopularniejszych jej zastosowań jest nawigacja samochodowa i piesza. Oprócz tego

geolokalizację wykorzystuje się np. w internecie, dostosowując treść wyświetlanych przez serwisy reklam, czy też dopasowując wyniki wyszukiwania. Portale społecznościowe wykorzystują ją do stworzenia sieci znajomych z możliwością śledzenia ich aktywności. Istnieją także strony, które w połączeniu z zewnętrznym urządzeniem pozwalają monitorować własną aktywność podczas np. jazdy na rowerze, udostępniając przy tym szereg ciekawych statystyk. Popularne stało się także tzw. geotagowanie zdjęć, czyli przypisanie do zdjęcia informacji o współrzędnych geograficznych miejsca w którym zostało ono wykonane. Funkcja ta jest szczególnie przydatna na różnego rodzaju wycieczkach – dzięki temu nawet jak zapomnimy gdzie jakieś zdjęcie zostało wykonane, szybko to sprawdzimy na mapie. Możliwe jest także zaprezentowanie całej trasy podróży, wraz z listą punktów w których robiliśmy zdjęcia. Możliwości wykorzystania geolokalizacji jest na prawdę wiele.

System Android jest silnie związany z geolokalizacją. Firma Google posiada bardzo obszerną bazę danych przechowującą współrzędne przeróżnych miejsc z całego świata. W Androidzie domyślnie znajduje się aplikacja mapy, która umożliwia nawigację, wyszukiwanie różnych miejsc w najbliższym otoczeniu (np. restauracji), czy też śledzenie znajomych. Oprócz map, jest wiele innych aplikacji społecznościowych opartych na geolokalizacji. Android potrafi uzyskać pozycję na trzy różne sposoby: wykorzystując dane z modułu GSM, na podstawie adresu IP oraz odczytując ją bezpośrednio z odbiornika GPS (ang. *Global Positioning System*).

2.1 Geolokalizacja GSM

Informację o aktualnym położeniu można uzyskać w każdym telefonie. Warunkiem jest posiadanie aktywnej karty SIM oraz dostęp do internetu – GPRS lub WiFi. Do geolokalizacji GSM wykorzystuje się informacje pobierane bezpośrednio z sieci komórkowej, do której jesteśmy aktualnie zalogowani. Każdy kraj, w którym istnieje sieć komórkowa, posiada swój unikalny numer identyfikacyjny MCC (ang. *Mobile Country Code*) [19]. Wszystkie sieci komórkowe posiadają swój unikalny w obrębie kraju numer identyfikacyjny MNC (ang. *Mobile Network Codego* [20]. Numer MNC wraz z numerem MCC jednoznacznie identyfikują kraj abonenta oraz sieć do której jest podłączony. Każda sieć komórkowa jest podzielona na obszary LA (ang. *Location Area*). Obszary posiadają swój unikalny w obrębie sieci komórkowej numer identyfikacyjny LAC (ang. *Location Area Code*) [21]. W zakresie jednego obszaru LA istnieje od kilkudziesięciu do kilkuset wież nadawczych, których obszar

zasięgu tworzy tzw. komórkę. Każda komórka posiada swój unikalny w obrębie obszaru LA numer identyfikacyjny CID (ang. *Cell Identifier*) [22]. Aby jednoznacznie zidentyfikować telefon, potrzebne są numery MCC, MNC, LAC oraz CID. Może je odczytać dowolny użytkownik zalogowany do sieci. Pierwszym krokiem w geolokalizacji GSM jest więc odczytanie tych numerów z modułu GSM. Numery same w sobie nic nie mówią o pozycji, w której znajduje się abonent. Aby uzyskać pozycję, należy je przetłumaczyć na wysokość i szerokość geograficzną. Oczywiście, każdy operator zna współrzędne, w których znajdują się jego wieże nadawcze, lecz zwykły użytkownik nie ma do nich dostępu. Z pomocą przychodzi Google, które posiada bazę danych ze współrzędnymi geograficznymi wszystkich wież nadawczych na świecie. Pod adresem *http://www.google.com/glm/mmap* znajduje się serwer, z którego można odczytać pozycję. Wystarczy wysłać pod ten adres zapytanie POST przesyłając numery MCC, MNC, LAC oraz CID a w odpowiedzi otrzymamy wysokość i szerokość geograficzną.

Geolokalizacja GSM jest mało znaną techniką pozyskiwania informacji o pozycji abonenta, szczególnie wśród użytkowników telefonów komórkowych. Mało kto wie, że operatorzy sieci w każdej chwili mają dostęp do informacji o naszym położeniu. Niejednokrotnie współpraca policji z operatorami sieci komórkowych sprawiła, że geolokalizacja GSM przyczyniła się do złapania przestępców, odnalezienia osób porwanych bądź zaginionych.

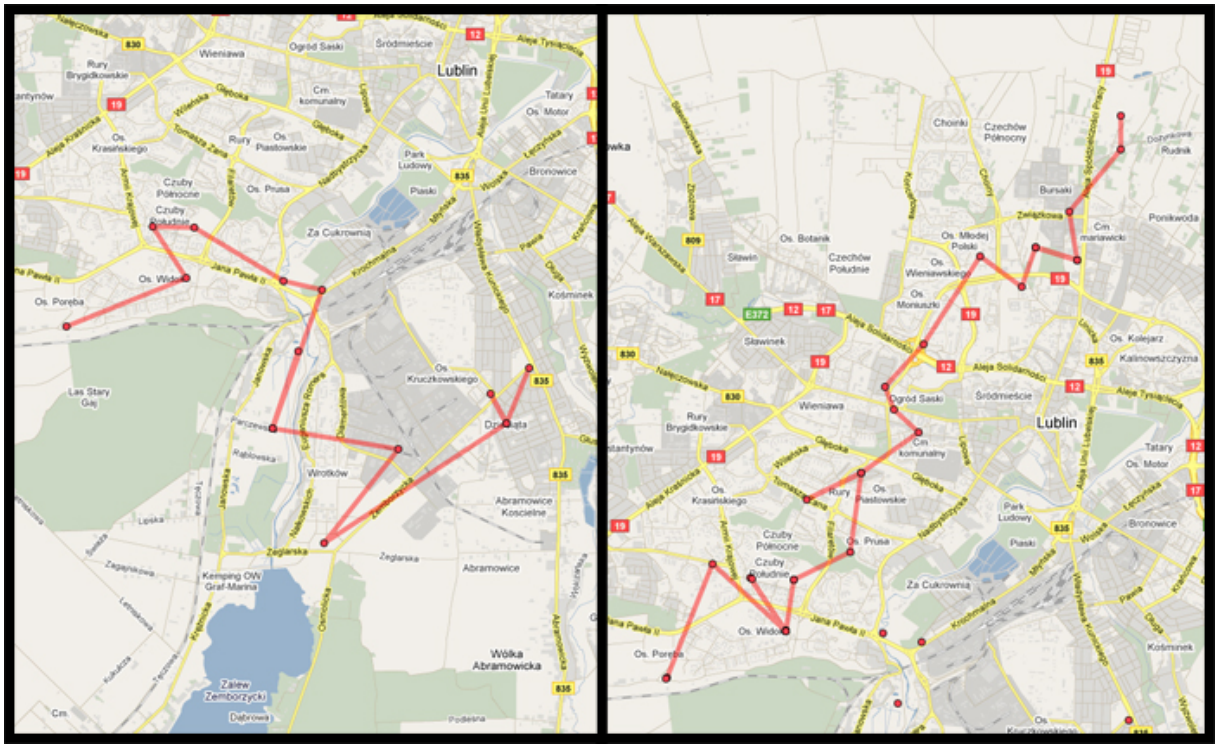
2.1.1 Zalety geolokalizacji GSM

Geolokalizacja GSM posiada kilka istotnych zalet, które sprawiają, że jest ona dosyć interesującą metodą pozyskiwania informacji o pozycji. Przede wszystkim nie są potrzebne dodatkowe zewnętrzne urządzenia, wystarczy zwykły telefon z dostępem do internetu oraz odpowiednim programem. Drugą zaletą jest niewielki pobór prądu przez moduł GSM. Jest to istotna sprawa z punktu widzenia urządzeń mobilnych takich jak telefony, które posiadają skończone rezerwy energii. Telefon może non stop zbierać informacje o położeniu, nie powodując przy tym szybszego wyczerpywania się baterii. Ostatnim plusem tej metody, jest to, że działa ona wszędzie tam, gdzie jest zasięg.

2.1.2 Wady geolokalizacji GSM

Główną wadą tej metody jest mała dokładność. Jest ona uzależniona od ilości wież nadawczych w naszym otoczeniu. Naturalnie w miastach jest ich więcej niż na wsiach.

Dokładność w miastach zazwyczaj mieści się w przedziale od czterystu metrów do nawet dwóch kilometrów. Dokładność na obszarach wiejskich jest dużo mniejsza i jest wyrażana w kilometrach. Drugim minusem jest wymagane połączenie z internetem, a co za tym idzie koszty, które użytkownik musi ponieść. Jeżeli pozycja nie jest nam potrzebna w czasie rzeczywistym, możemy ten problem rozwiązać poprzez zapisywanie wszystkich numerów identyfikacyjnych gdzieś w pamięci podręcznej, a samo tłumaczenie numerów na współrzędne geograficzne wykonać po zgraniu danych na komputer lub odłożyć do czasu gdy podłączymy się do sieci WiFi. Ostatnim minusem, a raczej ograniczeniem, jest uzyskanie pozycji w przestrzeni dwuwymiarowej – nie można uzyskać informacji o wysokości, na której znajduje się użytkownik.



Ilustracja 4: Przykładowe trasy zarejestrowane z wykorzystaniem geolokalizacji GSM. Pomiary dokonane przez autora pracy.

2.2 Geolokalizacja GPS

Obecnie większość smartphone'ów jest dodatkowo wyposażona w moduł GPS. Udostępnia on informacje o aktualnej pozycji, prędkości oraz czasie. Odczytywanie współrzędnych z odbiornika GPS, jest jak dotąd, najdokładniejszą metodą geolokalizacji. System GPS został zaprojektowany na potrzeby wojska. Jest on zarządzany przez rząd Stanów Zjednoczonych.

W 1993 roku system GPS został udostępniony cywilom na całym świecie. [23]

System GPS ma szereg różnych zastosowań. Najczęściej jest używany do nawigacji pieszej i samochodowej. Już dawno wyparł tradycyjne mapy papierowe. Gdy pojawiły się pierwsze smartphone'y programiści zaczęli pisać nowe programy w nowatorski sposób wykorzystujące ten system. Powstały aplikacje do rejestracji tras wycieczek, geotagowania zdjęć, śledzenia aktywności znajomych, monitorowania dzieci. Można go wykorzystać także jako licznik samochodowy lub rowerowy, oraz do sporządzania różnych zestawień danych, takich jak np. profil wysokościowy przebytej trasy. Ostatnio popularne stało się wykorzystanie systemu GPS w połączeniu z czujnikami orientacji, akcelerometru oraz kamery, jako mobilnego przewodnika. Turysta znajdujący się w obcym mieście, posiadający urządzenie z odpowiednią aplikacją, może włączyć aplikację i skierować telefon w stronę jakiegoś budynku. Na ekranie widać będzie ten obiekt wraz ze stosownym opisem. Obracając się, na ekranie będą widoczne coraz to inne miejsca i budynki, wszystkie z własną charakterystyką.

2.2.1 Opis działania systemu GPS

System GPS udostępnia informacje o aktualnej pozycji, prędkości oraz czasie. Jest dostępny na całym świecie, lecz zwykli użytkownicy maja narzucone pewne ograniczenia, zmniejszające częstotliwość odczytu z satelitów oraz mniejszą dokładność uzyskanych danych. System jest podzielony na trzy segmenty: kosmiczny, naziemny oraz użytkowy. [24]

2.2.1.1 Segment kosmiczny

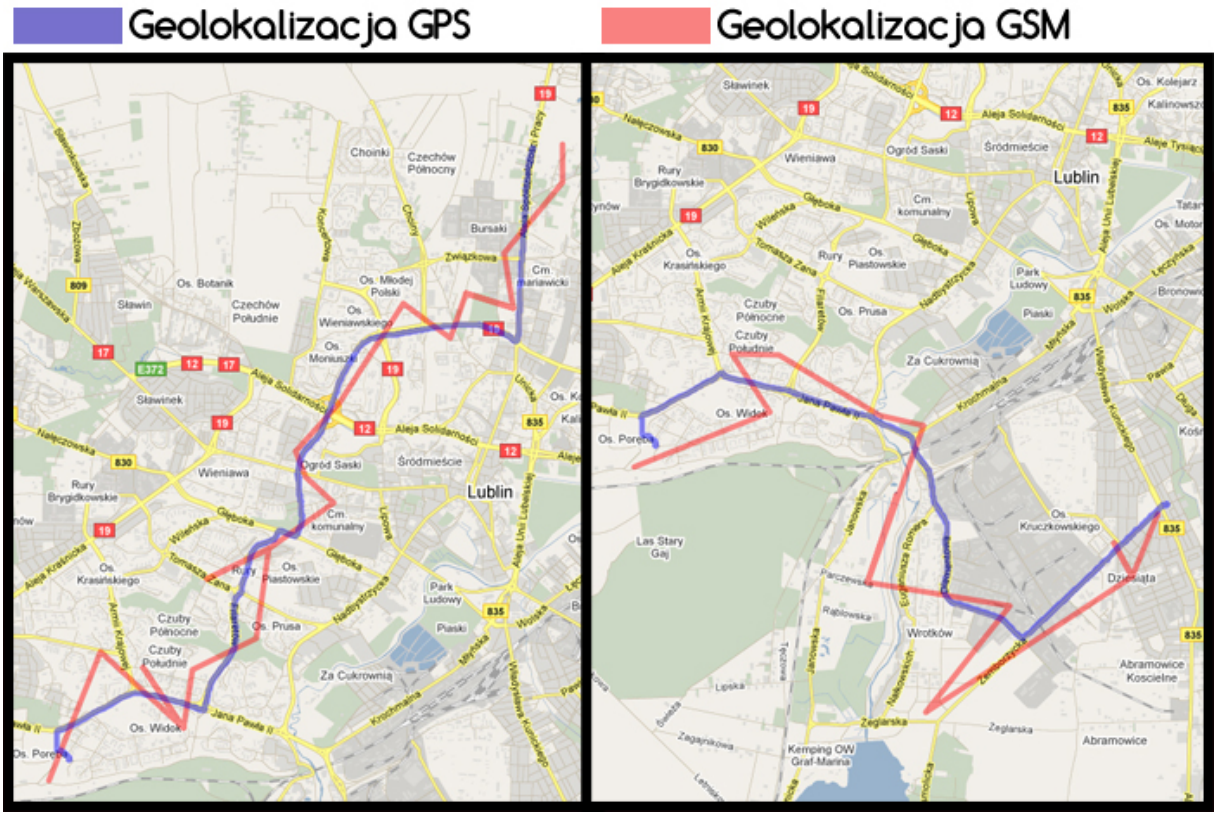
Składa się z 24 satelitów, bez przerwy nadających sygnał, oraz kilku dodatkowych, wykorzystywanych w celach testowych. Obecnie łącznie jest ich 31. Każdy satelita posiada swój własny zegar atomowy. Cały system opiera się o dokładność czasu, która jest niezmiernie istotna w prawidłowym wyznaczaniu pozycji odbiorcy sygnału. Satelity bez przerwy wysyłają sygnał zawierający informację o czasie nadania, pobraną z własnego zegara. Satelity nadają na dwóch częstotliwościach, z czego jedna jest zastrzeżona tylko dla wojska, a druga jest udostępniona cywilom.

2.2.1.2 Segment użytkowy

Składa się z użytkowników odbierających sygnał z satelitów. Użytkownik odbierający taki sygnał jest w stanie policzyć czas jego transmisji. Mając taką informację, oraz wiedząc, że

fala elektromagnetyczna rozchodzi się z prędkością światła (ok. 299 792 458 m/s), można obliczyć odległość pomiędzy odbiorcą a satelitą. Odbiornik oblicza własną pozycję korzystając z sygnałów otrzymywanych jednocześnie z kilku satelitów. Aby uzyskać informację o wysokości oraz współrzędnych geograficznych, odbiornik musi odbierać sygnał z co najmniej trzech satelitów. Większa ilość satelitów, co najmniej cztery, umożliwia uzyskanie pozycji w przestrzeni czterowymiarowej, gdzie czwartym wymiarem jest czas. Pozycja uzyskana z odbiornika GPS jest bardzo często nazywana fix'em.

Czas oczekiwania na fix z odbiornika, który przez dłuższy czas nie był używany, może trwać nawet kilka minut. Jest to tak zwany zimny start. Oznacza to, że odbiornik nie posiada aktualnych danych o położeniu satelitów, musi więc przeszukiwać przestrzeń w poszukiwaniu sygnałów. Oprócz zimnego startu, jest także ciepły oraz gorący start. Ciepły start odnosi się do sytuacji, w której większość danych o położeniu satelitów jest aktualna, wtedy czas oczekiwania na fix wynosi zazwyczaj nie dłużej niż 30 sekund. Gorący start ma miejsce w sytuacji gdy urządzenie było nieaktywne tylko przez kilka minut, dzięki czemu wszystkie dane o położeniu satelitów są aktualne, fix uzyskuje się w ciągu kilku sekund.



Ilustracja 5: Porównanie geolokalizacji GPS (kolor niebieski) oraz GSM (kolor czerwony). Pomiary dokonane przez autora pracy.

2.2.1.3 Segment naziemny

Składa się z kilku stacji monitorujących i korygujących działanie każdego z satelitów. Stałej korekcji podlegają zegary atomowe satelitów. Korekta jest niezbędna, gdyż czas odgrywa kluczową rolę w całym systemie. Dlaczego jednak zegary atomowe wymagają korekty? Okazuje się, że satelity krążące po orbicie ziemi, podróżują w czasie. Ich czas spowalnia wskutek oddziaływania pola grawitacyjnego Ziemi. Zegary każdego z satelitów gubią co trzy dni 0,000000001 sekundy. Taka różnica skutkuje jedno-metrowym błędem w wyznaczaniu położenia odbiorcy.

2.2.2 Dokładność systemu GPS

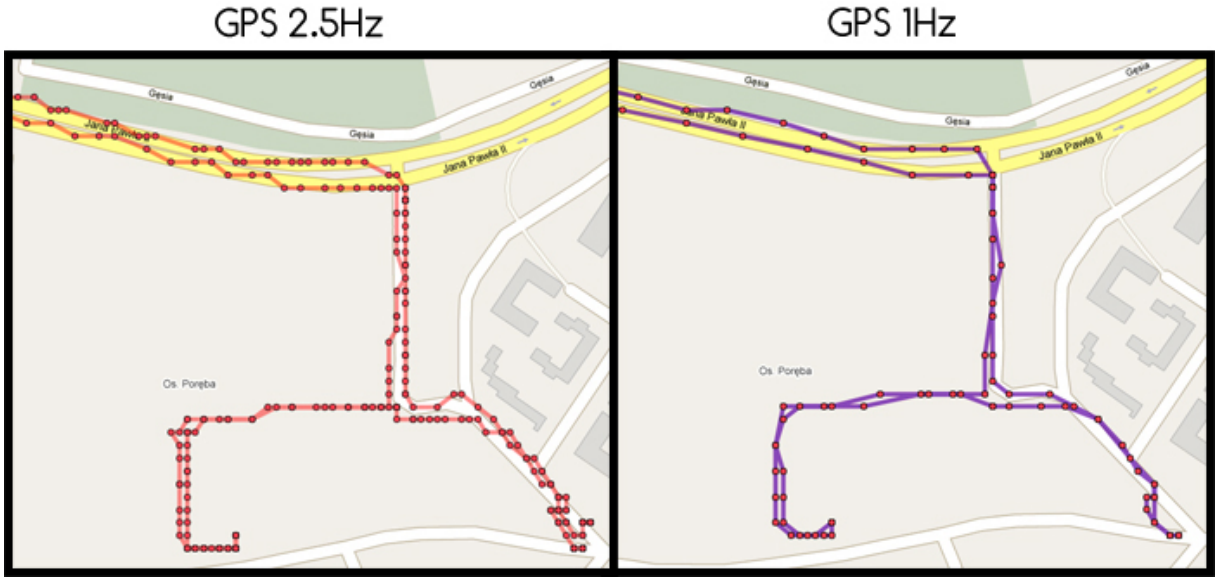
System GPS oferuje dwa poziomy dokładności: PPS (ang. *Precise Positioning System*) wykorzystywany w celach militarnych, oraz SPS (ang. *Standard Positioning System*) udostępniony bez żadnych opłat wszystkim cywilom na całym świecie. PPS określa pozycję w przestrzeni trójwymiarowej z błędem nie większym niż 16 metrów, oraz dostarcza informację o czasie z dokładnością do 100 nanosekund. SPS jest dużo mniej precyzyjny niż PPS. Dostarcza pozycję w przestrzeni trójwymiarowej z błędem nie większym niż 100 metrów, oraz informację o czasie z dokładnością do 337 nanosekund. Zazwyczaj dobre odbiorniki GPS wskazują pozycję w przestrzeni trójwymiarowej z dokładnością lepszą niż 40 metrów. System GPS dostarcza także informację o prędkości. Odbiorniki obliczają ją wykorzystując efekt Dopplera. W PPS można uzyskać prędkość z dokładnością do 0.2 m/s na każdej z osi, zaś w SPS oczekiwany błąd pomiaru wynosi 0.4 m/s. Prędkość można także obliczyć znając dwie współrzędne oraz różnicę czasu pomiędzy ich odczytami. Nie jest to jednak sposób dokładny, ze względu na częste przekłamania w pozycji. Prędkość obliczona przez odbiornik wykorzystujący do tego zjawisko Dopplera, jest zazwyczaj wartością dokładniejszą od obliczonej pozycji. [24]

Oprócz technicznych ograniczeń systemu GPS, wpływ na dokładność geolokalizacji ma częstotliwość próbkowania sygnału z satelitów. Większość modułów GPS wbudowanych w smartphone'y obsługuje częstotliwość odczytu 1Hz. Oznacza to, że pozycja jest uzyskiwana raz na sekundę. Częstotliwość ta jest w większości przypadków wystarczająca, umożliwia sprawną nawigację samochodową. Nie jest ona jednak wystarczająca do śledzenia dynamicznych aktywności sportowych takich jak np. wyścigi samochodowe, czy kolarstwo

górskie. W takim przypadku można kupić zewnętrzny odbiornik GPS, komunikujący się z telefonem przez Bluetooth. Bez problemu można znaleźć moduł pracujący z częstotliwością odczytu 5Hz, która znacząco zwiększa dokładność pozyskiwanych danych. Istnieją także odbiorniki obsługujące częstotliwości 10Hz czy nawet 20Hz.

Odbiorniki GPS mogą funkcjonować jedynie tam, gdzie jest silny sygnał z satelitów, najlepiej w miejscach z bezpośrednim dostępem do nieba. Fix'a nie uda się złapać wewnątrz budynków, tunelach czy metrze. Każda ściana, czy nawet dach samochodu, osłabiają sygnał docierający do odbiornika GPS, powiększając w konsekwencji błędy w określaniu pozycji. Nawet zła pogoda często pogarsza działanie całego systemu.

Pierwsze kilka współrzędnych odczytanych zaraz po zimnym starcie, jest zazwyczaj dosyć niedokładnych. Im większa liczba aktywnie śledzonych satelitów, tym dokładniejsza pozycja.



Ilustracja 6: Porównanie częstotliwości odczytu z GPS. Po lewej stronie odbiornik 2.5Hz, zarejestrował 156 punktów. Po prawej odbiornik 1Hz, zarejestrował 71 punktów. Pomiary dokonane przez autora pracy.

2.2.3 Geolokalizacja A-GPS

Czas oczekiwania na pierwszy fix może trać nawet kilka minut. W urządzeniach jednocześnie posiadających moduł GSM oraz GPS, istnieje możliwość skrócenia tego czasu. Odbiornik GPS jest wtedy wspomagany danymi z modułu GSM. Takie połączenie nazywa się A-GPS

(ang. *Assisted GPS*). Dane te zawierają między innymi informację o położeniu satelitów dostępnych w polu odbiornika GPS. Taka współpraca obu modułów znacząco skraca czas zimnego startu. A-GPS nie działa, jeżeli operator nie włączył tej funkcji. Obecnie większość operatorów sieci komórkowych jak i odbiorników GPS obsługuje A-GPS.

2.2.4 Zalety

Podstawową zaletą systemu GPS jest duża dokładność w określaniu pozycji w przestrzeni czterowymiarowej. Jest to obecnie najdokładniejsza metoda geolokalizacji.

2.2.5 Wady

Główną wadą systemu GPS jest to, że nie da się go używać wewnątrz budynków oraz pod ziemią. Uzyskiwane pozycje często potrafią być niedokładne. Występuje także problem ze śledzeniem satelitów w złych warunkach pogodowych, lub w centrach dużych miast, pośród wysokich budynków. Kolejną wadą, istotną z punktu widzenia urządzeń mobilnych, jest wysokie zużycie energii. Większość smartphone'ów nie jest w stanie pracować dłużej niż dwie godziny z aktywnie włączonym modułem GPS.

2.3 Geolokalizacja WiFi

Geolokalizacja WiFi stała się ostatnio bardzo popularnym sposobem pozyskiwania informacji o położeniu użytkownika. Jest to trzecia i zarazem ostatnia metoda geolokalizacji dostępna obecnie w systemie Android. Projekt został zaplanowany na bardzo dużą skalę, i pomimo, że jego rozwój wciąż trwa, już teraz odnosi duże sukcesy. Polega on na tworzeniu gigantycznej bazy danych zawierającej listę wszystkich znajdujących się na świecie punktów dostępowych do sieci WiFi, wraz ze współrzędnymi geograficznymi ich występowania oraz informacją o sile sygnału. Projekt jest realizowany przez kilka firm, np. Google czy Skyhook. Zatrudniają one pracowników, których zadaniem jest jazda samochodem po świecie. Są oni wyposażeni w specjalne urządzenie, które automatycznie wykrywa wszystkie sieci WiFi i dodaje je do bazy danych. Zadaniem kierowcy jest jedynie dokładne odwiedzanie wszystkich zakamarków dużych miast. Takim pracownikiem może zostać praktycznie każdy, wystarczy posiadać swój samochód, ważne prawo jazdy oraz ubezpieczenie. Aplikację o pracę składa się drogą internetową. Baza danych tworzona jest począwszy od największych miast w najbardziej rozwiniętych krajach świata. Obecnie jest to dosyć mocno rozwinięta sieć, pokrywająca

większość terytorium Ameryki Północnej, Europy zachodniej (w tym Polskę i jej najważniejsze miasta) oraz Japonię. Każdy obszar będący w zasięgu sieci, jest co jakiś czas ponownie skanowany, dzięki czemu baza danych jest ciągle aktualna. [25]

2.3.1 Dokładność geolokalizacji WiFi

Geolokalizacja WiFi jest zaskakująco skuteczną metodą uzyskiwania informacji o położeniu. Pozwala ona określić pozycję użytkownika z dokładnością mieszczącą się w przedziale od 10 do 60 metrów. Współrzędne geograficzne są przybliżane na podstawie siły sygnału sieci bezprzewodowej, w obrębie której znajduje się użytkownik. Najlepiej sprawdza się ona w miastach.

2.3.2 Zalety

Największą zaletą geolokalizacji WiFi jest to, że działa ona wszędzie tam, gdzie zawodzi GPS, udostępniając przy tym dokładne współrzędne geograficzne użytkownika. Sprawuje się ona świetnie w centrach dużych miast, wewnątrz budynków, wśród wysokich budowli gdzie zwykły GPS często gubi sygnał, w metrze oraz w wielu innych miejscach.

2.3.3 Wady

Zasięg działania tej metody ogranicza się tylko do obszaru większych miast. Informację o położeniu można uzyskać jedynie po zalogowaniu się do sieci bezprzewodowej. Niezbędna jest więc obecność punktów dostępowych sieci WiFi, których nie znajdziemy będąc w trasie lub poza granicami miast. Minusem jest także wysokie zużycie energii przez bezprzewodową kartę sieciową. Aby geolokalizacja była możliwa, w telefonie musi być włączone WiFi, które może szybko rozładować baterię w naszym urządzeniu.

Rozdział 3. Aplikacja AutoAndroid

Część praktyczną tej pracy, stanowi aplikacja AutoAndroid. Została napisana w języku Java na platformę mobilną Android w wersji 2.2, wykorzystując API systemu w wersji 8. Głównym zadaniem programu, jest monitorowanie i gromadzenie danych dotyczących stanu

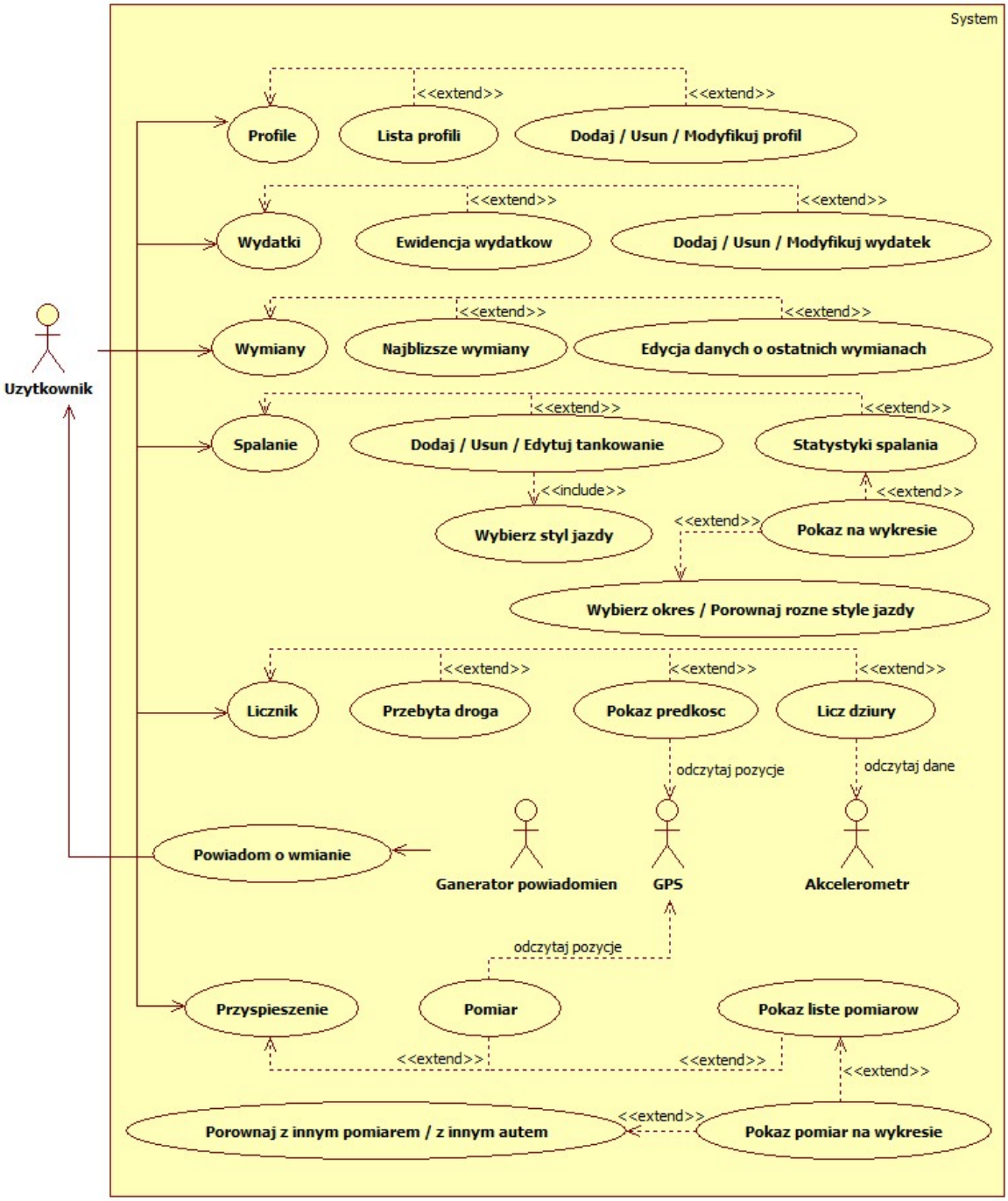
pojazdu, udostępniając na ich podstawie szereg praktycznych informacji statystycznych i zestawień, dzięki którym użytkownik lepiej pozna swój samochód. Aplikacja wykorzystuje akcelerometr oraz geolokalizację GPS. Wszystkie dane przechowywane są w bazie danych SQLite.

3.1 Funkcjonalność aplikacji

Funkcje programu można podzielić na kategorie główne:

- rutynowe wymiany płynów i elementów
- ewidencja wydatków stałych i ruchomych
- raporty spalania
- pomiar przyśpieszenia
- licznik samochodowy

Aplikacja pozwala zdefiniować kilka różnych profili samochodowych, dzięki czemu każdy z nich jest oddzielnie monitorowany. Wszystkie zdefiniowane w programie pojazdy są rozróżniane na postawie unikalnego numeru identyfikacyjnego. Umożliwia to osobne śledzenie przeznaczonych wydatków, osiągów bądź spalań kilku samochodów. Jest to szczególnie przydatne w rodzinie posiadamy kilka samochodów. Wszystkie funkcjonalności zostaną dokładniej opisane w kolejnych podrozdziałach.



Ilustracja 7: Diagram przypadków użycia aplikacji AutoAndroid

3.1.1 Profile

W programie można zdefiniować kilka profili samochodowych. Aby móc korzystać z aplikacji, musi być utworzony co najmniej jeden. Przy pierwszym uruchomieniu programu, zostanie wyświetlony formularz, w którym można go utworzyć. Profil przechowuje

informacje o samochodzie: marka, model, pojemność silnika, moc silnika, rocznik produkcji, przebieg całkowity oraz rodzaj paliwa. W każdej chwili możliwa jest zmiana aktywnego profilu oraz modyfikacja jego danych.

3.1.2 Wydatki

Użytkownik może prowadzić ewidencję wydatków przeznaczonych na samochód. Możliwe jest ich dodawanie, usuwanie oraz modyfikowanie. Dodając informację o wydatku należy podać: nazwę wydatku, koszt, kategorię, datę oraz opcjonalny opis. Domyślnie wbudowane są cztery kategorie wydatków: paliwo, naprawy, kosmetyka, inne. Program oblicza dane statystyczne: całkowite wydatki, wydatki w obecnym miesiącu, wydatki w obecnym roku, średnie miesięczne wydatki, średnie roczne wydatki. Oprócz tego obliczana jest suma wszystkich wydatków z poszczególnych kategorii.

3.1.3 Wymiany

Program umożliwia automatyczne powiadamianie o rutynowych czynnościach takich jak: wymiana oleju, rozrządu, przeglądu technicznego oraz ubezpieczenia. Aby powiadomienia były aktywne, użytkownik musi podać dla każdej pozycji z osobna, informację o częstotliwości wymian (co ile kilometrów i/lub co ile miesięcy) oraz o dacie ostatniej wymiany. Na tej podstawie, program oblicza ilość miesięcy oraz kilometrów, pozostałych do wymiany każdej z wyżej wymienionych rzeczy. Ilość kilometrów obliczana jest oparciu o raporty spalania. Gdy nadejdzie termin wymiany, system Android wygeneruje powiadomienie informując o tym użytkownika.

3.1.4 Spalanie

Użytkownik może dodawać, edytować oraz usuwać raporty spalania. Każdy raport składa się z informacji: ilość zatankowanych litrów paliwa, ilość przejechanych kilometrów, styl jazdy, data tankowania. Aby dane były dokładne, użytkownik powinien zawsze tankować do pełna. Dzięki temu, będzie miał dokładną informację o ilości litrów paliwa, które spalił na przejechanym dystansie. Raporty spalania są wyświetlane w formie listy. Każda pozycja wyświetla informację o uzyskanym średnim spalaniu na sto kilometrów. Oprócz tego udostępnione są dane statystyczne: minimalne średnie spalanie, maksymalne średnie spalanie, średnie spalanie ogólne, średnie spalanie przy stylu ekonomicznym, średnie spalanie przy stylu normalnym, średnie spalanie przy stylu dynamicznym, średnie spalanie w lecie, średnie

spalanie w zimie. Raporty spalania można obejrzeć także w formie wykresu. Można na nim wyświetlić i porównać uzyskane spalania przy różnych stylach jazdy: ekonomicznym, normalnym oraz dynamicznym.

3.1.5 Licznik

Licznik samochodowy wyświetla graficzny licznik, pokazujący aktualną prędkość auta. Obliczana jest także przebyta odległość oraz ilość dziur w przebytej drodze. Istnieje także możliwość zmiany skali graficznego licznika samochodowego, tak aby pasowała ona do prawdziwego licznika w konkretnym aucie.

3.1.6 Przyspieszenie

Użytkownik programu może zmierzyć przyspieszenie swojego samochodu podczas rozpędzania się od 0 do 100 km/h. Pomiar jest wykonywany automatycznie. Oznacza to, że program sam wykrywa moment ruszenia samochodu, zapamiętując przy tym czas startu i aktywuje pomiar. W momencie gdy samochód przekroczy prędkość 100 km/h, pomiar zostaje przerwany, a użytkownik otrzymuje informację o uzyskanym czasie. Rejestracja przyspieszenia rozpocznie się tylko wtedy, gdy kierowca się wcześniej zatrzyma. Jeżeli rejestracja pomiaru trwa, lecz użytkownik nie przekroczy granicy 100 km/h, a zamiast tego, zatrzyma się, wtedy pomiar zostanie przerwany, a jego dane zostaną porzucone. Migająca czerwona dioda oznacza, że program nagrywa przejazd.

Program tworzy dwie listy zarejestrowanych pomiarów przyspieszenia: trzy najlepsze uzyskane czasy oraz wszystkie zarejestrowane przejazdy. Każda pozycja zawiera informację o uzyskanym czasie oraz o dokładności pomiaru. Pojedynczy przejazd można wyświetlić na wykresach: prędkości od czasu oraz drogi od czasu. Na wykresach można jednocześnie wyświetlić i porównać kilka pomiarów, w tym również pomiary dokonane dla innych profili samochodowych.

3.2 Implementacja

W tym podrozdziale zostanie omówiony sposób implementacji wszystkich funkcjonalności. W dalszej części będę używał pojęć specyficznych dla programowania na platformie Android. Oto ich krótki opis: [26]

Widok – tak nazywane są podstawowe elementy interfejsu użytkownika, takie jak: przyciski, listy, etykiety, itp.. Widokiem jest również odpowiednio pogrupowany zbiór tych elementów, tworzący docelowo prezentowany formularz.

Aktywność – jest zazwyczaj reprezentacją pojedynczego ekranu aplikacji, wyświetlanego użytkownikowi. W aktywności zawarty jest co najmniej jeden widok. Występują także aktywności bez widoku, są nimi najczęściej usługi działające w tle.

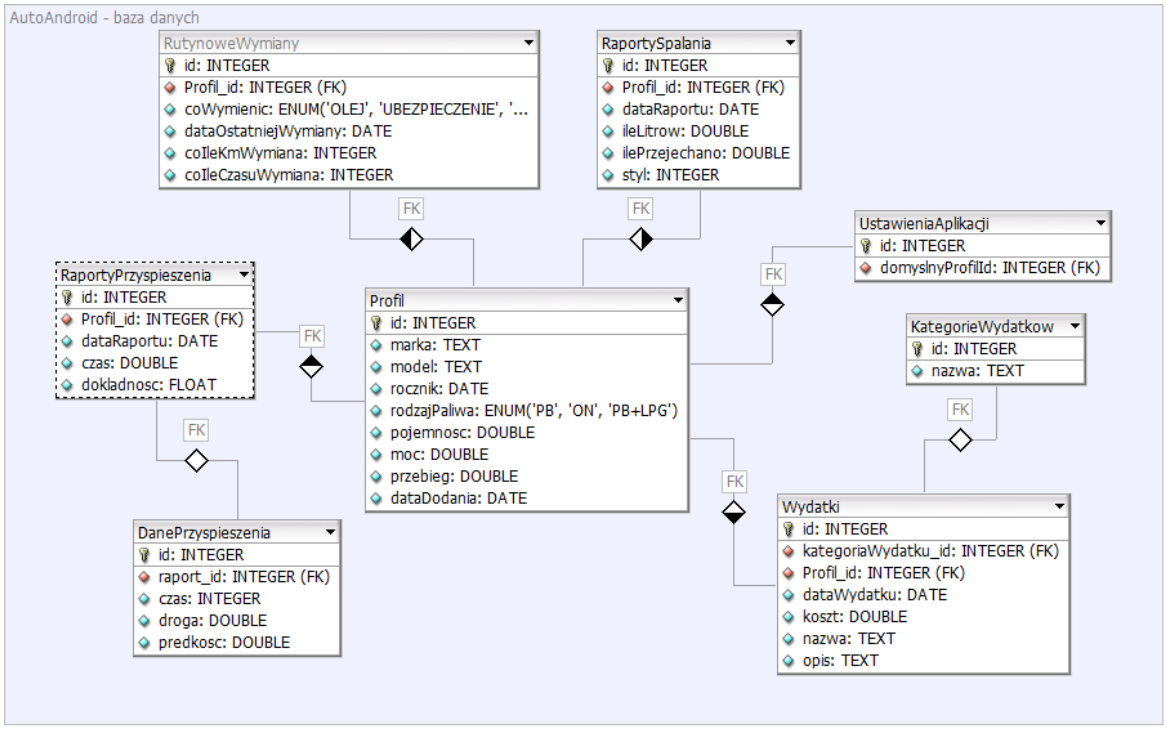
Intencja – jest to pojęcie dosyć trudne do zdefiniowania. Ogólnie wyraża ona jakiś „zamiar". Aplikacja może reagować na zamiary np. użytkownik ma zamiar otworzyć nasz program, musi więc ona na tę intencję zareagować, wyświetlając odpowiednią aktywność początkową. Także sama aplikacja może tworzyć intencję, np. gdy użytkownik kliknie jakiś przycisk który ma go przenieść do innego okna, wtedy aplikacja ma zamiar wyświetlić nową aktywność a do tego potrzebna jest intencja.

Manifest – jest plikiem definiującym zachowania aplikacji, jej części składowych takich jak aktywności oraz intencje, oraz listę uprawień które musi posiadać program do jego uruchomienia.

Cykl życia aplikacji – system Android ściśle kontroluje stany w których znajduje się każda aplikacja. Programy są tworzone, uruchamiane, wstrzymywane, wznawiane, stopowane oraz usuwane. Każdy proces, w trakcie swojego życia, przechodzi pomiędzy tymi stanami. Programista musi zaimplementować zachowanie aplikacji w poszczególnych stanach. Przykładowo, w trakcie tworzenia procesu, wywoływana jest metoda *onCreate()* w której najczęściej definiuje się i wyświetla wygląd aktywności.

3.2.1 Baza danych

Wszystkie dane aplikacji są zapamiętywane w bazie danych SQLite. Każdy program w systemie Android ma dostęp do swojej własnej bazy danych, oraz do baz tych aplikacji, które umożliwiają taki dostęp. Baza danych w programie AutoAndroid jest niepubliczna, dostępna jedynie dla jej właściciela. Poniższy diagram prezentuje logiczną strukturę bazy danych użytej w programie:



Ilustracja 8: Struktura bazy danych programu AutoAndroid.

Do obsługi bazy danych została zaimplementowana klasa pomocnicza *BazaDanych*, której implementacja jest zamieszczona na poniższym listingu. Dziedziczy ona po klasie *SQLiteOpenHelper* oraz implementuje dwie główne metody: *onCreate()* oraz *onUpgrade()*. W pierwsza z nich, wywoływana jest zawsze wtedy, gdy aplikacja próbuje uzyskać dostęp do bazy, lecz jest ona niezainicjowana. W ciele metody wykonywany jest szereg poleceń SQL tworzący wszystkie potrzebne tabele. Metoda *onUpgrade()* wywoływana jest, gdy zmieni się wersja bazy danych, przechowywana w polu *DB_VERSION*. Obecna implementacja tej metody, kasuje dotychczasowe tabele i wywołuje metodę *onCreate()* w której zostaną one na nowo utworzone według nowego schematu. Do tak utworzonej bazy danych można uzyskać uchwyt wywołując na jej obiekcie odziedziczoną metodę *getWritableDatabase()*.

Listing 1: Implementacji klasy zarządzającej bazą danych

```
public class BazaDanych extends SQLiteOpenHelper {
    private static final String DB_NAME = "AutoAndroid";
    private static final int DB_VERSION = 14;

    public BazaDanych(Context context) {
        super(context, DB_NAME, null, DB_VERSION);
    }

    @Override
    public void onCreate(SQLiteDatabase db) {
        try {
            // Tabela ustawieniaAplikacji
            db.execSQL("CREATE TABLE ustawieniaAplikacji("
```

```
                + "id INTEGER PRIMARY KEY AUTOINCREMENT,"
                + "domyslnyProfilId INTEGER)");
        db.execSQL("INSERT INTO ustawieniaAplikacji (id, domyslnyProfilId) VALUES (1, 0)");

        // Tabela profil
        db.execSQL("CREATE TABLE profil ("
                + "id INTEGER PRIMARY KEY AUTOINCREMENT," + "marka TEXT,"
                + "model TEXT," + "rocznik DATE,"
                + "rodzajPaliwa VARCHAR(3)," + "pojemnosc FLOAT,"
                + "moc FLOAT," + "przebieg FLOAT," + "dataDodania DATE)");

        // Tabela raportyPrzyspieszenia
        db.execSQL("CREATE TABLE raportyPrzyspieszenia ("
                + "id INTEGER PRIMARY KEY AUTOINCREMENT,"
                + "profil_id INTEGER," + "dataRaportu DATE,"
                + "czas FLOAT," + "dokladnosc FLOAT)");

        // Tabela danePrzyspieszenia
        db.execSQL("CREATE TABLE danePrzyspieszenia ("
                + "id INTEGER PRIMARY KEY AUTOINCREMENT,"
                + "raport_id INTEGER," + "czas INTEGER," + "droga FLOAT,"
                + "predkosc FLOAT)");

        // Tabela raportySpalania
        db.execSQL("CREATE TABLE raportySpalania ("
                + "id INTEGER PRIMARY KEY AUTOINCREMENT,"
                + "profil_id INTEGER," + "dataRaportu DATE,"
                + "ileLitrow FLOAT," + "ilePrzejechano FLOAT,"
                + "styl INTEGER)");

        // Tabela wydatki
        db.execSQL("CREATE TABLE wydatki ("
                + "id INTEGER PRIMARY KEY AUTOINCREMENT,"
                + "profil_id INTEGER," + "kategoriaWydatku_id INTEGER,"
                + "dataWydatku DATE," + "koszt FLOAT," + "nazwa TEXT, "
                + "opis TEXT)");

        // Tabela kategorieWydatkow
        db.execSQL("CREATE TABLE kategorieWydatkow ("
                + "id INTEGER PRIMARY KEY AUTOINCREMENT," + "nazwa TEXT)");
        db.execSQL("INSERT INTO kategorieWydatkow (nazwa) VALUES ('paliwo')");
        db.execSQL("INSERT INTO kategorieWydatkow (nazwa) VALUES ('naprawy')");
        db.execSQL("INSERT INTO kategorieWydatkow (nazwa) VALUES ('kosmetyka')");
        db.execSQL("INSERT INTO kategorieWydatkow (nazwa) VALUES ('inne')");

        // Tabela rutynowe wymiany
        db.execSQL("CREATE TABLE rutynoweWymiany ("
                + "id INTEGER PRIMARY KEY AUTOINCREMENT,"
                + "profil_id INTEGER," + "coWymienic TEXT,"
                + "dataOstatniejWymiany DATE," + "coIleKmWymiana INTEGER,"
                + "coIleCzasuWymiana INTEGER)");
    } catch (SQLException ex) {
        Log.d("BazaDanych", ex.getMessage());
    }
}

@Override
public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
    db.execSQL("DROP TABLE IF EXISTS profil");
    db.execSQL("DROP TABLE IF EXISTS ustawieniaAplikacji");
    db.execSQL("DROP TABLE IF EXISTS raportyPrzyspieszenia");
    db.execSQL("DROP TABLE IF EXISTS danePrzyspieszenia");
    db.execSQL("DROP TABLE IF EXISTS raportySpalania");
    db.execSQL("DROP TABLE IF EXISTS wydatki");
    db.execSQL("DROP TABLE IF EXISTS kategorieWydatkow");
    db.execSQL("DROP TABLE IF EXISTS rutynoweWymiany");
    onCreate(db);
}

}
```

W większości nazwy tabel i pól sugerują ich przeznaczenie. Dodatkowego objaśnienia wymaga jedynie tabela *ustawieniaAplikacji*. Zawiera ona zawsze tylko jeden wiersz danych, przechowujący obecne ustawienia programu. Wiersz ten jest dodawany zaraz po utworzeniu

tabeli. W obecnej wersji aplikacji, jedynym zapamiętywanym ustawieniem, jest numer identyfikacyjny aktualnie aktywnego profilu, przechowywany w kolumnie *domyslnyProfilId*. Jeżeli wartość tego pola wynosi 0, oznacza to, że brak jest zdefiniowanego domyślnego profilu. Implementacja tej klasy, została przedstawiona na poniższym listingu.

3.2.2 Profile użytkownika

Informacje o wszystkich profilach programu znajdują się w tabeli *profil*. Większość aktywności programu, wykorzystuje informację o aktywnym profilu. Aby ułatwić im dostęp do tych informacji, została utworzona specjalna klasa o nazwie *Profil*, bazująca na wzorcu projektowym Singleton. Takie rozwiązanie, ma na celu zagwarantowanie, że dane te będą zawsze spójne i aktualne. Aby uzyskać dostęp do instancji tej klasy, należy wywołać statyczną metodą *getInstance(Context contex)*, która przyjmuje w parametrze kontekst aplikacji. Kontekst jest niezbędny, ponieważ na jego podstawie uzyskuje się z dostęp do bazy danych aplikacji. Najczęściej wykorzystywaną metodą tej klasy jest funkcja *getId()* która zwraca identyfikator aktywnego profilu. Zmianę aktywnego profilu umożliwia metoda *zmienDomyslnyProfil(long profilId, Context context)*.

Listing 2: Implementacja klasy Profil

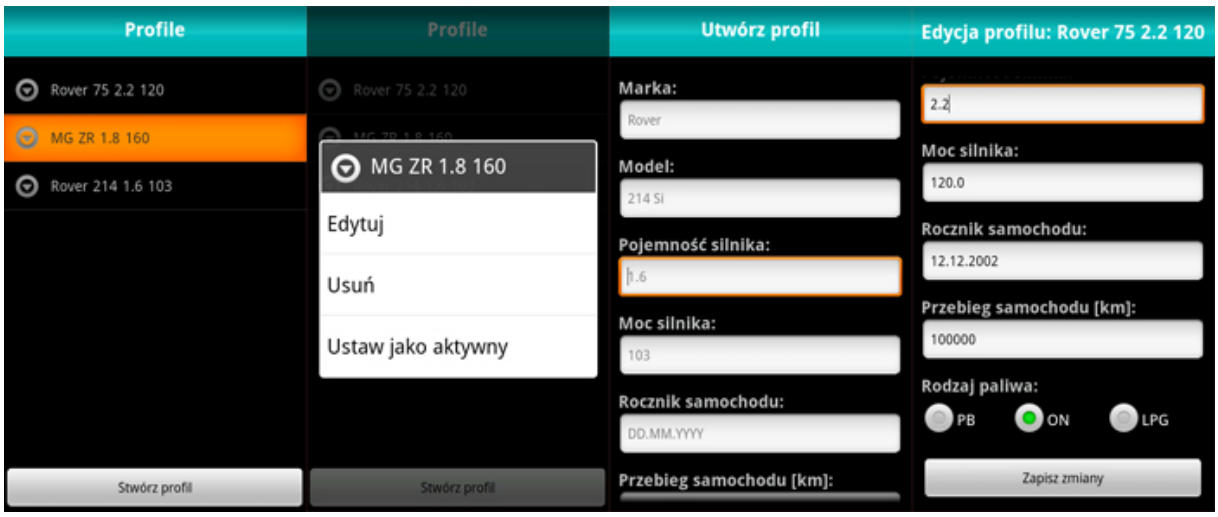
```
public class Profil {
    private static Profil profil = null;
    private int id = 0;
    private String marka = "", model = "", pojemnosc = "", moc = "", rodzajPaliwa = "";

    private Profil() {
    };

    // Konstruktor singletonu - pobiera z bazy danych domyslny profil
    // Jezeli nie ma zadnego stworzonego profilu, to id = 0
    private Profil(Context context) {
        zaktualizujDane(context);
    }

    public static synchronized Profil getInstance(Context context) {
        // Tworze jedyna instancje
        if (profil == null)
            profil = new Profil(context);
        // Aktualizuje wczesinej stworzona instancje obiektu
        else
            profil.zaktualizujDane(context);

        return profil;
    }

    // Pobiera z bazy danych dane dotyczace domyslnie wybranego profilu
    public void zaktualizujDane(Context context) {
        // Otwieram polaczenie z baza danych
        BazaDanych baza = new BazaDanych(context);
        SQLiteDatabase db = baza.getWritableDatabase();

        // Pobieram id domyslnego profilu
        Cursor cursor = db.rawQuery(
                        "SELECT domyslnyProfilId FROM ustawieniaAplikacji WHERE id = 1",
                        null);
```

```
        if (cursor.moveToFirst()) { // Zwrocilo jakies dane
            this.id = cursor.getInt(cursor.getColumnIndex("domyslnyProfilId"));
        }

        // Pobieram dane profilu
        cursor = db.rawQuery(
                "SELECT marka, model, rodzajPaliwa, pojemnosc, moc FROM profil WHERE id = "
                + this.id, null);
        if (cursor.moveToFirst()) { // Zwrocilo jakies dane
            this.marka = cursor.getString(cursor.getColumnIndex("marka"));
            this.model = cursor.getString(cursor.getColumnIndex("model"));
            this.rodzajPaliwa = cursor.getString(cursor.getColumnIndex("rodzajPaliwa"));
            this.pojemnosc = cursor.getString(cursor.getColumnIndex("pojemnosc"));
            this.moc = cursor.getString(cursor.getColumnIndex("moc"));
        } else { // Profil nie istnieje
            this.id = 0;
        }

        // Zamykam polaczenie z baza danych
        baza.close();
    }

    public boolean zmienDomyslnyProfil(long profilId, Context context) {
        // Otwieram polaczenie z baza danych
        BazaDanych baza = new BazaDanych(context);
        SQLiteDatabase db = baza.getWritableDatabase();

        // Sprawdzam czy podane ID jest prawidlowe
        String query = "SELECT id FROM profil WHERE id = " + profilId;
        Cursor c = db.rawQuery(query, null);
        if (!c.moveToFirst()) {
            // Profil o takim ID nie istnieje
            return false;
        }

        // Ustawiam podany profil jako domyslny
        query = "UPDATE ustawieniaAplikacji SET domyslnyProfilId = " + profilId
                + " WHERE id = " + 1;
        db.execSQL(query);

        // Zamykam polaczenie z baza danych
        baza.close();
        return true;
    }

    public int getId() {
        return this.id;
    }

    public String getModel() {
        return this.model;
    }

    public String getMarka() {
        return this.marka;
    }

    public String getPojemnosc() {
        return this.pojemnosc;
    }

    public String getMoc() {
        return this.moc;
    }
}
```

3.2.2.1 Zarządzanie profilami

Użytkownik programu może dodawać, usuwać oraz modyfikować profile. Może także

zmienić aktywny profil. Służą do tego klasy: *ActivityUtworzProfil*, *ActivityEdytujProfil oraz ActivityProfile*. Wszystkie z nich rozszerzają klasę *Activity*, tworzą więc widok prezentowany użytkownikowi. Klasa *ActivityProfile* pobiera z bazy danych i wyświetla listę wszystkich profili programu. Lista z profilami reaguje na długie kliknięcie, wyświetlając menu kontekstowe z opcjami usunięcia oraz zmiany aktywnego profilu. Usunięcie profilu skutkuje wykasowaniem z bazy danych wszystkich powiązanych z nim informacji. Służy do tego metoda *usunProfil(long profilId)*. Jeżeli użytkownik usunie aktywny profil, wtedy w jego miejsce zostanie automatycznie przypisany inny, taki, który został utworzony najpóźniej. Usunięcie ostatniego profilu, spowoduje zapisanie w kolumnie *domyslnyProfilId* tabeli *ustawieniaAplikacji* wartości 0. Spowoduje to zablokowanie funkcjonalności programu, do momentu, aż użytkownik stworzy nowy profil. Implementacja klas *ActivityUtworzProfil* oraz *ActivityEdytujProfil* jest podobna. Obie wyświetlają użytkownikowi ten sam formularz z danymi profilu. Różnica polega na tym, że w przypadku edycji, pobierane są z bazy danych informacje o istniejącym profilu. Obie klasy, przed zapisaniem danych, sprawdzają ich poprawność za pomocą metody *walidacjaPolFormularza()*. Jeżeli wartość jakiegoś pola będzie niepoprawna, wtedy użytkownikowi zostaje wyświetlona stosowna informacja. Nowo utworzony profil, automatycznie jest ustawiany jako aktywny.



Ilustracja 9: Widok aktywności zarządzających profilami.

3.2.3 Biblioteka aChartEngine

Program prezentuje różnego rodzaju dane oraz zestawienia na wykresach. Wykorzystuje do tego bibliotekę *aChartEngine* w wersji 0.6, którą można pobrać wraz z jej dokumentacją ze

strony: *http://code.google.com/p/achartengine/downloads/list*. Udostępnia ona obszerne API do tworzenia różnego rodzaju diagramów i wykresów. Daje też bardzo duże możliwości w dostosowywaniu wyglądu wykresu oraz prezentowanych danych. Wykres utworzony z pomocą tej biblioteki, tworzy wygodny interfejs użytkownika, obsługujący manipulację za pomocą gestów.

Biblioteka jest dosyć trudna w użyciu, a wyświetlenie wykresu wymaga sporej ilości kodu. Z tego powodu została zaimplementowana klasa *Wykres*, której zadaniem jest udostępnienie prostego API, wykorzystywanego w obrębie aplikacji. Definiuje ona także wygląd oraz ustawienia wspólne dla wszystkich wykresów prezentowanych w programie. Za pomocą biblioteki można tworzyć wiele rodzai wykresów, jednak w aplikacji występują tylko dwa typu: wykres liniowy oraz wykres czasowy. Różnica pomiędzy nimi jest taka, że na osi X pierwszego z nich znajdują się liczby rzeczywiste typu *Double*, zaś na osi X drugiego z nich znajduje się typ *Date*. Biblioteka przy tworzeniu wykresu, bazuje na dwóch wbudowanych typach: *XYMultipleSeriesDataset* oraz *XYMultipleSeriesRenderer*. Definiują one kolejno: serie danych do wyświetlenia oraz ustawienia prezentacji każdej z serii. Gotowy wykres jest reprezentowany przez typ *GraphicalView* oraz jest osadzany w widoku aktywności w elemencie typu *LinearLayout*. Głównymi polami klasy są zmienne, reprezentujące wyżej wspomniane typu.

Listing 3: Implementacja klasy Wykres. Główne pola.

```
// Typy wykresow
private static final int LINE_CHART = 1;
private static final int TIME_CHART = 2;
private int typWykresu;

// Zbior serii wykresow
private XYMultipleSeriesDataset mDataset = null;
// Ustawienia rysowania dla kazdej serii danych
private XYMultipleSeriesRenderer mRenderer = null;
// Widok wykresu
private GraphicalView mChartView = null;
```

W klasie Wykres zostały zaimplementowane wszystkie metody, umożliwiające w łatwy sposób opisanie, stworzenie i wyświetlenie wykresu. Zaprezentowane są one na poniższym listingu. Utworzeniem i wyświetleniem wykresu zajmuje się metoda *rysujWykres(Context context, LinearLayout wstawDo)* w której parametr *wstawDo* jest referencją do widoku w którym ma się pojawić wykres. Aby wykres prezentował jakieś dane, należy przed jego wyświetleniem, przekazać listę serii danych do wyświetlenia. Służą do tego dwie metody: *setLineChartData* oraz *setTimeChartData*. Różnią się one typem tworzonego wykresu

(wykres czasowy bądź liniowy). Obie przyjmują w parametrach: listę tytułów dla każdej z serii, listę tablic z wartościami leżącymi na osi X wraz z odpowiadającą jej listą tablic z wartościami leżącymi na osi Y, tablicę kolorów użytych do narysowania każdej z serii oraz tablicę ze stylami punktów.

Listing 4: Implementacja klasy Wykres. Publiczne API użytkownika.

```
public void setTytulWykresu(String tytul) { this.tytulWykresu = tytul; }
public void setTytulX(String tytul) { this.tytulX = tytul; }
public void setTytulY(String tytul) { this.tytulY = tytul; }
public void setKolorOsi(int kolor) { this.kolorOsi = kolor;     }
public void setKolorPodzialki(int kolor) { this.kolorPodzialki = kolor; }
public void setFormatDaty(String format) { this.format_daty = format; }
public void setPokazWartosci(boolean wybor) { this.pokaz_wartosci = wybor; }

public void rysujWykres(Context context, LinearLayout wstawDo) {
    // Usuwam z widoku poprzedni wykres
    wstawDo.removeAllViews();
    // Wykres liniowy
    if (typWykresu == LINE_CHART) mChartView =
        ChartFactory.getLineChartView(context, mDataset, mRenderer);
    // Wykres czasowy
    else mChartView =
        ChartFactory.getTimeChartView(context, mDataset, mRenderer, format_daty);
    // Wstawiam wykres
    wstawDo.addView(mChartView,
        new LayoutParams(LayoutParams.FILL_PARENT, LayoutParams.FILL_PARENT));
}

// Pokazuje wykres w nowej aktywnosci na pelnym ekranie
public void showFullscreen(Context context) {
    Intent i;
    if(typWykresu == LINE_CHART)
        i = ChartFactory.getLineChartIntent(context, mDataset, mRenderer);
    else
        i = ChartFactory.getTimeChartIntent(context, mDataset, mRenderer, format_daty);
    context.startActivity(i);
}

// Ustawia dane do wyswietlenia na wykresie - typ liniowy
public void setLineChartData(String[] titles, List<Double[]> xValues,
       List<Double[]> yValues, Integer[] colors, PointStyle[] styles)
{
    this.typWykresu = LINE_CHART;
    mDataset = buildDataset(titles, xValues, yValues);
    mRenderer = buildRenderer(colors, styles);
    setUstawieniaWykresu();
}

// Ustawia dane do wyswietlenia na wykresie - typ czasowy
public void setTimeChartData(String[] titles, List<Date[]> xValues,
        List<Double[]> yValues, Integer[] colors, PointStyle[] styles)
{
    this.typWykresu = TIME_CHART;
    mDataset = buildDateDataset(titles, xValues, yValues);
    mRenderer = buildRenderer(colors, styles);
    setUstawieniaWykresu();
}
```

Funkcja *setLineChartData* wywołuje trzy prywatne metody: *buildDataset, buildRenderer* oraz *setUstawieniaWykresu.* Do pierwszych dwóch, przekazywane są otrzymane w parametrach dane wykresu. Ich zadaniem jest utworzenie niezbędnych do wyświetlenia wykresu typów: *XYMultipleSeriesDataset* oraz *XYMultipleSeriesRenderer.* Funkcja *setTimeChartData* działa analogicznie, różnicą jest wywołanie innej metody tworzącej zbiór

serii, czyli metody *buildDateDataset*. W ciele funkcji *buildDataset* tworzone są obiekty wbudowanego w bibliotekę typu *XYSeries*, reprezentujące wszystkie serie danych. Seria posiada swoją nazwę oraz listę współrzędnych. Zadaniem metody jest także określenie minimalnych oraz maksymalnych wartości na osiach X oraz Y. Informacje te wykorzystywane są do późniejszego wycentrowania widoku w taki sposób, aby widoczny był cały wykres. Funkcja zwraca obiekt *XYMultipleSeriesDataset* zawierający wszystkie serie danych, przekazane do utworzenia wykresu. Metoda *buildDateDataset* działa analogicznie, różnica polega na tym, że przyjmuje ona inny typ parametru dla wartości wyświetlanych na osi X – typ *Date*. W ciele funkcji *buildRenderer* tworzone są obiekty typu *XYSeriesRenderer*, określające wygląd dla każdej serii danych. Ich lista tworzy zwracany typ *XYSeriesRenderer*. Funkcja *setUstawieniaWykresu* definiuje interfejs użytkownika wykresu.

Listing 5: Implementacja klasy Wykres. Metody buildDataset oraz buildRenderer.

```
protected XYMultipleSeriesDataset buildDataset(String[] titles, List<Double[]> xValues,
    List<Double[]> yValues)
{
    minX = minY = Double.MAX_VALUE;
    maxX = maxY = 0;
    XYMultipleSeriesDataset dataset = new XYMultipleSeriesDataset();
    int length = titles.length;
    for (int i = 0; i < length; i++) {
        XYSeries series = new XYSeries(titles[i]);
        Double[] xV = xValues.get(i);
        Double[] yV = yValues.get(i);
        int seriesLength = xV.length;
        for (int k = 0; k < seriesLength; k++) {
            series.add(xV[k], yV[k]);
            // wyznaczam minX, maxX, minY, maxY
            if(xV[k] < minX) minX = xV[k];
            if(xV[k] > maxX) maxX = xV[k];
            if(yV[k] < minY) minY = yV[k];
            if(yV[k] > maxY) maxY = yV[k];
        }
        dataset.addSeries(series);
    }
    return dataset;
}

protected XYMultipleSeriesRenderer buildRenderer(Integer[] colors, PointStyle[] styles) {
    XYMultipleSeriesRenderer renderer = new XYMultipleSeriesRenderer();
    ...
    for (int i = 0; i < length; i++) {
        XYSeriesRenderer r = new XYSeriesRenderer();
        // Ustawienia wygladu dla serii
        r.setColor(colors[i]);
        r.setPointStyle(styles[i]);
        r.setLineWidth(GRUBOSC_LINI);
        renderer.addSeriesRenderer(r);
    }
    return renderer;
}
```

3.2.4 Raporty spalania

Dane o raportach spalania przechowywane są w bazie danych w tabeli *raportySpalania*. Większość nazw kolumn tej tabeli, sugeruje ich przeznaczenie. Dodatkowego objaśnienia

wymaga jedynie kolumna *styl*. Jej zawartość określa jaki stylem jazdy samochodem, przeważał przez okres obejmujący dany raport spalania. Przyjęto, że wartość 0 to styl ekonomiczny, wartość 1 to styl normalny a wartość 2 to styl dynamiczny.

Każdy raport jest powiązany z konkretnym profilem. Użytkownik ma dostęp do aktywności: *ActivityDodajRaportSpalania, ActivityEdytujRaportSpalania, ActivityRaportySpalania oraz ActivityStatystykiSpalania.* Ostatnie dwie są wyświetlane w formie zakładek, za pomocą kolejnej aktywności *TabWidgetSpalanie*. Wszystkie raporty dla konkretnego samochodu, wyświetlane są w postaci listy w aktywności *ActivityRaportySpalania.* Wyświetlana jest także tabela z informacjami statystycznymi dotyczącymi średniego spalania w różnych warunkach. Zadaniem tym zajmuje się metoda *pobierzSrednieSpalania()*. Na szczególną uwagę zasługuje sposób obliczania średniego spalania w zimie oraz w lecie. Przyjęta została zasada, że raporty spalania w miesiącach od kwietnia do października, używane są do obliczenia średniego spalania w lecie, zaś reszta służy do obliczenia spalania w zimie.

Listing 6: Implementacja klasy ActivityRaportySpalania. Funkcja obliczająca średnie spalania.

```
// Pobiera z bazy srednie spalania oraz wstawia je do widoku
private void pobierzSrednieSpalania() {
    String spalanieOgolne, spalanieEkonomiczne, spalanieNormalne,
       spalanieDynamiczne, spalanieLato, spalanieZima, spalanieMin, spalanieMax;

    spalanieOgolne = spalanieEkonomiczne = spalanieNormalne = spalanieDynamiczne =
        spalanieLato = spalanieZima = spalanieMin = spalanieMax = "n/a";

    // Pobieram uchyt bazy danych
    BazaDanych baza = new BazaDanych(this);
    SQLiteDatabase db = baza.getWritableDatabase();

    Profil p = Profil.getInstance(this);
    // Ogolne
    String queryStart =
        "SELECT IFNULL(ROUND(100 * (SUM(ileLitrow)/SUM(ilePrzejechano)), 2), 'n/a') spalanie "
        +  "FROM raportySpalania ";
    String query = queryStart + "WHERE profil_id = "+p.getId();
    Cursor c = db.rawQuery(query, null);
    if(c.moveToFirst())       spalanieOgolne = c.getString(c.getColumnIndex("spalanie"));

    // Ekonomiczne
    query = queryStart + "WHERE styl = 0 AND profil_id = "+p.getId();
    c = db.rawQuery(query, null);
    if(c.moveToFirst()) spalanieEkonomiczne = c.getString(c.getColumnIndex("spalanie"));

    // Normalne
    query = queryStart + "WHERE styl = 1 AND profil_id = "+p.getId();
    c = db.rawQuery(query, null);
    if(c.moveToFirst()) spalanieNormalne = c.getString(c.getColumnIndex("spalanie"));

    // Dynamiczne
    query = queryStart + "WHERE styl = 2 AND profil_id = "+p.getId();
    c = db.rawQuery(query, null);
    if(c.moveToFirst()) spalanieDynamiczne = c.getString(c.getColumnIndex("spalanie"));

    // Lato
    query = queryStart + "WHERE profil_id = "+p.getId()+
            " AND cast(strftime('%m', dataRaportu) as int) BETWEEN 4 AND 10";
```

```
    c = db.rawQuery(query, null);
    if(c.moveToFirst()) spalanieLato = c.getString(c.getColumnIndex("spalanie"));

    // Zima
    query = queryStart + "WHERE profil_id = "+p.getId()+
            " AND cast(strftime('%m', dataRaportu) as int) NOT BETWEEN 4 AND 10";
    c = db.rawQuery(query, null);
    if(c.moveToFirst()) spalanieZima = c.getString(c.getColumnIndex("spalanie"));

    // Minimalne spalanie
    query =
        "SELECT (ROUND(100 * (ileLitrow/ilePrzejechano), 2)) spalanie FROM raportySpalania " +
        "WHERE profil_id = "+p.getId()+" ORDER BY spalanie ASC LIMIT 1";
    c = db.rawQuery(query, null);
    if(c.moveToFirst()) spalanieMin = c.getString(c.getColumnIndex("spalanie"));

    // Maksymalne spalanie
    query =
        "SELECT (ROUND(100 * (ileLitrow/ilePrzejechano), 2)) spalanie FROM raportySpalania " +
        "WHERE profil_id = "+p.getId()+" ORDER BY spalanie DESC LIMIT 1";
    c = db.rawQuery(query, null);
    if(c.moveToFirst()) spalanieMax = c.getString(c.getColumnIndex("spalanie"));

    // Zamykam polaczenie z baza
    baza.close();

    // Wpisuje wartosci do widoku
    ...
}
```

Klasa *ActivityStatystykiSpalania* jest aktywnością prezentującą na wykresie wszystkie spalania aktywnego profilu samochodowego. Wykorzystywana jest do tego wcześniej opisana implementacja klasy *Wykres*. Użytkownik może pokazać na wykresie wszystkie raporty spalania lub porównać ze sobą spalania w trybie: ekonomicznym, normalnym i dynamicznym. Może także zawęzić wykres do konkretnego okresu. Do realizacji tych funkcjonalności, klasa posiada odpowiednie pola, zaprezentowane na poniższym listingu. Zmienna *wykresLinearLayout* jest referencją do elementu widoku w którym wyświetlony będzie wykres. Pola *raportDataOd* oraz *raportDataDo* wykorzystywane są w budowaniu zapytań do bazy danych, określają z jakiego okresu pobrać dane. Łącznie na wykresie można wyświetlić maksymalnie cztery serie danych (spalania: wszystkie, ekonomiczne, ogólne, dynamiczne). Informacje o wybranych seriach zawiera tablica logiczna o nazwie *ktoreDaneRysowac*. Wartość pod każdym indeksem tablicy określa czy daną serię należy nanieść na wykres czy też nie. Przyjęto następujące znaczenia dla kolejnych indeksów: 0 – wszystkie raporty, 1 – styl jazdy ekonomiczny, 2 – styl jazdy normalny, 3 – styl jazdy dynamiczny. Utworzone zostały także stałe identyfikatory poszczególnych typów wykresów: *WYKRES_OGOLNY_ID, WYKRES_EKONOMICZNY_ID, WYKRES_NORMALNY_ID* oraz *WYKRES_DYNAMICZNY_ID*. Odpowiadają one wartościom poszczególnych stylów z kolumny *styl* w tabeli *raportySpalania*. Wykorzystywane są one przy tworzeniu odpowiedniego zapytania do bazy danych. Dodatkowo zostały zdefiniowane kolory dla

każdego typu serii: spalanie ogólne – cyjan, jazda ekonomiczna – zielony, jazda normalna – żółty, jazda dynamiczna – czerwony. Tytuły, style punktów oraz kolory każdej z serii danych, są przechowywane w czteroelementowych listach: *tytulySerii, stylePunktow* oraz *kolorySerii.* Współrzędne punktów każdej z serii przechowywane są również w czteroelementowych listach, zawierających tablice typu *Double*.

Listing 7: Implementacja klasy ActivityStatystykiSpalania. Najważniejsze pola klasy.

```
// Pola widoku
private LinearLayout wykresLinearLayout;
private Wykres wykres;

// Parametry wskazujace jakie dane pobrac z bazy, definicje kolorow i stylu dla serii danych
private String raportDataOd = "";
private String raportDataDo = "";
// Ktore wykresy rysowac: 0 - ogolny, 1 - ekonomiczny, 2 - normalny, 3 - dynamiczny
private boolean[] ktoreDaneRysowac = new boolean[] {true, false, false, false};
private static final int WYKRES_OGOLNY_ID = -1;
private static final int WYKRES_EKONOMICZNY_ID = 0;
private static final int WYKRES_NORMALNY_ID = 1;
private static final int WYKRES_DYNAMICZNY_ID = 2;
private static final int KOLOR_OGOLNY = Color.CYAN;
private static final int KOLOR_EKONOMICZNY = Color.GREEN;
private static final int KOLOR_NORMALNY = Color.YELLOW;
private static final int KOLOR_DYNAMICZNY = Color.RED;

// Serie danych prezentowanych na wykresie
List<Date[]> xV = new ArrayList<Date[]>();
List<Double[]> yV = new ArrayList<Double[]>();
List<String> tytulySerii = new ArrayList<String>();
List<PointStyle> stylePunktow = new ArrayList<PointStyle>();
List<Integer> kolorySerii = new ArrayList<Integer>();
```

Pobraniem i wyświetleniem na wykresie odpowiednich danych, zajmują się metody *zbudujWykres()* oraz *pobierzOkreslonyRaport(int stylJazdyId).* Pierwsza z nich, jest wywoływana bezpośrednio w metodzie cyklu życia aplikacji *onResume()*, powodując przerysowanie wykresu przy każdym powrocie do okna aktywności. Zadaniem funkcji, jest pozyskanie wszystkich wybranych przez użytkownika serii danych, oraz zaprezentowanie ich na wykresie. W tym celu przeglądana jest zawartość tablicy *ktoreDaneRysowac.* Gdy element tablicy zawiera wartość *true,* pobierana jest odpowiadająca mu seria danych. Po uzyskaniu wszystkich serii, następuje proces rysowania wykresu. Pierwszym krokiem jest inicjalizacja danych za pomocą obiektu *wykres* klasy *Wykres* oraz jej metody *setTimeChartData.* Następnie wykres jest rysowany poprzez wywołanie na obiekcie *wykres* metody *rysujWykres.* Funkcja ta przyjmuje w parametrze identyfikator serii danych którą należy pobrać z bazy danych oraz zwraca dwuelementową listę typów ogólnych *Object*. Przyjęto zasadę, że pierwszym elementem tej listy jest tablica elementów typu *Date*, czyli wartości które mają zostać wyświetlone na osi X wykresu. Drugim elementem listy jest tablica elementów typu *Double, jej wartości* zostaną wyświetlone na osi Y. Następnie budowane jest odpowiednie zapytanie

do bazy danych, pobierające średnie spalania z każdego raportu, uwzględniając aktywny profil, wybrane przez użytkownika ramy czasowe oraz styl jazdy. Wyniki zapytania są odpowiednio parsowane i dodawane do listy ostatecznie zwracanej przez funkcję.

Listing 8: Implementacja klasy ActivityStatystykiSpalania. Funkcja rysująca wykres.

```
private void zbudujWykres() {
    // Czyszcze poprzednie dane z list
    xV.clear();
    yV.clear();
    stylePunktow.clear();
    tytulySerii.clear();
    kolorySerii.clear();

    // Ogolny
    if(ktoreDaneRysowac[0]) {
        List<Object> seriaDanych = pobierzOkreslonyRaport(WYKRES_OGOLNY_ID);

        tytulySerii.add(getResources().getString(R.string.raportSpalania_seriaOgolna));
        xV.add( (Date[]) seriaDanych.get(0) );
        yV.add( (Double[]) seriaDanych.get(1) );
        kolorySerii.add(KOLOR_OGOLNY);
        stylePunktow.add(PointStyle.POINT);
    }

    // Ekonomiczny
    if(ktoreDaneRysowac[1]) {
        List<Object> seriaDanych = pobierzOkreslonyRaport(WYKRES_EKONOMICZNY_ID);

        tytulySerii.add(getResources().getString(R.string.raportSpalania_seriaEkonomiczna));
        xV.add( (Date[]) seriaDanych.get(0) );
        yV.add( (Double[]) seriaDanych.get(1) );
        kolorySerii.add(KOLOR_EKONOMICZNY);
        stylePunktow.add(PointStyle.POINT);
    }

    // Normalny
    if(ktoreDaneRysowac[2]) {
        List<Object> seriaDanych = pobierzOkreslonyRaport(WYKRES_NORMALNY_ID);
        tytulySerii.add(getResources().getString(R.string.raportSpalania_seriaNormalna));
        xV.add( (Date[]) seriaDanych.get(0) );
        yV.add( (Double[]) seriaDanych.get(1) );
        kolorySerii.add(KOLOR_NORMALNY);
        stylePunktow.add(PointStyle.POINT);
    }

    // Dynamiczny
    if(ktoreDaneRysowac[3]) {
        List<Object> seriaDanych = pobierzOkreslonyRaport(WYKRES_DYNAMICZNY_ID);
        tytulySerii.add(getResources().getString(R.string.raportSpalania_seriaDynamiczna));
        xV.add( (Date[]) seriaDanych.get(0) );
        yV.add( (Double[]) seriaDanych.get(1) );
        kolorySerii.add(KOLOR_DYNAMICZNY);
        stylePunktow.add(PointStyle.POINT);
    }

    // Wypelniam wykres danymi i go rysuje
    String[] tytulyArray = tytulySerii.toArray(new String[tytulySerii.size()]);
    Integer[] koloryArray = kolorySerii.toArray(new Integer[kolorySerii.size()]);
    PointStyle[] styleArray = stylePunktow.toArray(new PointStyle[stylePunktow.size()]);
    wykres.setTimeChartData(tytulyArray, xV, yV, koloryArray, styleArray);
    wykres.rysujWykres(this, wykresLinearLayout);
}


private List<Object> pobierzOkreslonyRaport(int stylJazdyId) {
    List<Object> res = new ArrayList<Object>();
    // Pod indeksem 0 beda wartosci x, pod indeksem 1 beda warotsci y
    Date[] x;
    Double[] y;
```

```
    // Pobieram uchyt bazy danych
    BazaDanych baza = new BazaDanych(this);
    SQLiteDatabase db = baza.getWritableDatabase();

    // Parametry dla zapytania sql
    Profil p = Profil.getInstance(this);
    String WHERE = " profil_id = "+p.getId();
    if(raportDataOd.compareTo("") != 0 && raportDataDo.compareTo("") != 0) {
        WHERE +=
            " AND dataRaportu BETWEEN DATE('"+raportDataOd+"') AND DATE('"+raportDataDo+"')";
    }

    if(stylJazdyId == WYKRES_EKONOMICZNY_ID) WHERE += " AND styl = "+WYKRES_EKONOMICZNY_ID;
    else if(stylJazdyId == WYKRES_NORMALNY_ID) WHERE += " AND styl = "+WYKRES_NORMALNY_ID;
    else if(stylJazdyId == WYKRES_DYNAMICZNY_ID) WHERE += " AND styl = "+WYKRES_DYNAMICZNY_ID;


    // Pobieram raporty srednich spalan
    String query = "SELECT (ileLitrow/ilePrzejechano) spalanie, dataRaportu FROM " +
        " raportySpalania WHERE "+WHERE+" ORDER BY dataRaportu DESC, id DESC";
    Cursor c = db.rawQuery(query, null);

    // Czy sa dane
    int ileWierszy = c.getCount();
    x = new Date[ileWierszy];
    y = new Double[ileWierszy];
    int i = 0;
    if( c.moveToFirst() ) {
        do {
            String dataString = c.getString(c.getColumnIndex("dataRaportu"));
            SimpleDateFormat df = new SimpleDateFormat("yyyy-MM-dd");
            Date data;
            try {
                data = df.parse(dataString);
            } catch (ParseException e) {
                // Jezeli blad to zwracam pusta liste punktow - nie bedzie wykresu
                res.add(new Date[] {});
                res.add(new Double[] {});
                return res;
            }
            Double spalanie = c.getDouble(c.getColumnIndex("spalanie"));
            spalanie = spalanie * 100.0;
            spalanie = (int)(100 * spalanie) / 100.0;
            x[i] = data;
            y[i] = spalanie;
            i++;
        } while(c.moveToNext());
    }

    // Zamykam polaczenie z baza
    baza.close();

    // Zwracam liste punktow
    res.add(x);
    res.add(y);

    return res;
}
```

Ilustracja 10: Widok aktywności raportów spalania.

3.2.4.1 Obsługa zmiany orientacji w systemie Android

Wyświetlenie wykresu w klasie *ActivityStatystykiSpalania* wymusza obsłużenie szczególnej sytuacji, występującej przy zmianie orientacji wyświetlanego widoku. Okazuje się, że przy każdej zmianie orientacji, system Android usuwa poprzednią aktywność oraz tworzy ją na nowo, wywołując jej metodę *onCreate* z cyklu życia aplikacji. Wiąże się to z utratą wszystkich danych, zgromadzonych w poprzedniej aktywności. W przypadku klasy

wyświetlającej wykres, może zdarzyć się następująca sytuacja: użytkownik będąc w orientacji pionowej, wyświetli na wykresie wszystkie możliwe serie danych, następnie zmieni orientację na poziomą. W tym momencie jego dotychczasowa aktywność zostanie usunięta a w zamian niej utworzona zostanie nowa. Niestety nowa aktywność nie będzie już zawierała informacji o seriach danych, które wcześniej wybrał użytkownik programu. Spowoduje to, że zostanie wyświetlona jedynie seria wszystkich raportów. Z taką sytuacją można sobie poradzić przeładowując dwie metody z klasy *Activity, są nimi: onSaveInstanceState(Bundle outState) oraz onRestoreInstanceState(Bundle savedState).* Pierwsza z nich jest wywoływana tuż przed zniszczeniem dotychczasowej aktywności. Funkcja ta otrzymuje w parametrze referencję do zasobu, w którym można umieścić własne dane. Druga funkcja jest wywoływana zaraz po utworzeniu aktywności. Otrzymuje ona w parametrze ten sam zasób, który otrzymała metoda *onSaveInstanceState*. Możliwa jest więc komunikacja między dwiema aktywnościami za pomocą zasobu typu *Bundle.* W praktyce zamiast metody *onRestoreInstanceState* można też bezpośrednio użyć metody *onCreate(Bundle bundle),* która w parametrze również otrzymuje ten sam zasób, który można modyfikować w metodzie *onSaveInstanceState.* [27]

W klasie *ActivityStatystykiSpalania* danymi które powinny być utrwalone w trakcie zmiany orientacji, są pola klasy: *ktoreDaneRysowac, raportDataOd* oraz *raportDataDo*. Poniższy listing zawiera implementację dwóch metod, zapewniających spójność danych wykresu.

Listing 9: Implementacja klasy ActivityStatystykiSpalania. Utrwalenie danych w trakcie zmiany orientacji systemu.

```
// Przywracam aktywnosc przed obrotem
@Override
protected void onRestoreInstanceState(Bundle savedState) {
    super.onRestoreInstanceState(savedState);
    ktoreDaneRysowac = savedState.getBooleanArray("ktoreDaneRysowac");
    raportDataOd = savedState.getString("raportDataOd");
    raportDataDo = savedState.getString("raportDataDo");
}

// Zapisuje stan aktywnosci przed obrotem
@Override
protected void onSaveInstanceState(Bundle outState) {
    super.onSaveInstanceState(outState);
    outState.putBooleanArray("ktoreDaneRysowac", ktoreDaneRysowac);
    outState.putString("raportDataOd", raportDataOd);
    outState.putString("raportDataDo", raportDataDo);
}
```

3.2.5 Ewidencja wydatków

Informacje o wydatkach są przechowywane w tabelach *wydatki* oraz *kategorieWydatkow.* Nazwy ich kolumn sugerują zawartość poszczególnych pól. Domyślnie w programie dostępne

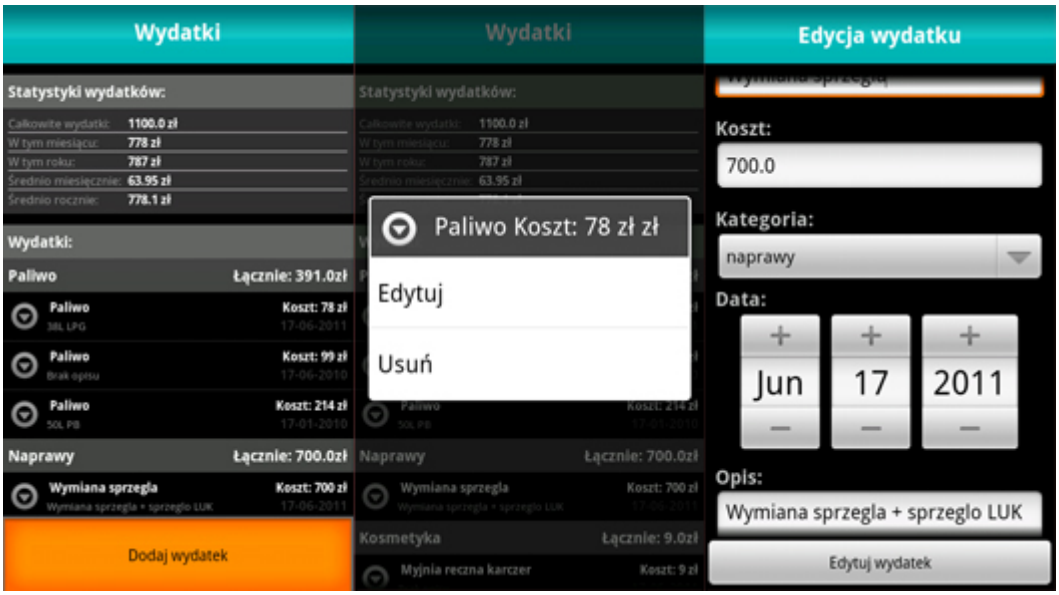
są cztery kategorie wydatków, dodawane w momencie tworzenia bazy danych aplikacji. Obecna wersja programu nie umożliwia użytkownikowi dodania nowej kategorii wydatków. Dodanie takiej funkcjonalności w przyszłości będzie bardzo proste, gdyż aby pojawiła się nowa kategoria, wystarczy jedynie dodać nowy wpis w tabeli *kategorieWydatkow*.

Użytkownik ma dostęp do aktywności: *ActivityDodajWydatek, ActivityEdytujWydatek* oraz *ActivityWydatki*. Pierwsze dwie klasy są bardzo do siebie zbliżone, różnica polega na tym, że druga z nich, pobiera z bazy danych informację o istniejącym wydatku i umożliwia jego edycję. Obie klasy weryfikują poprawność wprowadzanych przez użytkownika danych. Służy do tego metoda *walidacjaPolFormularza()*. Jeżeli zawartość któregoś pola będzie nieodpowiednia, aplikacja wskaże użytkownikowi błąd. Aktywność *ActivityWydatki* wyświetla listę wydatków, podzieloną według kategorii. Przy nagłówku każdej z kategorii, wyświetlana jest suma wszystkich należących do niej wydatków. Dodatkowo, wyświetlana jest tabela zawierająca informacje statystyczne o wydatkach, takie jak: całkowite wydatki, wydatki w obecnym miesiącu, wydatki w obecnym roku, średnie miesięczne wydatki oraz średnie roczne wydatki. Zadaniem tym zajmuje się część metody *zapelnijWidokDanymi()*. Jej implementacja pokazana jest na poniższym listingu. Zapytanie do bazy danych jest dosyć obszerne, lecz zwraca za jednym razem wszystkie potrzebne do obliczeń dane. Zmienna *subquery1* zawiera część zapytania która w odpowiedzi dostaje sumę wszystkich wydatków z obecnego miesiąca. Zmienna *subquery2* jest podzapytaniem, które w odpowiedzi dostaje sumę wszystkich wydatków w trwającym roku. Zmienna *query* jest złożeniem dwóch powyższych podzapytań, oraz dodatkowo dostaje w odpowiedzi informację o sumie wszystkich wydatków i różnicy wyrażonej w dniach, pomiędzy datą pierwszego wydatku a obecnym dniem. Zapytanie uwzględnia aktywny profil.

Listing 10: Implementacja klasy ActivityWydatki. Funkcja obliczająca dane statystyczne.

```
// Funkcja zapelnia widok danymi
private void zapelnijWidokDanymi() {
    // Pobieram uchyt bazy danych
    BazaDanych baza = new BazaDanych(this);
    SQLiteDatabase db = baza.getWritableDatabase();

    // Pobieram statystyki wydatki: calkowite, miesieczne, roczne
    Profil p = Profil.getInstance(this);
    // Pobieram numer aktualnego miesiaca i roku
    Calendar calendar = Calendar.getInstance();
    int tmp = calendar.get(Calendar.MONTH) + 1;
    String miesiac = tmp + "";
    if(tmp < 9) miesiac = "0" + miesiac;
    String rok = calendar.get(Calendar.YEAR) + "";
    // Zapytanie o wydatki w tym miesiacu
    String subquery1 =
        "SELECT IFNULL( SUM(koszt), 0 ) FROM wydatki WHERE profil_id = "+p.getId()+
```

```
        " AND strftime('%m', dataWydatku) = '"+miesiac+"' AND " +
        "strftime('%Y', dataWydatku) = '"+rok+"'";
    // Zapytanie o wydatki w tym roku
    String subquery2 =
        "SELECT IFNULL( SUM(koszt), 0 ) FROM wydatki WHERE profil_id = "+p.getId()+
        " AND strftime('%Y', dataWydatku) = '"+rok+"'";
    // Zapytanie o wszystko na raz
    String query = "" +
            "SELECT " +
                "IFNULL( SUM(w.koszt), 0 ) calkowite ," +
                "IFNULL( JULIANDAY('now') - JULIANDAY(MIN(dataWydatku)), 0 ) roznica ," +
                "( "+subquery1+" ) miesieczne ," +
                "( "+subquery2+" ) roczne " +
            "FROM wydatki w " +
            "WHERE profil_id = "+p.getId();
    Cursor c = db.rawQuery(query, null);
    if(c.moveToFirst()) {
        String waluta = getResources().getString(R.string.waluta);
        Double calkowite = c.getDouble(c.getColumnIndex("calkowite"));
        wydatkiCalkowite.setText( calkowite + " " + waluta );
        wydatkiMiesieczne.setText(
            c.getString(c.getColumnIndex("miesieczne")) + " " + waluta
        );
        wydatkiRoczne.setText( c.getString(c.getColumnIndex("roczne")) + " " + waluta );
        int roznica = c.getInt(c.getColumnIndex("roznica"));
        double dzielnikM = roznica / 30.0;
        double dzielnikR = roznica / 365.0;
        if(dzielnikM != 0.0 && dzielnikM > 1) {
            double srednia = ((int)(100 * (calkowite / dzielnikM))) / 100.0;
            wydatkiSrednieMiesieczne.setText( srednia + " " + waluta );
        } else wydatkiSrednieMiesieczne.setText( calkowite + " " + waluta );
        if(dzielnikR != 0.0 && dzielnikR > 1 ) {
            double srednia = ((int)(100 * (calkowite / dzielnikR))) / 100.0;
            wydatkiSrednieRoczne.setText( srednia + " " + waluta );
        } else wydatkiSrednieRoczne.setText( calkowite + " " + waluta );
    }
    c.close();

    ...
}
```



Ilustracja 11: Widok aktywności użytkownika związanych z wydatkami.

3.2.6 Rutynowe wymiany

Informacje o wymianach są przechowywane w tabeli *rutynoweWymiany*. Aplikacja generuje powiadomienia o potrzebie wymiany: oleju, rozrządu, ubezpieczenia oraz przeglądu. Wartość zero wpisana w polach tabeli: *coIleKmWymiana* oraz *coIleCzasuWymiana,* oznacza, że dane powiadomienie jest niezainicjowane. Aby powiadomienia były aktywne, użytkownik musi wprowadzić informacje o dacie ostatniej wymiany oraz jej częstotliwości. W przypadku oleju oraz rozrządu, podaje się co ile kilometrów oraz co ile miesięcy następuje wymiana. Informacje o wymianie ubezpieczenia i przeglądu podaje się w liczbie miesięcy. Użytkownik ma dostęp do aktywności *ActivityWymiany*, która przy pierwszym uruchomieniu wymusza podanie wszystkich niezbędnych informacji. Zajmuje się tym funkcja *zainicjujWszystkieWymiany()*. Aktywność wyświetla również tabelę z informacjami o każdej z rzeczy, informując użytkownika o czasie, bądź ilości kilometrów, pozostałych do jej wymiany. Zajmuje się tym metoda *zapelnijWidokDanymi*. W klasie zaimplementowana została statyczna funkcja *sprawdzCoNalezyWymienic*. *S*prawdza ona czy nadszedł czas wymiany którejś z rzeczy, ustawiając przy tym powiadomienie wyświetlane w górnym pasku systemu. Program wywołując funkcję *ustawUslugeWymian()* ustawia powtarzający się alarm systemowy, który co pewien czas sprawdza, czy należy coś wymienić, nawet gdy aplikacja nie jest aktualnie uruchomiona. Odbiorcą alarmu jest klasa *OnAlarmReceiver* która w wywołaniu swojej funkcji zwrotnej *onReceive* uruchamia specjalną usługę zdefiniowaną w klasie *ServiceWymiany*. W usłudze wywoływana jest metoda *sprawdzCoNalezyWymienic* z klasy *ActivityWymiany.* W ten sposób użytkownik zostanie automatycznie powiadomiony o potrzebie wymiany którejś z rzeczy.

Użytkownik jest powiadamiany o wymianach tak długo, aż w końcu poinformuje program, że dana rzecz została wymieniona.

Ilustracja 12: Widoki użytkownika związane z rutynowymi wymianami.

3.2.7 Graficzny prędkościomierz

W widokach dwóch aktywności programu AutoAndroid, wyświetlany jest graficzny licznik samochodowy, prezentujący aktualną prędkość pojazdu. Został on stworzony specjalnie na potrzeby aplikacji. Jego implementacja znajduje się w klasie *Licznik*. Tak jak każdy element graficzny systemu Android, rozszerza ona klasę *View* oraz przeładowuje jej metodę *onDraw(Canvas canvas)*. W tej funkcji odbywa się proces rysowania widżetu, na przekazanym w parametrze obiekcie typu *Canvas* (pl. płótno). Licznik składa się ze statycznej tarczy z zaznaczoną skalą oraz ze wskazówki i wartości prędkości pojazdu, które są nanoszone dynamicznie w trakcie działania programu. Aby usprawnić proces wyświetlania licznika, rozdzielono rysowanie tarczy od nanoszenia na nią informacji o prędkości. Tarcza licznika jest tworzona tylko raz, przy pierwszym jej rysowaniu, po którym zapamiętywana jest w polu klasy typu *Bitmap w* zmiennej o nazwie *tarcza*. Klasa *Licznik* zawiera także pola definiujące wygląd wykresu oraz jego ustawienia, dzięki którym można dostosować lub zmienić wygląd licznika bez ingerencji w kod odpowiedzialny za rysowanie. Najistotniejsze są zmienne *maxV, arcAngle, unitsScale* oraz *degreesPerUnit*. Pierwsza z nich określa maksymalną możliwą prędkość wyświetlaną przez licznik. Druga zmienna definiuje miarę kąta wycinka koła, który stanowi pole dla tarczy. Zmienna *unitsScale* określa co ile jednostek prędkości, należy nanosić na tarczę podziałkę, zaś zmienna *degreesPerUnit* wyrażą tę częstotliwość za pomocą miary kąta. Pole *currentV* przechowuje aktualnie wyświetlaną przez licznik prędkość.

Listing 11: Implementacja widoku Licznik. Pola klasy.

```
// Bitmapa z tarcza licznika
Bitmap tarcza = null;

// Ustawienia rysowania
private Paint cNiebieski;
private Paint cBialy;
private int arcStrokeWidth = 3; // Grubosc luku licznika
private int thinDashStrokeWidth = 2;
private int thickDashStrokeWidth = 3;
private int currentVFontSize = 18; // Wielkosc czcionki wyświetlanej predkości
private int VValueFontSize = 18; // Wielkosc czcionki dla wielkosci liter na predkosciomierzu

// Ustawienia licznika
private int maxV;
private float currentV;
private int arcAngle; // kat łuku licznika
private int unitsScale; // jednostki na liczniku
private float degreesPerUnit; // zmiana kata na jedna jednostke

// Ustawienia widoku
private int availableWidth; // szerokosc ktora dysponuje widok
```

W klasie został zdefiniowany sparametryzowany konstruktor *Licznik(Context context, AttributeSet attrs),* który umożliwia tworzenie obiektu *Licznik* za pomocą deklaracji w osobnych plikach XML. Przedstawione jest to na poniższym listingu. W konstruktorze nadawane są domyślne wartości pól. W większości samochodów, kąt wycinka okręgu tarczy licznika, wynosi 240 stopni, a skala podziałki jest zaznaczana co każde 5 km/h. Takie wartości zostały przypisane zmiennym *arcAngle* oraz *unitsScale*. Domyślna maksymalna prędkość jest ustawiona na 220 km/h. Po przypisaniu tych wartości, obliczana jest wartość zmiennej *degreesPerUnit*.

Listing 12: Implementacja widoku Licznik. Konstruktor oraz kod osadzający licznik w widoku za pomocą XML.

```
public Licznik(Context context, AttributeSet attrs) {
    super(context, attrs);

    // Inicjalizacja wartosci rysowania
    ...

    // Inicjalizacja ustawien licznika
    currentV = (float)0.0;
    maxV = 220;
    arcAngle = 240;
    unitsScale = 5;
    degreesPerUnit = arcAngle / ( maxV / (float)unitsScale );

    // Inicjalizacja kolorów
    ...
}

// Sposób osadzenia licznika w widoku zdefiniowanego w XML
<marcingv.autoandroid.Licznik
    android:id="@+id/licznik"
    android:layout_width="fill_parent"
    android:layout_height="250dp"
    android:layout_weight="1"/>
```

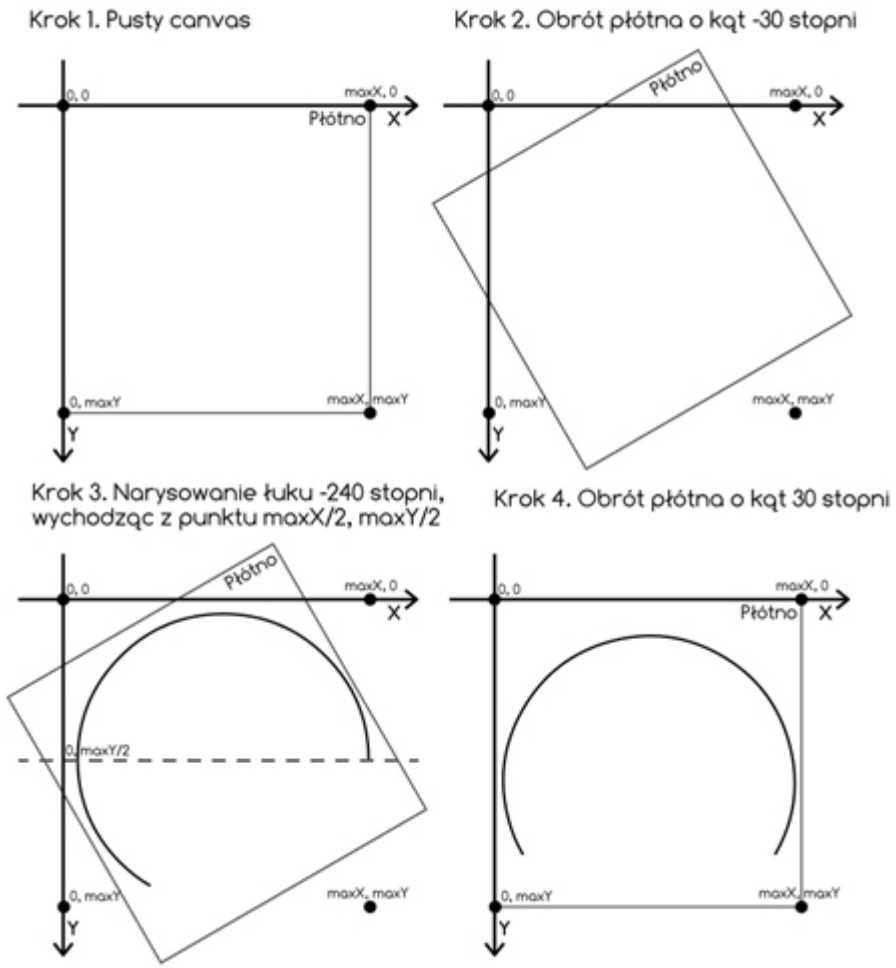
Metoda *onDraw* jest automatycznie wywoływana za każdym razem gdy na ekranie trzeba wyświetlić dany widok. Przyjmuje ona w parametrze obiekt *Canvas* na którym przeprowadza proces rysowania. Rysowanie licznika składa się z czterech etapów. Najpierw sprawdza się czy tarcza licznika została już wcześniej narysowana. Jeżeli nie, to jest ona tworzona jako nowa, pusta bitmapa, następnie uruchamiany jest jej proces rysowania, za który odpowiada metoda *narysujTarczeLicznika()*. Przygotowaną tarczę licznika przechowuje pole klasy o nazwie *tarcza.* W następnym kroku jest ona nanoszona na otrzymane w parametrze płótno. W trzecim oraz czwartym kroku na płótno nanoszona jest wskazówka oraz aktualna prędkość. Odpowiadają za to metody *narysujWskazowke(Canvas canvas)* oraz *narysujPredkosc(Canvas canvas)*.

Listing 13: Implementacja widoku Licznik. Funkcja rysująca onDraw.

```
@Override
public void onDraw(Canvas canvas) {
    // Tarcze licznika rysuje tylko raz
    if( tarcza == null ) {
        tarcza = Bitmap.createBitmap(getWidth(), getWidth(), Bitmap.Config.ARGB_8888);
        narysujTarczeLicznika();
    }

    // Rysuje tarcze na aktualnym canvasie
    canvas.drawBitmap(tarcza, 0, 0, null);
    // Rysuje wskazowke
    narysujWskazowke(canvas);
    // Nanosze napis z aktualna predkoscia
    narysujPredkosc(canvas);
}
```

Implementacja funkcji rysującej tarczę licznika jest przedstawiona na poniższym listingu. Wykorzystuje ona obiekt *Canvas* stanowiący warstwę płótna, należącego do obiektu *tarcza*. Licznik zajmuje całe dostępne miejsce zarezerwowane do wyświetlenia widoku. Dostępna szerokość jest odczytywana za pomocą metody *getWidth()* i przechowywana w zmiennej *availableWidth*. Większość czynności rysujących po widoku jest poprzedzona serią obrotów lub przesunięć, które zmieniają punkt odniesienia układu współrzędnych do płaszczyzny płótna. Takie podejście ułatwia programiście proces rysowania, zwalniając go z wyznaczania trudnych do obliczenia współrzędnych punktów. Idea tego sposobu zaprezentowana jest na poniższej ilustracji. W pierwszej kolejności rysowany jest łuk tarczy licznika. Następnie nanoszona jest podziałka skali. Są dwa rodzaje kresek podziałki. Co czwarta kreseczka jest rysowana grubą linią a obok niej umieszczana jest wartość prędkości na jaką wskazuje. Pozostałe kreseczki są rysowane cienkimi liniami bez umieszczania obok nich wartości. Przed rozpoczęciem tworzenia podziałki, płótno jest obracane o kąt -30 stopni (w kierunku przeciwnym do ruchu wskazówek zegara) względem środka widoku. W wyniku tej operacji, początek łuku licznika znalazł się w punkcie (0, *availableWidth*/2). Pętla zajmująca się

tworzeniem podziałki, stopniowo obraca płótno o kąt *degreesPerUnit* dzięki czemu kolejne miejsca w których należy narysować podziałkę znajdują się na prostej równoległej do osi X oraz przechodzącej przez punkt (0, *availableWidth*/2). Po wykonaniu podziałki skali, płótno jest obracane do położenia wyjściowego.



Ilustracja 13: Sposób rysowania łuku tarczy licznika.

Listing 14: Implementacja widoku Licznik. Funkcja rysująca tarczę licznika.

```
private void narysujTarczeLicznika() {
    Canvas canvas = new Canvas(this.tarcza);
    // Obliczam dostepne miejsce, przesuwam canvas aby zrobic marigines na obramowanie luku
    availableWidth = getWidth() - 2*arcStrokeWidth;
    canvas.translate(arcStrokeWidth, arcStrokeWidth);

    // Rysuje łuk licznika - caly canvas przesuwam o grubosc obramowania
    cNiebieski.setStrokeWidth(arcStrokeWidth);
    canvas.drawArc(
        new RectF(0, 0, availableWidth, availableWidth), 30, -240, false, cNiebieski
    );

    cNiebieski.setStrokeWidth(thinDashStrokeWidth);
    cBialy.setStrokeWidth(thickDashStrokeWidth);

    //Obracam canvas, tak aby poczatek luku lezal na prostej rownoleglej do osi Ox
    canvas.rotate(-30, availableWidth/2, availableWidth/2);
```

```
    float rotateBackDegrees = 30;

    for( int i=0; i <= (maxV / unitsScale); i++ ) {
        // Grubsza kreseczka przy ktorej drukuje wartosc
        if( i%2 == 0 ) {
            //rysuje kreseczke
            canvas.drawLine(5, availableWidth/2, 20, availableWidth/2, cBialy);

            //rysuje wartosc licznika przy kreseczce
            if( i%4 == 0 ) {
                String liczba = (i*unitsScale)+"";
                Rect rect = new Rect();
                cBialy.setTextSize(VValueFontSize);
                cBialy.getTextBounds(liczba, 0, liczba.length(), rect);

                canvas.rotate(-1 * i * degreesPerUnit + 30, 25 + rect.exactCenterX(),
                              availableWidth/2 + rect.exactCenterY() / 2);
                canvas.drawText(liczba, 25, availableWidth/2, cBialy);
                canvas.rotate(i * degreesPerUnit - 30, 25 + rect.exactCenterX(),
                              availableWidth/2 + rect.exactCenterY() / 2);
            }
        }
        // Ciensza kreseczka
        else canvas.drawLine(5, availableWidth/2, 15, availableWidth/2, cNiebieski);

        //Obracam canvas do nastepnej kreseczki
        canvas.rotate(degreesPerUnit, availableWidth/2, availableWidth/2);
        rotateBackDegrees -= degreesPerUnit;
    }

    //Odwracam caly canvas do wyjsciowego polozenia
    canvas.rotate(rotateBackDegrees, availableWidth/2, availableWidth/2);

    // Przesuwam canvas w swoje docelowa miejsce
    canvas.translate(-arcStrokeWidth, -arcStrokeWidth);
}
```

Funkcja *narysujWskazowke* również wykonuje serię obrotów płótna. Najpierw jest ono obracane o kąt -30 stopni. Później jest obliczany kąt nachylenia wskazówki względem początku tarczy licznika. W wyniku tych operacji prosta, na której powinna znajdować się wskazówka jest równoległa do osi X oraz przechodzi przez punkt (0, *availableWidth*/2). Na koniec płótno jest odwracane do swojego wyjściowego położenia. Ostatnia metoda *narysujPredkosc* nanosi na środek tarczy licznika napis z aktualnie wskazywaną przez licznik prędkością.

Listing 15: Implementacja widoku Licznik. Pozostałe funkcje rysujące oraz akcesory.

```
private void narysujWskazowke(Canvas canvas) {
    // Obliczam dostepne miejsce, przesuwam canvas aby zrobic marigines na obramowanie luku
    availableWidth = getWidth() - 2*arcStrokeWidth;
    canvas.translate(arcStrokeWidth, arcStrokeWidth);

    // Obracam canvas, tak aby poczatek luku byl na osi Ox
    canvas.rotate(-30, availableWidth/2, availableWidth/2);
    float rotateBackDegrees = 30;

    // Wyliczam pod jakim katem ma byc nachylona wskazowka
    float degrees = arcAngle * currentV / maxV;

    // Obracam canvas o wyliczony kat
    canvas.rotate(degrees, availableWidth/2, availableWidth/2);

    // Rysuje wskazkowke
```

```
    Path path = new Path();
    path.moveTo(availableWidth/2, availableWidth/2 - 3);
    path.lineTo(availableWidth/2, availableWidth/2 + 3);
    path.lineTo(30, availableWidth/2);
    path.close();
    canvas.drawPath(path, cBialy);

    // Odwracam canvas o wczesniej wyliczony kat
    canvas.rotate(-degrees, availableWidth/2, availableWidth/2);

    // Odwracam caly canvas do wyjsciowego polozenia
    canvas.rotate(rotateBackDegrees, availableWidth/2, availableWidth/2);

    // Przesuwam canvas w swoje docelowa miejsce
    canvas.translate(-arcStrokeWidth, -arcStrokeWidth);
}

private void narysujPredkosc(Canvas canvas) {
    String predkosc = this.currentV + " km/h";
    Rect rect = new Rect();
    cBialy.getTextBounds(predkosc, 0, predkosc.length(), rect);
    cBialy.setTextSize(currentVFontSize);
    canvas.drawText(predkosc, availableWidth/2 - rect.exactCenterX(),
                    (int)(availableWidth * 0.75) - rect.exactCenterY()/2, cBialy);
}

//Akcesory:
public void setSpeed(float speed) {
    speed = ((int)(100 * speed)) / (float)100.0;
    if( speed <= this.maxV && speed >= 0 ) this.currentV = speed;
    else if( speed < 0 ) this.currentV = 0;
    else this.currentV = this.maxV;
    // zmiana wymaga przerysowania licznika
    invalidate();
}

public float getSpeed() {
    return this.currentV;
}

public void setMaxV(int maxV) {
    this.maxV = maxV;
    this.tarcza = null;
    degreesPerUnit = arcAngle / ( maxV / (float)unitsScale );
    // zmiana wymaga przerysowanie licznika
    invalidate();
}

public int getMaxV() {
    return this.maxV;
}
```

Aktywności korzystające z widoku graficznego licznika komunikują się z jego obiektem za pomocą akcesorów: *setSpeed(float speed), getSpeed(), setMaxV(int maxV)* oraz *getMaxV()*. Użycie funkcji *setSpeed* oraz *setMaxV* zmieniają stan licznika, wymuszając jego częściowe bądź całkowite przerysowanie. Widok można przerysować wywołując metodę *invalidate()* odziedziczona po klasie *View*. Tak przygotowany graficzny prędkościomierz jest gotowy do prezentacji do danych otrzymywanych z zewnętrznego źródła. Jego wygląd przedstawia poniższa ilustracja.

Ilustracja 14: Graficzny prędkościomierz.

3.2.8 Pomiar przyspieszenia

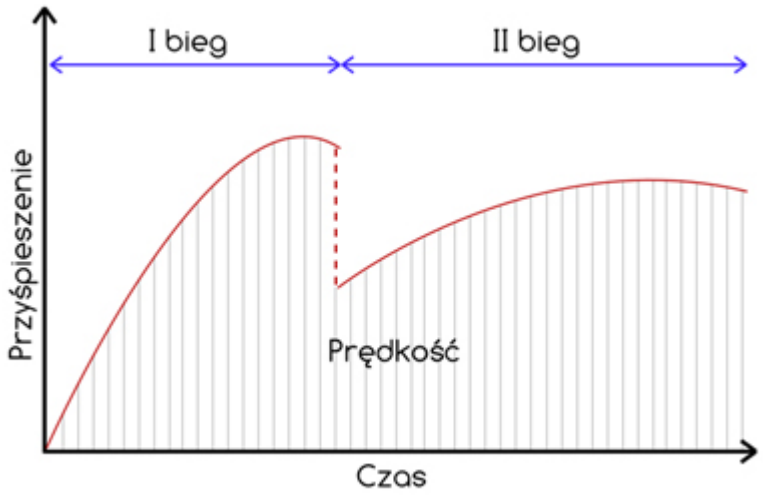
Najciekawszą funkcją aplikacji AutoAndroid jest pomiar przyspieszenia samochodu. Wszystkie przejazdy są zapamiętywane w bazie danych w tabelach *raportyPrzyspieszenia* oraz *danePrzyspieszenia.* Pierwsza z nich przechowuje datę, uzyskany czas oraz dokładność, wszystkich poprawnie wykonanych pomiarów. Sposób wyznaczania dokładności pomiaru, zostanie opisana później. Druga tabela, gromadzi wszystkie dane zarejestrowane podczas nagrywania przejazdu, niezbędne do obliczenia czasu przyśpieszenia od 0 do prędkości 100 km/h. Pomiar składa się z serii danych zawierających informację o aktualnej prędkości pojazdu, przebytej drodze oraz o czasie odczytania tych danych. Rejestrowaniem pomiarów zajmuje się aktywność *ActivityPomiarPrzyspieszenia*. Wyświetla ona graficzny licznik samochodowy, którego implementacja została opisana wcześniej. Listę dokonanych pomiarów oraz najlepszych czasów wyświetla aktywność *ActivityListaPomiarow.* Każdy przejazd można wyświetlić na dwóch wykresach: prędkości od czasu, oraz drogi od czasu. Zadanie to realizuje klasa *ActivityPomiar,* wykorzystująca do tego wcześniej opisaną klasę *Wykres*. Na wykresy można nanieść i porównać dowolne zarejestrowane przez program przejazdy, w tym także pomiary dokonane przez inne profile samochodowe.

Większość obecnych na rynku telefonów, posiada wbudowany moduł GPS, który można wykorzystać do pomiaru przyspieszenia. Oprócz tego, posiadają one wbudowany czujnik przyśpieszenia. Akcelerometr jest urządzeniem bardzo czułym na zmiany. Co więcej, można z niego odczytywać dane z bardzo dużą częstotliwością, która zwiększa ich dokładność.

Sugerując się nazwą czujnika oraz jego właściwościami, można by stwierdzić, że świetnie się on nadaje do pomiaru przyśpieszenia samochodu. W teorii jest to możliwe, jednak wykonane przez autora pracy testy oraz analizy wykazały, że w praktyce jest to niezmiernie trudne.

3.2.8.1 Teoretyczny sposób pomiaru przyśpieszenia samochodu z wykorzystaniem akcelerometru.

Akcelerometr mierzy przyspieszenie działające na urządzenie w jego trzech wymiarach. Znając siły działające na telefon w dwóch różnych, następujących po sobie momentach, oraz różnicę czasu pomiędzy ich odczytami, da się obliczyć przyrost prędkości i przebytą w tym czasie drogę. Jest bardzo istotnym, że wykorzystując akcelerometr nie można bezpośrednio uzyskać prędkości lecz jedynie jej przyrost w czasie. Informację tę można jednak obliczyć, zakładając, że znana jest prędkość ciała przy pierwszym odczycie danych z przyspieszeniomierza. Wystarczy wtedy zsumować prędkość początkową z kolejnymi jej przyrostami. Można więc z bardzo dużą częstotliwością uzyskiwać dane niezbędne do pomiaru przyśpieszenia samochodu. W tym celu należałoby w jednakowych odstępach czasu próbkować dane z akcelerometru. Pomiar musiałby się rozpocząć w momencie gdy samochód stoi w miejscu, dzięki czemu będzie znana jego prędkość początkowa. Zakładając, że tor jazdy po którym przyśpiesza samochód to linia prosta oraz, że telefon znajdujący się wewnątrz samochodu jest stabilnie przymocowany w sposób uniemożliwiający jego poruszenie, można obliczyć przyśpieszenie wypadkowe samochodu (wykluczając przyśpieszenie grawitacyjne). Oczekiwany podczas rozpędzania samochodu, przebieg zmian przyśpieszenia wypadkowego w odniesieniu do czasu, przedstawia poniższa ilustracja. Największa siła działa w trakcie przyśpieszania na pierwszym biegu. W górnym zakresie obrotów silnika, moment obrotowy maleje, wtedy następuje zmiana biegu. Każdy kolejny wyższy bieg, zmniejsza wartość przyśpieszenia, lecz umożliwia szybszą jazdę. Pole pod wykresem jest równe uzyskanej prędkości.



Ilustracja 15: Oczekiwany przebieg wartości przyśpieszenia w trakcie rozpędzania samochodu.

Rejestracja przyspieszenia samochodu z wykorzystaniem akcelerometru jest możliwa przy spełnieniu szeregu założeń, które z doświadczeń i analiz przeprowadzonych przez autora pracy okazały się w praktyce nieosiągalne. Badania ujawniły szereg problemów uniemożliwiających prawidłowe wykonanie takiego pomiaru. Pierwsze z nich są związane z samą dokładnością czujnika przyśpieszenia, która została omówiona już wcześniej w rozdziale drugim. Głównym problemem jest sposób eliminacji zewnętrznych sił działających na urządzenie. Stabilne zamocowanie telefonu wewnątrz pojazdu zapobiega jego przemieszczaniu w trakcie jazdy. Jednak nie we wszystkich samochodach można pozbyć się drgań i wibracji pochodzących z silnika samochodu. Nie można także wyeliminować sił powstałych na wskutek jazdy po nierównej drodze. Wjechanie nawet w niewielką dziurę czy też nierówność jezdni, skutkuje pojawieniem się chwilowej, stosunkowo dużej siły działającej na cały samochód, w tym także na telefon i jego czujnik. Może się wydawać, że tak powstałe siły są niewielkie i mało znaczące. Rzeczywiście są one małe. Jednak gdy porówna się je z wartościami sił działających na przyspieszający samochód, okaże się, że wprowadzają one bardzo duży błąd. W poniższej tabeli zaprezentowane są wyniki przeprowadzonej analizy z jazdy po różnych nawierzchniach z prędkością 50 km/h, mierząc wartości dodatkowych zewnętrznych sił powstałych na jej skutek.

Typ nawierzchni	Wartość wytwarzanej siły
Równa droga	0 – 0.05 g
Niewielkie nierówności	0.05 – 0.15 g
Średnie nierówności	0.15 – 0.35 g
Niewielka dziura	0.35 – 0.60 g
Duża dziura	0.60 – 1.20 g
Bardzo duża dziura	Powyżej 1.20 g

Tabela 1: Wyniki analizy przyśpieszeń powstałych na wskutek jazdy po różnych nawierzchniach z prędkością 50km/h.

Okazuje się, że na zwykły samochód w trakcie gwałtownego przyśpieszania, rzadko kiedy działa większa siła niż 0.5 g. Dla kontrastu, najszybszy na świecie, dopuszczony do ruchu samochód osobowy o mocy 1001 koni mechanicznych, rozpędza się do prędkości 100km/h w ciągu 2.4 sekundy, osiągając przyśpieszenie zaledwie 1.55 g. Taką siłę, można bez problemu wytworzyć poprzez lekkie ale zdecydowane potrząśnięcie telefonu. Inne przykładowe wartości sił przedstawione są w poniższej tabeli [28]. Widać więc, że jazda po nierównościach oraz inne źródła dodatkowych sił i wibracji wprowadzają bardzo duże błędy w stosunku do wartości przyśpieszenia generowanego przez silnik samochodu. To wszystko sprawia, że akcelerometr bardziej nadaje się do pomiaru ilości dziur i nierówności w drodze, niż do pomiaru przyśpieszenia samochodu.

Przykład sytuacji	Wartość działającej siły
Siła przyciągania ziemskiego	1 g
Potrząśnięcie telefonu	~1.2 g
Kolejka górska	3.5 – 6.3 g
Ostre hamowanie bolidu formuły 1	~5 g
Szybka jazda bolidem formuły 1 po ostrym zakręcie	5.2 g

Tabela 2: Wartości sił oddziaływających w przykładowych sytuacjach.

3.2.8.2 Implementacja pomiaru przyśpieszenia z wykorzystaniem modułu GPS

Wykorzystanie systemu GPS jest obecnie najlepszą i zarazem najprostszą metodą do pomiaru przyśpieszenia samochodu za pomocą urządzenia mobilnego. Jego dokładność zależy od jakości użytego w tym celu modułu GPS. Pomiar dokonywany jest przez aktywność

ActivityPomiarPrzyspieszenia. Aby uzyskać dostęp do systemowej usługi lokalizacji należy dodać do manifestu aplikacji pozwolenie – *ACCESS_FINE_LOCATION*. Usługa jest reprezentowana przez obiekt *LocationManager,* który obsługuje wszystkie trzy rodzaje geolokalizacji, opisane w rozdziale drugim. Referencję do tego obiektu uzyskuje się wywołując metodę *getSystemService(Context.LOCATION_SERVICE)* odziedziczoną po klasie *Activity*. Odczytywanie danych z usługi lokalizacji odbywa się poprzez odpowiednio zdefiniowany przez programistę obiekt nasłuchiwacza, implementującego interfejs *LocationListener*. Każdy nasłuchiwacz zanim zacznie otrzymywać dane o pozycji, musi zostać zarejestrowany w usłudze lokalizacji. W momencie gdy współrzędne użytkownika nie są już potrzebne, należy wyrejestrować nasłuchiwacz. W przeciwnym razie, usługa lokalizacji będzie ciągle aktywna, co może doprowadzić do szybkiego rozładowania baterii urządzenia. [18]

Klasa przechowuje obiekt usługi lokalizacji w polu *locationManager* oraz obiekt nasłuchiwacza w zmiennej o nazwie *locationListener*. Zmienne *Location loc1* oraz *Location loc2* zawierają dwie ostatnio odczytane współrzędne, zaś pola *t1* i *t2* zapamiętują czas ich odczytu. Zmienna *loc2* jest ostatnio odczytaną pozycją, zaś *loc1* jest jej poprzedniczką. W polach *Vcur* oraz *Vprev* przechowywane są obecna i poprzednia prędkość samochodu.

Listing 16: Implementacja klasy ActivityPomiarPrzyspieszenia. Główne pola klasy.

```
// Pola widoku
private Licznik licznik;
...

// Zmienne do obliczen
private static final int PREDKOSC_POMIARU = 100; // Okresla koncowa predkosc pomiaru
private Location loc1 = null, loc2 = null;
private long t1, t2;
private double Vcur = 0, Vprev = 0;
private int iloscOdczytowGPS = 0;
private int iloscOdczytowGPSwPomiarze = 0;
private long czasPierwszegoOdczytuGPS;
private double czestotliwoscOdczytuGPS;
private boolean nagrywaj = false;
private boolean jestFix = false;

// Zmienne przechowujace nagranie
private List<Long> nagranieZnacznikCzasu = new ArrayList<Long>();
private List<Double> nagranieDroga = new ArrayList<Double>();
private List<Double> nagraniePredkosc = new ArrayList<Double>();
private long timeStart;
private double dlugoscPrzejazdu;
private double uzyskanyCzas;
private double dokladnoscPomiaru = 0.0; // Wyrazona w sekundach

// Location manager i listener
private LocationManager locationManager;
private LocationListener locationListener = new LocationListener() { ... }
```

Pola te wykorzystywane są w implementacji nasłuchiwacza w metodzie *onLocationChanged(Location location)*. Jest ona wywoływana za każdym razem gdy GPS odczyta nową współrzędną, która jest potem przekazywana w parametrze funkcji. Definiuje więc ona reakcję na zmianę pozycji. Na podstawie obecnej oraz poprzedniej prędkości, wykrywane są momenty ruszenia oraz zatrzymania samochodu. Ruszenie jest wykrywane gdy poprzednio odczytana prędkość jest mniejsza od 1 km/h, a obecna prędkość jest od tej wartości większa. Moment zatrzymania wykrywany jest wtedy, gdy obecna prędkość jest mniejsza niż 1 km/h, a poprzednia była od tej wartości większa. W momencie gdy samochód ruszy, rozpoczyna się pomiar przyspieszenia, poprzez ustawienie zmiennej *nagrywaj* na wartość *true* oraz zapamiętanie czasu startu w polu *timeStart*. Użytkownik jest o tym informowany poprzez animowaną mrugającą diodę, która jest włączana przez funkcję *uruchomAnimacje()* a zatrzymywana przez metodę *zatrzymajAnimacje()*. Od tego momentu w listach – *nagranieZnacznikCzasu, nagraniePredkosc* oraz *nagranieDroga,* zapamiętywane są dane o czasie kolejnych odczytów współrzędnych i prędkościach samochodu oraz przebytej drodze, obliczanej na podstawie odległości pomiędzy współrzędnymi *loc1* i *loc2*. Pomiar kończy się w jednym z dwóch przypadków – samochód się zatrzymał lub została osiągnięta prędkość 100km/h. W pierwszej sytuacji wszystkie zapamiętane w listach dane są porzucane. W przypadku gdy samochód osiągnie docelową prędkość, pomiar jest przerywany, następnie obliczany jest uzyskany czas pomiaru oraz wywoływana jest funkcja *zapiszPrzejazd()*, w której następuje zapis danych pomiaru w bazie danych. Program jest gotowy do kolejnego pomiaru od razu po zakończeniu poprzedniego przejazdu.

Listing 17: Implementacja klasy ActivityPomiarPrzyspieszenia. Funkcja nasłuchiwacza mierząca przyspieszenie samochodu.

```
private LocationListener locationListener = new LocationListener() {

    ...

    @Override
    // Metoda pobiera najnowszy odczyt i wylicza aktualna predkosc
    // z jaka porusza sie pojazd, przebyta droge, czestotliwosc odczytu z GPS,
    // aktualizuje licznik, wykrywa start pomiaru przyspieszenia
    public void onLocationChanged(Location location) {
        if( loc1 == null ) {
            // Pierwszy odczyt wspolrzednej
            loc1 = loc2 = location;
            iloscOdczytowGPS = 1;
            Vcur = Vprev = location.getSpeed();
            czasPierwszegoOdczytuGPS = System.currentTimeMillis();
            t1 = t2 = System.currentTimeMillis();
        } else {
            loc1 = loc2;
            loc2 = location;
            t1 = t2;
            t2 = System.currentTimeMillis();
```

```
            Vprev = Vcur;
            //obliczenie predkosci wraz ze zmiana jednostek na km\h (mnoznik * 3600)
            Vcur = ((int)( location.getSpeed() * 3.6 * 100 )) / (float)100.0;
            // Obliczam czestotliowsc odczytu z GPS
            iloscOdczytowGPS++;
            czestotliwoscOdczytuGPS = iloscOdczytowGPS /
                (float)( (System.currentTimeMillis() - czasPierwszegoOdczytuGPS) / 1000.0 );
            czestotliwoscOdczytuGPS = ((int)(czestotliwoscOdczytuGPS * 10)) / (float)10.0;

            //wykrywanie startu - gdy predkosc wieksza od 1km\h wtedy jest start
            if( Vprev < 1 && Vcur > 1 ) {
                nagrywaj = true;
                // Wstawiam wartosci pierwszego punktu nagrania
                nagranieZnacznikCzasu.add(t1);
                nagranieDroga.add(0.0);
                nagraniePredkosc.add(0.0);
                // Zapamietuje czas startu pomiaru
                timeStart = t2;
                // Uruchamiam diode sygnalizujaca zapis danych
                uruchomAnimacje();
            }

            //wykrywanie zatrzymania = gdy predkosc spadnie do 1km\h
            if( Vprev > 1 && Vcur < 1 ){
                nagrywaj = false;

                // Czyszcze dane z poprzedniego zapisu przejazdu
                nagranieZnacznikCzasu.clear();
                nagranieDroga.clear();
                nagraniePredkosc.clear();
                dlugoscPrzejazdu = 0;
                iloscOdczytowGPSwPomiarze = 0;

                // Zatrzymuje diode
                zatrzymajAnimacje();
            }
        }

        // Zapisuje dane przejazdu, sprawdzam czy osiagnalem porzadana predkosc
        if( nagrywaj ) {
            // Przechowuje dane
            iloscOdczytowGPSwPomiarze++;
            dlugoscPrzejazdu += loc2.distanceTo(loc1);
            //nagranieZnacznikCzasu.add(location.getTime());
            nagranieZnacznikCzasu.add(t2);
            nagranieDroga.add(dlugoscPrzejazdu);
            nagraniePredkosc.add(Vcur);


            if( Vcur >= PREDKOSC_POMIARU ) {
               // Koncze nagrywac - uzyskalem porzadana predkosc
               nagrywaj = false;
               // Zatrzymuje diode
               zatrzymajAnimacje();
               // Obliczam uzyskany czas i dokladnosc
               uzyskanyCzas = ( t2 - timeStart ) / 1000.0;
               dokladnoscPomiaru = iloscOdczytowGPSwPomiarze / ( (t2 - timeStart) / 1000.0 );
               dokladnoscPomiaru = ((int)(100 * (1 / dokladnoscPomiaru))) / 100.0;
               // Zapisuje dane przejazdu do bazy danych
               zapiszPrzejazd();
            }

        }

        // Aktualizacja widoku
        updateView();
    }
};
```

Na samym końcu metody *onLocationChanged* wywoływana jest funkcja *updateView()* w której następuje odświeżenie widoku użytkownika. Aktualizowana jest wskazywana wartość

graficznego licznika przechowywanego w polu *Licznik licznik*, oraz częstotliwość odczytu z GPS.

3.2.8.3 Dokładność pomiaru przyśpieszenia za pomocą GPS

Przeprowadzone przez autora pracy badania częstotliwości odczytu współrzędnych w różnych urządzeniach z systemem Android, wykazały, że większość wbudowanych modułów GPS pracuje z częstotliwością odczytu 1Hz. W rozdziale drugim zostały opisane wady modułów pracujących z tą częstotliwością. Rejestrując przejazd, zliczana jest ilość odczytanych współrzędnych. Po dokonaniu pomiaru obliczana jest jego dokładność, która jest odwrotnością stosunku ilości odczytanych punktów do czasu przyspieszania. Inaczej mówiąc, jest ona odwrotnością częstotliwości odczytu z GPS uzyskanej podczas pomiaru. Częstotliwość odczytu nie jest stała, może się zmniejszać wskutek np. złych warunków pogodowych. Czas przyspieszenia, zmierzony za pomocą odbiornika GPS o częstotliwości odczytu 1Hz powoduje duży błąd, wynoszący +/- 1 sekundę. Tak więc pomiar wykonany przy pomocy takiego odbiornika jest dosyć niedokładny. Precyzję można znacząco zwiększyć wykorzystując zewnętrzny moduł GPS. Stosując odbiornik GPS pracujący z częstotliwością 5Hz uzyskuje się akceptowalny błąd pomiaru +/- 0.2 sekundy. Wpływ na precyzję wykonanego pomiaru ma także szereg innych czynników takich jak zła pogoda czy niewłaściwe umiejscowienie odbiornika. W takich sytuacjach zdarza się, że odczytywane współrzędne oraz informacje o prędkości są nieprawidłowe. Jest także istotne, aby informację o prędkości odczytywać bezpośrednio z odbiornika GPS. Służy do tego metoda *getSpeed()* wywoływana na rzecz obiektu pozycji *Location*. Posiadając dwie współrzędne geograficzne, oraz czas pomiędzy ich odczytami, można obliczyć prędkość. Nie będzie ona jednak dokładna. Powodem jest to, że współrzędne odczytywane z GPS są często niedokładne. Występuje też tak zwany efekt pływania pozycji, który polega na tym, że GPS odczytuje różne współrzędne geograficzne, nawet wtedy, gdy się nie porusza. Kolejne pozycje wskazują na punkty dookoła naszej prawdziwej lokalizacji, powodując efekt „pływania”. W takiej sytuacji, samodzielne obliczona prędkość, może wskazywać na to, że się poruszamy, gdy tak na prawdę stoimy w miejscu. Metoda *getSpeed()* pobiera prędkość obliczoną z wykorzystaniem efektu Dopplera, o czym była mowa w rozdziale drugim. Jest ona wartością dosyć dokładną. W trakcie używania programu, można zauważyć, że prędkość odczytywana z odbiornika GPS jest niższa od tej wskazywanej przez fizyczny licznik samochodu. Przy niższych prędkościach jest to mało zauważalne, lecz przy wyższych jest to już wyraźnie widoczne, ponieważ różnica rośnie wraz ze wzrostem szybkości. Nie jest to błąd programu

czy też systemu GPS. Okazuje się, że to właśnie prędkość wskazywana przez licznik w samochodzie, jest wartością niepoprawną. Zawyżenie faktycznej prędkości przez licznik w aucie, jest zabiegiem celowym, stosowanym przez wszystkich producentów samochodów. Dodatkowo wskazania prawdziwego prędkościomierza ulegają zmianie, zarówno w jedną jak i w drugą stronę, w zależności od tego, czy auto ma założone felgi i opony większe lub mniejsze od seryjnie stosowanych w danym modelu.

Ilustracja 16: Widok aktywności użytkownika związanych z pomiarem przyspieszenia.

3.2.9 Licznik samochodowy

Implementację funkcji licznika samochodowego zawiera klasa *ActivityLicznik*. Aktywność wyświetla graficzny licznik, informację o przebytej w trakcie jazdy drodze oraz o ilości dziur i nierówności w które wjechaliśmy w trakcie jazdy. Możliwa jest zmiana skali graficznego licznika, aby jak najbardziej przypominał prędkościomierz w naszym samochodzie. Prędkość oraz przebyta droga jest obliczana z danych uzyskanych z modułu GPS. Klasa *ActivityLicznik*, podobnie do aktywności dokonującej pomiar przyspieszenia, uzyskuje dostęp do obiektu *LocationManager* oraz rejestruje swój nasłuchiwacz *LocationListener*. Posiada też pola przechowujące informacje o dwóch ostatnio odczytanych współrzędnych – *Location loc1* oraz *Location loc2*. Wykorzystywane są one do pomiaru przebytej drogi, która musi być obliczona możliwie jak najdokładniej. W tym celu został zaimplementowany mechanizm odrzucania nieprawidłowych współrzędnych geograficznych. W przypadku pomiaru przyśpieszenia, nie można było sobie pozwolić na selekcję odczytanych pozycji, ponieważ trwa on zbyt krótko i

mogłoby to jeszcze bardziej powiększyć jego błąd. W funkcji licznika samochodowego, częstotliwość odczytu z GPS jest mniej istotna, nacisk jest położony na dokładność w obliczaniu przebytej drogi. W tym wypadku, można sobie pozwolić na porzucenie nawet kilkunastu nieprawidłowych współrzędnych.

3.2.9.1 Kontrola poprawności odczytanych współrzędnych geograficznych

Nieprawidłowo odczytane, niedokładne lub nieaktualne współrzędne są wykrywane w metodzie *isGoodLocation(Location location)* zdefiniowanej wewnątrz obiektu nasłuchiwacza usługi lokalizacyjnej. Funkcja jest wykorzystywana w metodzie nasłuchiwacza implementującej reakcję na zmianę pozycji. Jej wynik decyduje czy odczytaną pozycję uznać za prawidłową, czy też ją odrzucić. Poprawność najnowszej pozycji jest określana w odniesieniu do ostatniej, uznanej za wystarczająco dokładną współrzędnej. Jednak nigdy nie ma pewności, że jest ona rzeczywiście prawidłowa. Pierwszą odczytaną z odbiornika GPS współrzędną uznaje się za dokładną, ponieważ musi istnieć pozycja, do której będą przyrównywane kolejne współrzędne. Takie założenie jest jednak zgubne, ponieważ kilka pierwszych odczytów z GPS jest zawsze dosyć niedokładnych. Uznanie błędnej pozycji za dokładną, mogłoby spowodować odrzucenie szeregu pozostałych, dokładnych współrzędnych. Aby zabezpieczyć się przed taką sytuacją, został zdefiniowany maksymalny okres czasu, przez który pojedyncza współrzędna może być aktualna. Limit przechowuje stałe pole o nazwie *LOCATION_DELAY* zainicjowane wartością odpowiadającą pięciu sekundom. Każda odczytana współrzędna, która jest nowsza o czas *LOCATION_DELAY* w stosunku do poprzedniej prawidłowej pozycji, jest automatycznie uznawana za poprawną, niezależnie czy rzeczywiście jest dokładna czy też nie. Takie podejście sprawia, że na pewno w którymś momencie działania programu, odczytamy współrzędną która jest precyzyjna i w odniesieniu do niej, będą oceniane pozostałe pozycje. Pierwszym krokiem jest więc sprawdzenie, czy poprzednia lokalizacja nie uległa przedawnieniu. Przy okazji obliczana jest różnica w czasie odczytania wcześniejszej prawidłowej i najnowszej pozycji. Następnie obliczana jest wartość przyspieszenia, która powstałby przy przemieszczeniu się z jednej pozycji do drugiej. Jeżeli obliczona wartość będzie większa niż 1.5 g, wtedy lokalizacja jest uznawana za nieprawidłową. Zwykły samochód osobowy, nie jest w stanie osiągnąć takiego przyśpieszenia w trakcie jazdy. W następnym kroku porównywana jest dokładność dwóch współrzędnych, wykorzystując metodę *getAccuracy()* na rzecz obiektów *Location* przechowujących pozycję. Metoda ta, zwraca dokładność odczytanej współrzędnej wyrażoną w metrach. Liczba ta nie

zawsze jest dokładna, jest też zależna od konkretnego urządzenia mobilnego. Jeżeli poprzednio uznana za prawidłową pozycja nie uległa jeszcze przedawnieniu, ale ostatnio odczytana lokalizacja posiada lepszą dokładność, wtedy zostaje ona uznana za aktualną. Implementacja funkcji jest zaprezentowana na poniższym listingu. Selekcja prawidłowych współrzędnych umożliwia dokładny pomiar przebytej odległości. [29]

Listing 18: Implementacja klasy ActivityLicznik. Funkcja odrzucająca nieprawidłowo odczytane pozycje.

```
private static final int LOCATION_DELAY = 1000 * 5; //5 sekund

/*
 * Sprawdza czy nowo odczytana lokacja jest wystarczajaco dobra,
 * aby ja zapisac i wykorzystac pozniej do obliczania przebytej
 * drogi. Lokacja z parametru jest porownywana z polem loc2 listenera
*/
protected boolean isGoodLocation(Location location) {
    // Sprawdzam czy lokacja jest o wiele nowsza od ostatniej
    long timeDelta = location.getTime() - loc2.getTime();
    boolean isSignificantlyNewer = timeDelta > LOCATION_DELAY;
    boolean isNewer = timeDelta > 0;

    // Jezeli lokacja jest nowsza niz LOCATION_DELAY, to ja biore pod uwage,
    // poniewaz na pewno w tym czasie nastapil ruch i trzeba go uwzglednic
    if (isSignificantlyNewer) {
         return true;
    }

    // Obliczam wartosc przyspieszenia potrzebnego do przebycia drogi
    // od ostatniej lokacji do aktyalnie odczytanej, jezeli liczba
    // ta bedzie zbyt duza, to nie przyjmuje tej lokacji, gdyz moze
    // to byc nieprawidlowy odczyt z gps
    double acceleration = Math.abs( loc2.getSpeed() - location.getSpeed() ) /
                          ( (location.getTime() - loc2.getTime()) / 1000.0 );

    // Odrzucam lokacje jezeli przyspieszenie wynioslo ponad 15 m/s^2
    if( acceleration > 15 ) {
          return false;
    }

    // Sprawdza czy fix jest bardziej lub mniej dokladny
    int accuracyDelta = (int) (location.getAccuracy() - loc2.getAccuracy());
    boolean isLessAccurate = accuracyDelta > 0;
    boolean isMoreAccurate = accuracyDelta < 0;
    boolean isSignificantlyLessAccurate = accuracyDelta > 100;

    // Okreslenie jakosci pozycji na podstawie czasu i dokladnosci
    if (isMoreAccurate) {
        return true;
    } else if (isNewer && !isLessAccurate) {
        return true;
    } else if (isNewer && !isSignificantlyLessAccurate) {
        return true;
    }
    return false;
}
```

Odległość pomiędzy dwiema współrzędnymi *loc1* oraz *loc2*, reprezentowanymi przez obiekt *Location*, można szybko policzyć wywołując metodę – *loc1.distanceTo(loc2)*. Zwraca ona dystans wyrażony w metrach. Warto wiedzieć, że obliczenie jest dokonywane poprzez przybliżenie kształtu Ziemskiej geoidy, za pomocą formuły Haversine'a. Obliczenia

przybliżające kształt Ziemi do sfery są obarczone dużym błędem. W jednym z kiedyś przeprowadzonych przez autora pracy doświadczeń, wynikło, że taka metoda pomiaru odległości może powodować niedokładność rzędu 40 km przy trasie przebiegającej z północy na południe Irlandii, o rzeczywistej odległości 220 km. Formuła Haversine'a jest dokładna, dlatego jest powszechnie wykorzystywana.

3.2.9.2 Wykorzystanie akcelerometru do pomiaru ilości dziur w jezdni

Kolejnym zadaniem realizowanym przez klasę *ActivityLicznik* jest zliczanie ilości dziur i nierówności, na które najechaliśmy podczas jazdy samochodem. Do tego celu wykorzystywany jest akcelerometr, który został zaprogramowany tak, aby wyliczał przeciążenia działające na telefon. Aby skorzystać z przyśpieszeniomierza, należy uzyskać dostęp do obiektu systemowej usługi zarządzającej czujnikami – *SensorManager*. Nadzoruje ona pracę wszystkich dostępnych w urządzeniu sensorów. Wywołując metodę *getDefaultSensor* na rzecz obiektu usługi, otrzymuje się referencję do obiektu reprezentującego konkretny czujnik urządzenia. Aby odczytywać z niego dane, należy zdefiniować i zarejestrować nasłuchiwacz implementujący interfejs *SensorEventListener*. Należy pamiętać o tym, aby wyrejestrować nasłuchiwacz w momencie gdy dane z czujnika nie są już potrzebne. W przeciwnym wypadku może to doprowadzić do szybkiego rozładowania baterii urządzenia. Rejestracją i wyrejestrowywaniem nasłuchiwaczy, zamują się metody *zarejestrujListenery()* oraz *wyrejestrujListenery(),* wywoływane w odpowiednich momentach cyklu życia aktywności. Implementacja nasłuchiwacza akcelerometru zaprezentowana jest na poniższym listingu. Reprezentuje go obiekt przechowywany w polu o nazwie *sensorEventListener*. Zmienna *gPower* przechowuje ostatni odczyt siły przeciążenia a pole *ostatniOdczytSensora* przechowuje czas jego pobrania. Zmienna *sensorManager* przechowuje referencję do obiektu usługi zarządzającej czujnikami i jest uzyskiwana w konstruktorze klasy *ActivityLicznik*. Pole *gPowerDelta* określa wartość siły, której powstanie uznawane jest za najechanie na nierówność. Domyślnie przypisana jest wartość 0.5 g która odpowiada wjechaniu w średniej wielkości dziurę. W obiekcie nasłuchiwacza należy zdefiniować metodę *onSensorChanged(SensorEvent event)* która jest wywoływana za każdym razem gdy pojawią się nowe dane z akcelerometru. Przekazana w parametrze zmienna *event* przechowuje wartości sił oddziałujących na każdej z osi w przestrzeni trójwymiarowej. Aby obliczyć siłę przeciążenia, należy policzyć wartość przyśpieszenia wypadkowego oraz wykluczyć z niej siłę przyciągania ziemskiego, która jest zdefiniowana w jako stała

STANDARD_GRAVITY w klasie *SensorManager [18]*. Gdy obliczona siła przeciążenia przekracza wartość zdefiniowaną w zmiennej *gPowerDelta*, zwiększana jest zmienna *iloscDziur* oraz uruchomiona zostaje animacja sygnalizująca użytkownikowi fakt wjechania w dziurę. Wystąpienia sił powstałych na skutek jazdy po nierównościach, są odczytywane co 300 ms, co zapobiega kilkukrotnemu zliczeniu tej samej dziury. Dodatkowo, dziury są wykrywane tylko podczas jazdy. Aby pomiar ilości dziur był dokładny, urządzenie musi być stabilnie zamocowane wewnątrz samochodu w sposób uniemożliwiający jego poruszenie.

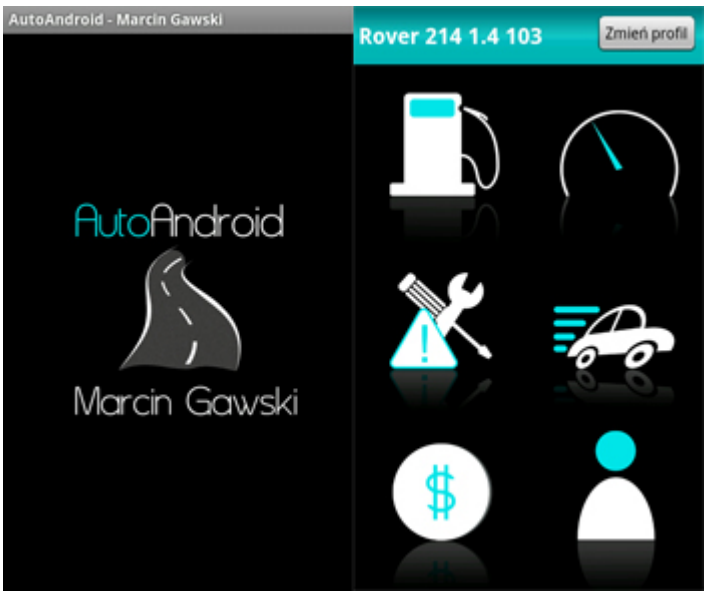
Listing 19: Implementacja klasy ActivityLicznik. Metoda wykrywająca ilość dziur w jezdni.

```
// Obsluga akcelerometru
private double gPower = 0;
private double gPowerDelta = 0.5;
private int iloscDziur = 0;
private long ostatniOdczytSensora = 0;
private SensorManager sensorManager;
private SensorEventListener sensorEventListener = new SensorEventListener() {
    @Override
    public void onAccuracyChanged(Sensor sensor, int accuracy) {}
    /*
     * Oblicza przeciazenia dzialajace na urzadzenie
     */
    @Override
    public void onSensorChanged(SensorEvent event) {
        double x = event.values[0];
        double y = event.values[1];
        double z = event.values[2];
        double g = Math.round(Math.sqrt(Math.pow(x, 2) + Math.pow(y, 2) + Math.pow(z, 2)));
        // Wykluczam z obliczen sile grawitacji
        gPower = Math.abs((g - SensorManager.STANDARD_GRAVITY)) /
                 SensorManager.STANDARD_GRAVITY;
        gPower = (int)(gPower * 1000) / 1000.0;

        if(System.currentTimeMillis() - ostatniOdczytSensora > 300 &&
           gPower >= gPowerDelta && aktualnaPredkosc > 1)
        {
            ostatniOdczytSensora = System.currentTimeMillis();
            iloscDziur++;
            iloscDziurValTextView.setText(iloscDziur+"");

            zatrzymajAnimacje();
            uruchomAnimacje();
        }
    }
};
```

Ilustracja 17: Widok aktywności ActivityLicznik.

3.2.10 Start programu i menu główne

W pierwszej kolejności, zaraz po uruchomieniu aplikacji, wyświetlana jest aktywność *ActivityInitApplication*. Wyświetla ona grafikę z informacją o nazwie programu i jego autorze. Nie jest to jednak jej jedyne zadanie. Przy pierwszym uruchomieniu programu, tworzona jest w niej baza danych. Po krótkiej chwili aktywność automatycznie kończy swoje działanie, tworząc intencję uruchamiającą główną aktywność programu – *MainActivity*. Tworzy ona proste menu, umożliwiające dostęp do wszystkich funkcjonalności programu. W górnej części jej widoku, znajduje się belka wyświetlająca nazwę aktualnie załadowanego profilu oraz przycisk umożliwiający jego szybką zmianę. Widok tych dwóch aktywności jest przedstawiony na poniższej ilustracji.



Ilustracja 18: Widoki początkowe programu AutoAndroid.

3.3 Możliwości rozbudowy programu AutoAndroid

Program AutoAndroid stanowi solidną podstawę do późniejszej rozbudowy. Obecne funkcjonalności można rozszerzyć lub dodać całkowicie nowe. Przykładowo, w raportach spalania warto dodać podział na raporty spalań na trasie, w mieście lub w cyklu mieszanym. Ciekawą funkcją byłoby wprowadzenie rozszerzeń pozwalających dodawać różne komunikaty na portalach społecznościowych. Na przykład, użytkownik wirtualnej społeczności mógłby podzielić się ze znajomymi swoim nowym najlepszym czasem pomiaru przyśpieszenia, czy też uzyskaniem niskiego spalania. Można by także pokusić się o przeniesienie bazy danych z urządzenia na zewnętrzny serwer. Byłoby wtedy możliwe, wprowadzenie funkcjonalności umożliwiającej porównywanie osiągów samochodów, czy też ich spalań, pomiędzy profilami użytkowników z całego świata. Co więcej, tak przygotowana baza danych, mogłaby później okazać się źródłem cennych informacji na temat różnych samochodów.

Zakończenie

W dzisiejszych czasach telefony nie są już zwykłymi urządzeniami do prowadzenia rozmów. Przypominają bardziej zaawansowane komputery o bardzo małych rozmiarach. Obecnie większość komputerów na świecie stanowią właśnie zaawansowane urządzenia mobilne. Wykorzystywane są niemal w każdej dziedzinie życia. Firma Google wypuszczając system Android stała się liderem na rynku i to ona obecnie wyznacza kierunek rozwoju urządzeń mobilnych. Program stanowiący część tej pracy, wykorzystuje możliwości jakie daje system Android. Prezentuje także, w jaki sposób można praktycznie wykorzystać wbudowane czujniki przyśpieszenia oraz moduł GPS. Aplikacja jest w pełni funkcjonalna i gotowa do codziennego użytku.

Bibliografia

1: Wikipedia, http://pl.wikipedia.org/wiki/Telefon_kom%C3%B3rkowy, 21.06.2011

2: Wikipedia, http://pl.wikipedia.org/wiki/Laptop, 21.06.2011

3: Agile Mobility, http://agilemobility.net/2009/04/the-history-of-personal-digital-assistants1/, 21.06.2011

4: Wikipedia, http://en.wikipedia.org/wiki/PDA, 21.06.2011

5: Wikipedia, http://en.wikipedia.org/wiki/BlackBerry, 21.06.2011

6: Wikipedia, http://en.wikipedia.org/wiki/Palm_Treo, 21.06.2011

7: Wikipedia, http://en.wikipedia.org/wiki/IPAQ, 21.06.2011

8: Wikipedia, http://en.wikipedia.org/wiki/List_of_HTC_phones, 21.06.2011

9: Garmin, https://buy.garmin.com/shop/shop.do?pID=177, 21.06.2011

10: Wikipedia, http://en.wikipedia.org/wiki/HTC_Corporation, 21.06.2011

11: Wikipedia, http://pl.wikipedia.org/wiki/Motorola_DynaTAC, 21.06.2011

12: Wikipedia, http://en.wikipedia.org/wiki/Mobile_phone, 21.06.2011

13: mobile88, http://www.mobile88.com/cellphone/Samsung/Samsung-SGH-M100/preview.asp, 21.06.2011

14: Wikipedia, http://en.wikipedia.org/wiki/IPhone, 21.06.2011

15: Wikipedia, http://en.wikipedia.org/wiki/Android_%28operating_system%29, 21.06.2011

16: Wikipedia, http://pl.wikipedia.org/wiki/HTC_Dream, 21.06.2011

17: Wirtualne Media, http://www.wirtualnemedia.pl/artykul/w-2015-r-sprzedanych-zostanie-prawie-miliard-smartfonow-w-b-r-wzrost-o-55-proc#, 21.06.2011

18: Reto Meiser, Professional Android 2 Application Development, 2010, strony: 245-253, 457-475

19: Wikipedia, http://pl.wikipedia.org/wiki/MCC, 21.06.2011

20: Wikipedia, http://pl.wikipedia.org/wiki/MNC, 21.06.2011

21: Wikipedia, http://pl.wikipedia.org/wiki/LAC, 21.06.2011

22: Wikipedia, http://pl.wikipedia.org/wiki/Cell_Identifier, 21.06.2011

23: Wikipedia, http://en.wikipedia.org/wiki/Global_Positioning_System, 21.06.2011

24: Navigation Center, NAVSTAR GPSUSER EQUIPMENT INTRODUCTION, 1996, strony: 12-17, 51-52

25: Skyhook, http://www.skyhookwireless.com/howitworks/coverage.php, 21.06.2011

26: Sayed Hashimi, Satya Komatineni, Dave MacLean, Android 2 Tworzenie aplikacji, 2010,

strony: 49-50, 68-71
27: Android Developers, http://developer.android.com/reference/android/app/Activity.html, 21.06.2011
28: Wikipedia, http://en.wikipedia.org/wiki/G-force, 21.06.2011
29: Android Developers, http://developer.android.com/guide/topics/location/obtaining-user-location.html, 21.06.2011

Indeks ilustracji