Uniwersytet Warszawski
Wydział Matematyki, Informatyki i Mechaniki

Łukasz Kidziński
Nr albumu: 234151

Wykorzystanie języka Objective-C w tworzeniu aplikacji mobilnych na przykładzie aplikacji społecznościowej

Praca magisterska
na kierunku INFORMATYKA

Praca wykonana pod kierunkiem
dra Janusz Jabłonowski
Instytut Informatyki

Grudzień 2010

Oświadczenie kierującego pracą

Potwierdzam, że niniejsza praca została przygotowana pod moim kierunkiem i kwalifikuje się do przedstawienia jej w postępowaniu o nadanie tytułu zawodowego.

Data Podpis kierującego pracą

Oświadczenie autora (autorów) pracy

Świadom odpowiedzialności prawnej oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie i nie zawiera treści uzyskanych w sposób niezgodny z obowiązującymi przepisami.

Oświadczam również, że przedstawiona praca nie była wcześniej przedmiotem procedur związanych z uzyskaniem tytułu zawodowego w wyższej uczelni.

Oświadczam ponadto, że niniejsza wersja pracy jest identyczna z załączoną wersją elektroniczną.

Data Podpis autora (autorów) pracy

Streszczenie

W pracy omówiono język Objective-C ze szczególnym uwzględnieniem jego roli w programowaniu aplikacji na platformę iPhone. Wskazano elementy wyróżniające tę technologię na tle innych platform mobilnych takich jak Java i Android. W celu prezentacji i analizy możliwości platformy, w ramach pracy magisterskiej powstała również aplikacja mobilna pozwalająca na obsługę serwisu społecznościowego.

Słowa kluczowe

Objective-C, tworzenie aplikacji na telefony komórkowe, iPhoneOS, wzorce projektowe

Dziedzina pracy (kody wg programu Socrates-Erasmus)

11.3 Informatyka

Klasyfikacja tematyczna

D. Software
D.3. Programming Languages
D.3.3. Language Constructs and Features

Tytuł pracy w języku angielskim

Application of the language Objective-C in mobile development, case study

Spis treści

Wprowadzenie

W pracy zaprezentowałem aktualne koncepcje z zakresu inżynierii tworzenia aplikacji mobilnych. Przedstawiłem mocne i słabe strony języka Objective-C w realizacji wzorców projektowych. W celu empirycznej analizy procesu powstawania aplikacji zaimplementowałem prosty program dla platformy iPhone OS umożliwiający wyszukanie najciekawszych lokali usługowych w okolicy użytkownika.

Aplikacja ta wykorzystuje najpopularniejsze funkcjonalności iPhone SDK takie jak: GPS, widok map, połączenie internetowe, widok zakładek, kontroler nawigacji i widoki tabelek. Dzięki temu oraz dzięki swojej prostocie dobrze obrazuje ona mechanizmy wykorzystywane zarówno w procesie produkcji oprogramowania, jak i w czasie wykonania aplikacji. Motywacją do stworzenia takiej aplikacji była moja współpraca przy projekcie dood.pl - wyszukiwarce lokalnych usług i firm.

W rozdziale 1. omówione zostały podstawowe cechy aplikacji mobilnych - historia ich rozwoju oraz krótka charakterystyka najpopularniejszych systemów. W rozdziale 2. skupiłem się na prezentacji i analizie samego języka Objective-C, by następnie w rodziale 3. pokazać jego wady i zalety w realizacji wzorców przydatnych przy tworzeniu aplikacji mobilnych.

Kolejne dwa rozdziały stanowią opis realizacji aplikacji. Rozdział 4. jest wprowadzeniem do platformy, dla której aplikacja została przygotowana, a rozdział 5. jest opisem samej implementacji. Program został stworzony zgodnie z podstawami metodologii projektowania aplikacji - wyszedłem od opisania celów aplikacji i rozbiciu ich na grupy. Następnie opisane zostały przypadki użycia, widoki i modele oraz ogólna specyfikacja protokołu komunikacji z serwerem. Rozdział 6. stanowi podsumowanie i wnioski z części teoretycznej i implementacyjnej.

Rozdział 1

Aplikacje mobilne

Od wielu lat telefon komórkowy przestał służyć wyłącznie do dzwonienia. Już w latach 90tych pojawiły się pierwsze aparaty dające możliwość instalowania zewnętrznych aplikacji pobranych z internetu lub z komputera osobistego. Pierwsze aplikacje pisano natywnie na dany system operacyjny - w ten sposób rozpoczęła się era smartphone'ów z telefonem Nokia Communicator 9000 jako jej kluczowym reprezentantem.

Od roku 2000 pojawiły się także popularne aparaty z możliwością instalowania aplikacji uruchamianych na maszynie wirtualnej Javy. Zrewolucjonizowało to rynek aplikacji mobilnych, gdyż udało się w znacznym stopniu ujednolicić platformę i ta sama aplikacja mogła już działać na wielu aparatach różnych producentów.

Jednak do zastosowań biznesowych korzystano i wciąż korzysta się ze smartphonów, choć podział stopniowo się zaciera. Zaczęły się pojawiać i upowszechniać aparaty niebiznesowe o bardzo dużych możliwościach. Kamieniem milowym było pojawienie się iPhonea marki Apple - telefonu całkowicie pozbawionego klawiatury, którego większą część stanowi wyświetlacz. Według badań z lipca 2010 Apple iPhone stanowi ponad 25% światowego rynku smartphone'ów [MSH].

Jak się stosunkowo szybko okazało, o sukcesie zadecydowała błyskawicznie rosnąca liczba aplikacji tworzonych przez niezależne firmy i pojedynczych developerów. Firma Apple zrezygnowała z wprowadzania maszyny wirtualnej Java. Stworzyli system operacyjny iPhone OS, którego jedynym językiem programowania stał się Objective-C.

Nasuwa się pytanie jaki wpływ na liczbę tworzonych aplikacji miał język Objective-C? Czy ogromna liczba programów powstała jedynie na skutek agresywnego marketingu Apple i związaną z nim popularnością iPhonea, czy może sam język na tyle wspiera developerów, że tworzenie aplikacji stało się nieco łatwiejsze i szybsze? W niniejszej pracy przeanalizuję wady i zalety Objective-C pod kątem tworzenia aplikacji mobilnych.

1.1. Specyfika aplikacji

Aplikacje mobilne, z racji gabarytów urządzenia, są z reguły małe i proste. Z jednej strony telefony komórkowe i palmtopy mają bardzo ograniczone zasoby, a z drugiej użytkownik nakłada na aplikacje dodatkowe wymagania: chce, aby program uruchamiał się i działał szybko. Systemy mobilne są przeważnie jednozadaniowe.

1.2. Platformy i języki

Jest bardzo wiele platform, na które można tworzyć aplikacje mobilne i niestety większość z nich nie jest ze sobą zgodna. Dlatego firma planująca stworzenie oprogramowania mobilnego musi bardzo ostrożnie podjąć decyzję, które platformy będzie obsługiwać.

Do najważniejszych możemy zaliczyć Java ME. Jest to platforma najbardziej przenośna, oparta o maszynę wirtualną Javy, zaimplementowana na wielu współczesnych telefonach komórkowych. Mimo, że język i biblioteki są dobrze wyspecyfikowane, a aplikacje teoretycznie przenośne, to programista musi jednak zadbać o zgodność sprzętu. Przy tworzeniu programów trzeba wziąć pod uwagę różne parametry urządzeń, takie jak wielkości wyświetlacza, szybkość procesora, wielkość dostępnej pamięci, rodzaj sterowania urządzeniem (klawiatura, ekran dotykowy) oraz istnienie wbudowanych urządzeń takich jak GPS, aparat fotograficzny, akcelerometr, kompas, termometr itd.

Aktualnie do najpopularniejszych systemów (tj. obejmujących największą część rynku telefonów komórkowych) możemy zaliczyć:

- Symbian - zaprojektowany od początku pod kątem urządzeń mobilnych. Aktualnie urządzenia wyposażone w ten wielozadaniowy system operacyjny stanowią ponad 50% wszystkich telefonów komórkowych. Większość z nich posiada wirtualną maszynę Javy, choć możliwe jest również tworzenie aplikacji w języku C++ na konkretne urządzenia [COG].

- Android - system zaimplementowany przez konsorcjum Open Handset Alliance na czele którego stoją takie korporacje jak Google, HTC, Motorola i T-Mobile. Oparty jest o jądro linuxowe. Posiada wirtualną maszynę Javy, choć nie jest zgodny ze standardem Java ME. Do tworzenia aplikacji konieczne jest specjalne Android Java SDK.

- Windows CE - system operacyjny Microsoftu dla urządzeń mobilnych. Zaimplementowano tam .NET Compact Framework - platformę bazująca na standardowej wersji .NET. Umożliwia tworzenie aplikacji w językach .NET czyli m.in. C#, C++, Visual Basic.

- Palm OS - system opracowany dla palmtopów - popularny głównie na rynku USA.

Cechą wspólną powyższych systemów jest to, że urządzenia, które je wykorzystują, są produkowane przez różnych producentów. Np. producentami telefonów z systemem Symbian są m.in. Motorola, Nokia, HTC.

Platformami stworzonymi dla urządzeń pojedynczych producentów są:

- BlackBerry - system wspierający urządzenia firmy BlackBerry wyposażone w pełną klawiaturę QWERTY.

- iPhone OS - system operacyjny firmy Apple przeznaczony wyłącznie dla urządzeń iPhone. Umożliwia tworzenie aplikacji jedynie w języku Objective-C.

Podstawową zaletą powyższych dwóch platform jest zminimalizowanie problemów związanych z testowaniem aplikacji. O ile sprawdzenie poprawności programu w języku Java wiąże się z analizowaniem różnych platform przez developera, tutaj za zgodność odpowiada producent urządzenia. Jest to kolejny powód, przez który powstaje dużo aplikacji dla iPhonea.

1.3. Częste błędy

Po wybraniu platformy musimy mieć na uwadze inne problemy związane z tworzeniem aplikacji mobilnych. Zagadnieniami, na które powinniśmy zwrócić szczególną uwagę (za [ARC]), są:

- autentykacja i autoryzacja (scenariusze z tymczasowymi połączeniami, błędy synchronizacji, bezpieczeństwo przesyłania danych),
- spamiętywanie (ang. caching, zapisywanie niepotrzebnych danych na urządzeniach o ograniczonych zasobach, błędne poleganie na wiarygodności spamiętywanych danych),
- komunikacja (problemy ochrony wrażliwych danych, problemy ograniczonej łączności, niebranie pod uwagę kosztu połączenia, niebranie pod uwagę minimalizowania poboru mocy przy pracy na zasilaniu z baterii),
- konfiguracja (błędne odtwarzanie danych konfiguracyjnych na starcie i po powrocie z wyjątku, nieużywanie technik dostarczonych przez producenta),
- dostęp do danych (problemy z wydajnością bazy danych, niewykorzystywanie technologii dostarczonych przez producenta),
- własności urządzenia (niebranie pod uwagę rozmiaru urządzenia, mocy procesora i innych parametrów urządzenia),
- zarządzanie logami (zapominanie o logach zdalnych, zapominanie o ograniczonych zasobach, trudny dostęp do logów),
- synchronizacja (złe zarządzanie konfliktami, przerwaniami, zapominanie o różnych drogach synchronizacji: przez kabel i radiowo),
- interfejs użytkownika (niewykorzystywanie UI producenta, zapominanie o jednozadaniowości aplikacji mobilnych, brak wsparcia dla różnych rozmiarów wyświetlaczy i różnych kontrolerów, zapominanie o ograniczeniach API),
- walidacja (niesprawdzanie otrzymanych danych).

Mając na uwadze powyższe zagadnienia możemy przystąpić do projektowania aplikacji. W tym procesie pomagają nam przede wszystkim sprawdzone i dopracowywane na przestrzeni lat wzorce projektowe.

Większość wzorców znanych z aplikacji desktopowych czy serwerowych znalazło swoje zastosowanie w aplikacjach mobilnych. Zanim jednak przejdę do omawiania ich, postaram się krótko przedstawić język, z którego będę korzystać.

Rozdział 2

Język Objective-C

Język Objective-C to rozszerzenie języka C o możliwości obiektowe poprzez dodanie komunikatów w stylu Smalltalka. Aktualnie język jest wykorzystywany przede wszystkim na platformach Apple: Mac OS i iPhone OS. Jest głównym językiem Cocoa API - podstawowego API w systemach firmy Apple.

Język stworzyli Brad Cox i Tom Love w 1981 roku. W 1988 roku firma NeXT (której właścicielem był Steve Jobs, pozbawiony wówczas udziałów i stanowiska prezesa firmy Apple) zakupiła licencję na język Objective-C i wprowadzając go na swoje platformy stopniowo popularyzowała. W 1996 Apple wykupił firmę NeXT, a język Objective-C stopniowo wprowadził jako główne narzędzie programistyczne na systemach MacOS.

Współczesne Cocoa API bazuje głównie na elementach API firmy NeXT [NXT].

2.1. Objective-C jako nadzbiór C

Objective-C jest ścisłym nadzbiorem języka ANSI C [OPL]. W szczególności każdy program poprawny składniowo w C skompiluje się kompilatorem Objective-C. Dzięki temu wszelkie standardowe biblioteki C mogą być z powodzeniem wykorzystywane w programach Objective-C. Dodane funkcjonalności są niezależne od dialektu języka C - w ustawieniach kompilatora gcc możemy decydować, z której wersji ANSI C korzystamy [GCC].

2.2. Składnia

Składnia elementów dodanych w Objective-C bazuje przede wszystkim na języku Smalltalk. Wszystko co nie jest związane z obiektowością, ma składnię identyczną z C, a z kolei elementy obiektowe przypominają wysyłanie komunikatów w Smalltalku.

2.2.1. Komunikaty

Model obiektowości Objective-C oparty jest na przekazywaniu komunikatów do obiektów. W wielu popularnych językach programowania, takich jak C++, Java czy Object Pascal (lub bardziej ogólnie: w językach typu Simula) operacje na obiektach wykonujemy odwołując się do ich metod. Tymczasem w Objective-C projektanci języka wprowadzili wysyłanie komunikatów w stylu Smalltalka. Ważne jest, że różnica nie jest tylko na poziomie składniowym. W językach typu Simula w metodach niewirtualnych nazwa metody jest związana z implementacją w klasie przez kompilator. Tymczasem zarówno w Objective-C odpowiednia

implementacja zawsze jest wyszukiwana w czasie wykonania programu na podstawie nazwy komunikatu i klasy obiektu [OPG].

Metoda jest identyfikowana przez specjalny typ zwany selektorem - `SEL` i jest związana ze wskaźnikiem do implementacji zwanym `IMP`. Wiązanie następuje w czasie wykonania - po pierwszym wywołaniu selektora `SEL` zostaje on bezpośrednio związany z kodem, który implementuje wiadomość. System przekazywania wiadomości nie ma kontroli typów - obiekt, do którego zostaje wysłany komunikat, może nie mieć implementacji danej wiadomości i zgłosi wyjątek w czasie wykonania.

Składnia wysyłania komunikatu `method` z argumentem `argument` do obiektu `obj` wygląda następująco:

```
[obj method: argument];
```

i jest to odpowiednik wywołania metody (uwzględniając opisane powyżej różnice) z C++:

```
obj->method(argument);
```

Oba podejścia mają swoje wady i zalety. Obiektowość w stylu Simuli pozwala na szybsze wywołanie metod niewirtualnych dzięki wiązaniu w czasie kompilacji. Z kolei w Objective-C nie musimy nawet deklarować, że obiekt implementuje daną metodę - może się to zmienić w czasie wykonania.

W Objective-C komunikat może być wirtualny, tj. bez implementacji - klasy dziedziczące mogą go wtedy przedefiniować tak jak w przypadku wirtualnych metod w C++.

W związku z kosztem interpretacji pierwsze wywołanie komunikatu trwa około 3 razy dłużej niż w przypadku niewirtualnych metod w C++, przy czym kolejne wywołania (po związaniu metody ze wskaźnikiem do implementacji) są porównywalnie szybkie do C++ [CDI].

2.2.2. Interfejs i implementacja

Konwencja interfejsu i implementacji klasy jest podobna do tej znanej z języka C++. Pliki nagłówkowe (interfejs) tworzymy z rozszerzeniem .h, a implementację z rozszerzeniem .m („m” pierwotnie pochodziło od „messages”, gdyż podczas prezentowania języka Objective-C to właśnie komunikaty stanowiły jego centralny punkt [MEX]). Powszechną konwencją jest umieszczanie każdego interfejsu w osobnym pliku nazywając plik nazwą klasy. Przykładowa deklaracja interfejsu wygląda następująco:

```
@interface classname : superclassname {
    // Zmienne prywatne
}
// Metody klasowe
+classMethod1;
+(return_type)classMethod2;
+(return_type)classMethod3:(param1_type)param1_varName;

// Metody obiektu
-(return_type)instanceMethod1:(param1_type)param1_varName;
-(return_type)instanceMethod2:(param1_type)param1_varName
   andOtherParameter:(param2_type)param2_varName;
@end
```

Analogiczny kod w C++ wygląda tak:

```
class classname : public superclassname {
 private:
  // Zmienne prywatne

 public:
  // Metody klasowe
  static void classMethod1();
  static return_type classMethod2();
  static return_type classMethod3(param1_type param1_varName);

  // Metody obiektu
  return_type instanceMethod1(param1_type param1_varName);
  return_type instanceMethod2(param1_type param1_varName, param2_type
      param2_varName);
};
```

Warto zwrócić uwagę na `instanceMethod2:andOtherParameter:` z parametrem nazwanym, który nie ma swojego odpowiednika w C/C++. Ogólną konwencją jest nazywanie komunikatów tak, by podczas wywołania było widać jaka jest rola poszczególnych parametrów, np.:

```
[screen changeColorToRed:0.2 green:0.5 blue:0.7];
```

którego odpowiednikiem w C++ jest

```
screen->changeColor(0.2, 0.5, 0.7);
```

gdzie nie widać do czego służą poszczególne parametry.

Typy wyników mogą być dowolnymi typami z C, wskaźnikami do obiektów lub generycznym typem `(id)`.

Zmienne obiektu Objective-C są prywatne i żeby uzyskać do nich dostęp musimy stworzyć odpowiednie akcesory.

Implementacja interfejsu zawarta jest w bloku `@implementation @end`. Np.:

```
@implementation classname
// Metody klasowe
+classMethod {
    // implementacja
return;
}
// Metody obiektu
-(int)method:(int)i {
    return [self add: i];
}
@end
```

Odpowiednikiem drugiej metody w C++ byłoby

```
int method(int i) {
    return this->add(i);
}
```

Składnia pozwala na przemianowanie parametrów. Np. komunikat zmieniający kolory może mieć następującą postać:

```
-(int)changeColorToRed:(float) green:(float) blue:(float)
```

a wtedy w treści metody korzystamy z nazw `changeColorToRed, green` i `blue`. Aby w ciele metody używać nazw `red, green, blue` musimy zadeklarować ją w następujący sposób:

```
-(int)changeColorToRed:(float)red green:(float)green blue:(float)blue
```

W obu przypadkach komunikat wywołujemy w ten sam sposób

```
[myColor changeColorToRed:5.0 green:2.0 blue:6.0];
```

2.2.3. Egzemplarze klas

Stworzenie egzemplarza zadeklarowanej wcześniej klasy wiąże się z przydzieleniem pamięci i wywołaniem konstruktora, co realizujemy poprzez wysłanie dwóch komunikatów `alloc` i `init`:

```
classname* o = [[classname alloc] init]
```

przy czym w Objective-C od wersji 2.0 można wykonać to jednym komunikatem `new`

```
classname* o = [classname new]
```

Komunikat `init` odpowiada domyślnemu kontruktorowi z języka C++. Może również zostać przedefiniowany. Wtedy z reguły ma następującą postać:

```
-(id) init {
    if ( self = [super init] )
 {
        var1 = value1;
        var2 = value2;
        // ...
    }
    return self;
}
```

Wynikiem komunikatu `[super init]` może być inny obiekt - przypisanie `self = [super init]` zapewnia nas, że będziemy działać w obrębie tego samego obiektu.

Możliwość przekazywania innego obiektu przez komunikat `init` uważana jest za dużą zaletę języka Objective-C [COC]. Mechanizm ten możemy wykorzystać m.in. w następujących sytuacjach:

- realizując wzorzec singleton (przekazujemy za każdym razem ten sam obiekt),
- realizując inne unikatowe obiekty (np. `[NSNumber numberWithInteger:0]` może dawać globalne zero),
- realizując wzorzec klastrów, tj. tworząc „nadklasę” agregującą klasy o podobnych własnościach (przykładem jest klasa `NSNumber`, którą możemy zainicjować zarówno liczbą całkowitą jak i zmiennoprzecinkową),
- tworząc klasy, których konstruktor decyduje o rozmiarze przydzielonej pamięci na podstawie otrzymanych parametrów (poprzez utworzenie odpowiedniego obiektu i przekazaniem go jako wynik `init`).

2.2.4. Protokoły

Protokoły realizują ideę wielokrotnego dziedziczenia specyfikacji, ale nie implementacji. W C++ ten wzorzec osiąga się poprzez wielokrotne dziedziczenie klas bazowych (abstrakcyjnych), a w C# i w Javie poprzez „interface".

Protokół to lista metod, które klasa musi zadeklarować i zaimplementować. Od wersji Objective-C 2.0 mogą one posiadać metody opcjonalne. Np. klasa `TextField` może mieć delegata implementującego protokół z opcjonalną metodą `autocomplete`. Obiekt `TextField` może sprawdzić, czy delegat implementuje `autocomplete` poprzez refleksję i jeśli tak, to wysyłać odpowiedni komunikat, by skorzystać z tej funkcjonalności [WPR].

Warto zwrócić uwagę na różnicę pomiędzy interfejsami z Javy, a protokołami z Objective-C, polegającą na tym, że w Objective-C klasa może implementować protokół, nawet jeśli nie zostało to zadeklarowane (w szczególności, wykorzystując kategorie, możemy nawet poprawić istniejącą już klasę tak, by zaczęła implementować jakiś protokół - szczegóły omówię w rozdziale 2.2.7).

2.2.5. Dynamiczne typy

Objective-C, podobnie jak Smalltalk, wspiera dynamiczne typowanie. Wynika to z opisanego wcześniej faktu, że metody nie są bezpośrednio związane z klasą [rozdział 2.2.1]. Tzn. mając obiekt nieznanego typu, możemy do niego wysłać dowolny komunikat. Na etapie kompilacji nie musimy znać typu obiektu, by wykonać operacje właściwe jego klasie. W przypadku wysłania nieodpowiedniego komunikatu zostanie zgłoszony wyjątek.

Możemy też przejąć komunikat w obiekcie przed przetworzeniem by, jeśli nie umiemy go obsłużyć, ani przekazać go dalej (ten wzorce nazywamy dalej delegacją). Jeśli obiekt nie umie obsłużyć komunikatu, ani przekazać go, ani nie zgłasza wyjątku, to komunikat jest ignorowany. Komunikaty przesłane do stałej `nil` są domyślnie ignorowane (chyba, że zmieniliśmy odpowiednią opcję kompilatora).

Kompilator Objective-C zapewnia kontrolę typów na etapie kompilacji i jeśli nie jest jasne czy dany obiekt może obsłużyć komunikat, to ostrzeże programistę o potencjalnym błędzie. Dla przykładu rozważmy następujące deklaracje

```
- feedMe:(id <Food>) foo; // metoda której argument dziedziczy po (id) i
                          // implementuje Food
- feedMe:(id) foo;
```

Jeśli dany obiekt `o` implemetuje protokół `Food` mimo, że go nie zadeklarował to

```
[dog feedMe: o]
```

nie będzie błędem w żadnych z dwóch przypadków, ale w pierwszym programista zostanie ostrzeżony.

Zaletą typowania dynamicznego jest np. możliwość prostego tworzenia kolekcji obiektów różnych typów. W językach typowanych statycznie, tworząc taką kolekcję musimy, przy dodawaniu obiektów, dbać o rzutowanie na typ abstrakcyjny i potem, przy pobieraniu, na rzeczywisty. Tymczasem rzutowanie burzy ideę statycznego typowania.

2.2.6. Przekierowanie i delegacja

Jedną z głównych cech języka Objective-C, która nie ma swojego bezpośredniego odpowiednika w C++ czy Javy, jest możliwość przekazywania komunikatów. Odbiorca, który nie umie

obsłużyć komunikatu, może przekazać go dalej. Ułatwia to implementowanie wzorców projektowych takich jak Obserwator, Pośrednik, Cel-Akcja i innych, o których napiszę w rozdziale 3.2.

Chcąc wykorzystać przekierowanie wystarczy przedefiniować metodę `forward`:

```
- (id) forward:(SEL) sel :(marg_list) args;
```

Przypominam, że typ `SEL` to typ komunikatów w Objective-C. Aby wykonać dany komunikat mając jego selektor `sel` wystarczy przesłać komunikat performv z danym selektorem i argumentami.

```
- (id) performv:(SEL) sel :(marg_list) args;
```

Możemy przedefiniować `performv`, przy czym jest to wykorzystywane stosunkowo rzadko, gdyż zadaniem tej metody jest jedynie „uruchomienie" żądanego komunikatu `sel`.

Poniższy przykład obrazujący delegację jest nieco zmodyfikowaną wersją obiektu dostawcy danych w zaimplementowanej przeze mnie aplikacji. Moduł pytający o dane zgłasza się do dostawcy, a ten w zależności od tego, czy jako źródło danych wybrany jest zewnętrzny serwer, czy lokalna baza danych, przesyła zapytanie dalej:

```
DataProvider.h
#import <objc/Object.h>

@interface DataProvider : Object
{
    id recipient; // Obiekt który będziemy pytać o dane
}

// Akcesory
- (id) recipient;
- (id) setRecipient:(id) _recipient;

@end
DataProvider.m
#import "DataProvider.h"

@implementation DataProvider

- (retval_t) forward: (SEL) sel : (arglist_t) args
{
     /*
     * Możemy sprawdzić czy odbiorca jest w stanie obsłużyć komunikat.
     * Taka funkcjonalność może się przydać np. gdy pytamy o najnowsze wiadomości,
     * których lokalna baza nie może nam udostępnić i nie implementuje
     * odpowiedniego komunikatu.
     */
    if([recipient respondsTo:sel])
        return [recipient performv: sel : args];
    else
        return [self error:"Nie masz dostępu do tych danych"];
}
```

```
- (id) setRecipient: (id) _recipient
{
    recipient = _recipient;
    return self;
}

- (id) recipient
{
    return recipient;
}
@end
Recipient.h
#import <objc/Object.h>

// Uproszczony obiekt InternetDataProvider.
@interface InternetDataProvider : Object
- (id) hello;
@end
InternetDataProvider.m
#import "InternetDataProvider.h"

@implementation InternetDataProvider

- (id) getNews
{
// tutaj implementujemy połączenie z serwerem
// ...
   // dalej zakładam, że wynik jest zapisany w zmiennej results
return results;
}

@end
main.m
#import "DataProvider.h"
#import "InternetDataProvider.h"

int main(void)
{
    DataProvider *forwarder = [DataProvider new];
    InternetDataProvider *recipient = [InternetDataProvider new];

    [forwarder setRecipient:recipient]; //Set the recipient.
    /*
     * Forwarder nie ma zaimplementowanego komunikatu getNews - będzie on
     * przekazany. Wszysykie metody nierozpoznawalne przez forwarder
     * zostaną przekazane do obiektu recipient.
     * (o ile recipient umie je obsłużyć - zgodnie z imlementacją DataProvidera)
     */
```

```
    [forwarder getNews];

    return 0;
}
```

2.2.7. Kategorie

Jednym z problemów, przed którym stanęli projektanci języka Objective-C, było zadbanie o ułatwienie kontroli nad dużymi fragmentami kodu, gdyż był to powszechny problem w przypadku programowania strukturalnego.

Kategorie to fragmenty klasy odpowiedzialne za określoną funkcjonalność. Przykładowo klasa `String` może mieć część odpowiedzialną za tłumaczenie tekstu. Co więcej metody kategorii dodawane są w czasie wykonania. W związku z tym metody mogą być dodane do istniejących klas bez rekompilacji i bez dostępu do źródła. Przykładem jest kategoria JSON, z której skorzystałem w aplikacji - dodaje ona do `NSString` komunikat `JSONValue`, który zamienia napis na odpowiednią tablicę lub słownik.

```
NSString* jsonString = @"{'name': 'Andrzej', 'age': 17}";
NSDictionary *dictionary = [jsonString JSONValue];
```

Metody w kategoriach są nierozróżnialne z metodami klasy - w szczególności mają dostęp do zmiennych prywatnych. Kategorie mogą też przedefiniowywać istniejące metody (np. mogą poprawić błędną implementację), przy czym jeśli dwie kategorie przedefiniujące tę samą metodę to specyfikacja języka nie określa, która z metod będzie użyta.

2.2.8. Własności

W Objective-C 2.0 wprowadzono nową składnię, pozwalającą deklarować zmienne obiektów jako własności (properties). Są to publiczne zmienne obiektu. Mogą być zadeklarowane jako „readonly" i można określić sposób zarządzania pamięcią jako „assign", „copy" i „retain". Dodatkowo określamy, czy zmienna jest „atomic" czy „nonatomic". Zmienne atomowe blokują równoległy dostęp. Deklaracja jako „nonatomic" zdejmuje tę blokadę.

Przykładowa definicja własności może wyglądać tak:

```
@interface Person : NSObject {
    @public
        NSString *name;
    @private
        int age;
}

@property(copy) NSString *name;
@property(readonly) int age;

-(id)initWithAge:(int)age;
@end
```

Automatyczną imlementację tworzymy z wykorzystaniem `@synthesize`. Jeśli sami chcemy implementować methody wykorzystujemy słowo kluczowe `@dynamic`. Dla przykładu powyżej, plik implementacji może wyglądać tak:

```
@implementation Person
@synthesize name;
@dynamic age;

-(id)initWithAge:(int)initAge
{
    age = initAge;
    return self;
}

-(int)age
{
    return 29;
}
@end
```

Do własności możemy się odwoływać używając standardowej składni komunikatów, za pomocą operatora kropki lub z wykorzystaniem metod "`valueForKey:`"/"`setValue:forKey:`"

2.3. Porównanie z C++

Powyższe własności języka świadczą o tym, że projektanci Objective-C mieli inne podejście do rozszerzenia C, niż projektanci C++.

W przeciwieństwie do C++ język Objective-C nie ma swoich „standardowych bibliotek", niemniej jednak z reguły w zastosowaniach dołączane są biblioteki w stylu „OpenStep" takie jak OPENSTEP, Cocoa czy GNUstep, które dodają funkcjonalności podobne do tych z bibliotek C++.

Kolejną rzeczą, która w zasadniczy sposób odróżnia Objective-C od C++ są refleksje i generalnie własności wykorzystywane i sprawdzane w czasie wykonania. C++ jest językim, w którym te funkcjonalności zostały ograniczone do minimum. W Objective-C obiekt może być pytany o swoje zmienne i o to, czy jest w stanie odpowiedzieć na jakiś komunikat. Jest to niezbędne tym bardziej, że w czasie kompilacji nie wiadomo, które komunikaty będą obsługiwane (przykładem są kategorie w rozdziale 2.2.7).

Oba języki różnią się również podejściem do typów uogólnionych. W C++ są one realizowane przez szablony (and. templates), a w Objective-C znów za pomocą dynamicznego typowania i refleksji. Np. kontener w Objective-C może przechowywać dowolne obiekty, więc nie są potrzebne różne kontenery dla różnych typów tak jak w przypadku C++.

Różnica między językami dynamicznymi i statycznymi jest zasadnicza i nie da się jednoznacznie określić, które podejście jest lepsze. Ocena zależy od kontekstu - w rozdziale 3.1 postaram się przedstawić wady i zalety obu podejść w kontekście tworzenia aplikacji mobilnych.

2.4. Efektywność języka

Ze względu na dynamiczne typowanie i rozwiązania „w czasie wykonania" programy napisane w języku Objective-C niestety tracą na efektywności. Wykorzystują więcej pamięci i są wolniejsze [PER]. Ze względu na dynamiczne typowanie nie można przeprowadzić niektórych optymalizacji wydajnościowych takich jak funkcje inline, optymalizacje interproceduralne (nie

można skracać pętli wysyłającej komunikaty, bo nie wiadomo kto odbierze komunikat), zastępowanie agregatów skalarami (tj. zamiana metod dających stałe na skalary).

Dużą zaletą jest całkowita zgodność z C, dzięki czemu kompilatory Objective-C bez problemu kompilują biblioteki C. Istnieją również aplikacje (tzw. wrappery) zamieniające biblioteki C na styl obiektowy.

Powszechnym problemem języka jest brak wsparcia dla przestrzeni nazw, w związku z czym komunikaty powinny mieć unikatowe identyfikatory. W rezultacie programiści są zmuszeni do używania prefixów, które z racji tego, że są z reguły krótsze i mniej mówią, mogą prowadzić do pomyłek.

Rozdział 3

Objective-C w kontekście aplikacji mobilnych

Jak wspomniałem wcześniej, Objective-C jest bezpośrednio związany z dwiema platformami, tj. z MacOS i iPhoneOS. Są to praktycznie jedyne dwa systemy, dla których aplikacje pisane są w tym języku, jednak warte są uwagi choćby ze względu na dynamicznie rosnącą popularność.

Postanowiłem zająć się platformą iPhoneOS i językiem Objective-C w kontekście tworzenia aplikacji mobilnych. Jako współautorowi aplikacji na inne platformy mobilne, daje mi to możliwość wykonania rzetelnych porównań. Aplikację nazwałem iDood nawiązując do portalu dood.pl.

Zacznę od omówienia fundamentalnej różnicy pomiędzy Objective-C, a językami takimi jak Java, C++ itp. tj. od jego dynamiczności. Następnie przejdę do komunikatów, delegacji i wynikających z nich korzyści przy realizowaniu wzorców projektowych.

3.1. Wady i zalety podejścia dynamicznego

Spór pomiędzy zwolennikami języków statycznych i dynamicznych trwa odkąd tylko pojawiły się te drugie. Takie cechy jak dynamiczne typowanie, refleksje i modyfikowanie klas w czasie wykonania, choć drażnią niektórych językowych purystów, na stałe znalazły swoje miejsce w wielu zastosowaniach. Języki takie jak PHP, Python czy Ruby odgrywają istotną rolę na rynku aplikacji internetowych. Za przełom można uznać wprowadzenie dynamicznego typowania w wersji 4.0 języka C#, gdzie wyszukiwanie metod obiektu zadeklarowanego jako `dynamic` wykonuje się w czasie wywołania.

Według Microsoftu [MEP] zwolennicy typowania statycznego agumentują swój wybór przede wszystkim:

- możliwością wczesnego wykrycia wielu błędów, tj. już na etapie kompilacji,
- możliwością prowadzenia bardziej precyzyjnej specyfikacji bibliotek, interfejsów i protokołów,
- większą liczbą optymalizacji przeprowadzanych przez kompilator,
- krótszym czasem wykonywania,
- ułatwieniami na etapie tworzenia kodu (IDE dla języków statycznych mogą lepiej „podpowiadać” kod).

Erik Meijer i Peter Drayton z firmy Microsoft twierdzą [MEP], że „Zwolennicy typowania dynamicznego uważają statyczne typowanie za zbyt sztywne. Z kolei fleksyjność języków dynamicznych sprawia, że idealnie nadają się do prototypów, w których stale zmieniają się wymagania lub do współdziałania z innymi systemami, które często się zmieniają".

Autorzy publikacji zwracają również uwagę na to, że dynamiczność języka odgrywa dużą rolę w zarządzaniu danymi.

Dobrym przykładem jest odczyt zewnętrznych danych w aplikacji iDood. Klient otrzymuje z serwera słownik w postaci napisu:

```
"{
   'name': 'Adam',
   'surname': 'Kowalski',
   'gender': 'M'
}";
```

W dalszej części zakładamy, że zmienna `NSString* resp` ma jako wartość powyższy napis. Dzięki dołączeniu do programu kategorii JSON (roz. 2.2.7) `NSString` obsługuje komunikat `JSONValue`

```
#import "JSON.h"
// ...
NSDictionary *jsonData = [[NSDictionary alloc] initWithDictionary :[resp JSONValue]];
```

Dzięki temu, że Objective-C umożliwia iterowanie przez zmienne obiektu klasy według ich nazw, możemy łatwo zainicjować obiekt klasy `User` danymi ze słownika `jsonData`:

```
User* user = [User new];
[user setValuesForKeysWithDictionary: jsonData];
```

3.2. Wzorce projektowe

Projektując dowolną aplikację staramy się projektować ją tak, by kod był przejrzysty i łatwy do zrozumienia przez inne osoby (które być może będą z niego korzystały w innych aplikacjach lub będą go rozbudowywały). Dlatego poszechnie korzystamy ze sprawdzonych schematów zwanych wzorcami projektowymi.

Według [ARC] wzorcami, które przede wszystkim warto brać pod uwagę projektując aplikację mobilną, są:

3.2.1. Lazy Acquisition

Cel: odraczanie pobierania danych tak długo, jak tylko jest to możliwe, w celu zoptymalizowania wykorzystania zasobów.

Urządzenia mobilne stosunkowo wolno pobierają dane z internetu. Wysłanie prostego zapytania HTTP wiąże się z nawiązaniem połączenia GPRS (które pociąga za sobą m.in. cały proces autoryzacji), co jest czaso- i zasobochłonne. Warto projektować aplikacje, tak by dane były pobierane z pamięci (tj. nie pobierać dwukrotnie tych samych informacji). Jeśli natomiast niezbędne jest połączenie z internetem, to czynność tę warto jak najdalej odłożyć.

Np. w iDood mimo, że dane lokali użytkownik otrzymuje już w wyszukiwarce, to informacje szczegółowe pobiera dopiero, gdy dojdzie do odpowiedniego widoku.

3.2.2. Active Object

Cel: wspieranie asynchronicznego przetwarzania komunikatów przez zgłaszanie zapytania i otrzymywanie odpowiedzi.

Jak już wcześniej wspomniałem, w przypadku aplikacji mobilnej bardzo istotne jest szybkie reagowanie. Dlatego np. po kliknięciu w szczegóły lokalu nie są pobierane jego dane, lecz najpierw zostaje zmieniony widok i wyświetla się ikona oczekiwania, a informacje odbierane są symultanicznie.

3.2.3. Communicator

Cel: zamykanie szczegółowej implementacji komunikacji w odrębnym module, który może wykorzystać różne kanały komunikacji.

Warto wydzielić funkcjonalność pobierania danych do modułu, który w przypadku braku aktywnego połączenia z internetem pobiera informacje z bazy danych. Jeśli z kolei połączenie jest nawiązane, to pobiera dane z serwera.

3.2.4. Entity Translator

Cel: obiekt zmieniający typy danych w obiekty biznesowe i odwrotnie w celu obsłużenia zapytań.

W iPhoneOS ten wzorzec jest realizowany przez obiekt `ManagedObjectContext`.

3.2.5. Reliable Sessions

Cel: wiarygodne przesyłanie wiadomości od nadawcy do adresata niezależnie od liczby i rodzaju pośredników.

Wzorzec nie jest wykorzystywany w iDood. Kontrola przepływających danych wiązałaby się z wprowadzeniem autentykacji i szyfrowania, co z kolei bardzo skomplikowałoby API.

3.2.6. Active Record

Cel: zamykanie operacji bazodanowych w metodach obiektów takich jak „save", „delete" i w metodach klasowych np. „serachById:", zwracających listy obiektów danej klasy.

Według Microsoftu jest to model dostępu do danych odpowiedni dla aplikacji mobilnych (w których zapytania są proste i operuje się na pojedynczych obiektach).

Wzorzec jest jednak krytykowany ze względu na łączenie logiki z danymi, czyli zaburzenie fundamentalnego podziału zdefiniowanego w MVC.

3.2.7. Data Transfer Object

Cel: obiekt nadzorujący przesyłanie danych pomiędzy procesami, redukujący liczbę wywołań funkcji.

W iDood rolę DTO pełni obiekt `IDatabaseManager`.

3.2.8. Transaction Script

Cel: organizowanie logiki biznesowej dla każdej transakcji w pojedynczej procedurze, która wywołuje bezpośrednio bazę danych lub kontaktuje się z nią poprzez wąską warstwę wrappera.

W aplikacji iDood wzorzec ten wykorzystuję przy takich operacjach jak rejestracja użytkownika, czy pobieranie wyników wyszukiwania.

3.2.9. Synchronization

Cel: komponent, monitorujący zmiany danych i wymiany informacji z serwerem, gdy połączenie jest aktywne.

3.2.10. Application Controller

Cel: obiekt zarządzający logiką przepływu danych, używany przez inne Kontrolery pracujące z Modelem w celu synchronizacji.

Można powiedzieć, że realizowanie tego wzorca jest narzucone przez Apple - każda aplikacja musi implementować IApplicationDelegate, który jest jej kontrolerem.

3.2.11. Model-View-Controller

Cel: ozdzielenie programu na 3 części: Model (dane), Widok (interfejs) i Kontroler (logika).

Istnieją dwa warianty tego wzorca: Pasywny Widok i Nadzorujący Kontroler, różniące się tym jak Widok współpracuje z Modelem. Tutaj podział również jest narzucony przez Apple - programista powinien jednak uważać, by nie wiązać zbyt mocno Modelu z Kontrolerem.

3.2.12. Model-View-Presenter

Cel: odmiana MVC w której zdarzenia interfejsu odbiera widok, a pozostałe funkcjonalności Kontrolera przejmuje Prezenter.

3.2.13. Pagination

Cel: dzielenie dużych ilości danych na mniejsze części w celu optymalizowania zasobów i używania, jak najmniej pamięci.

Aplikacja iDood pobiera jedynie po 10 wyników wyszukiwania (nawet jeśli jest ich dużo więcej), gdyż z reguły najistotniejsze wyniki są na początku. Użytkownik w każdym momencie może pobrać dalsze strony wyników.

3.2.14. Singleton

Cel: implementowanie matematycznej idei singletonu, poprzez ograniczenie tworzonych obiektów klasy do jednego egzemplarza.

Wykorzystywany jest między innymi do implementowania Fabryki, Fasady, czy Prototypu (więcej o wzorcach w [DPO]).

3.3. Implementacja wzorców projektowych w Objective-C

Wiele ze wzorców projektowych opisanych w rozdziale 3.2 bardzo łatwo realizuje się w języku Objective-C.

W szczególności delegacja jest łatwo implementowalna za pomocą przekazywania komunikatów. To z kolei ułatwia wprowadzanie następujących wzorców:

- Reliable Sessions - możemy wykorzystać delegację komunikatów,
- Data Transfer Object - realizujemy jako standardowy delegat,
- Application Controller - również jako delegat, singleton,

- Model-View-Controller, Model-View-Presenter - Widok, który nie umie przetworzyć komunikatu, przekazuje go Kontrolerowi lub Prezenterowi.

W powyższych wzorcach koncepcja komunikatów wydaje się naturalna i w związku z tym kod jest bardziej intuicyjny.

Przykładem DTO jest zaprezentowany w rozdziale 2.2.6 `DataProvider`, a do realizacji wzorca Singleton możemy skorzystać z kontrukcji zaprezentowanej w rozdziale 2.2.3.

Oprócz tego w aplikacjach iPhoneOS wykorzystałem m.in. wzorzec Target-action, w którym dla danej kontrolki wybieramy odbiorcę zdarzenia (target) i wysyłamy mu komunikat zdarzenia (action).

Rozdział 4

Wprowadzenie do systemu iPhoneOS

Oprócz języka, wiele elementów technicznych zostało narzuconych przez Apple - od sprzętu, na którym tworzymy aplikację, aż po kanał dystrybucji. Jedynym edytorem, w którym możemy stworzyć aplikację jest Xcode (działający tylko w systemie MacOS). Ten z kolei narzuca tworzenie aplikacji zgodnie z ideą MVC, gdzie do tworzenia widoku dostajemy specjalne narzędzie o nazwie Interface Builder.

Kod można napisać w innym programie, ale jest to co najmniej uciążliwe - żadne inne IDE nie jest zintegrowane z systemem iPhoneOS. Natomiast po zainstalowaniu iPhone SDK, Xcode oprócz edytora kodu daje emulator telefonu i umożliwia łatwe instalowanie aplikacji na urządzeniu.

4.1. Podstawowe wzorce

Podstawowymi wzorcami projektowymi w iPhoneOS wymienionymi przez Apple są:

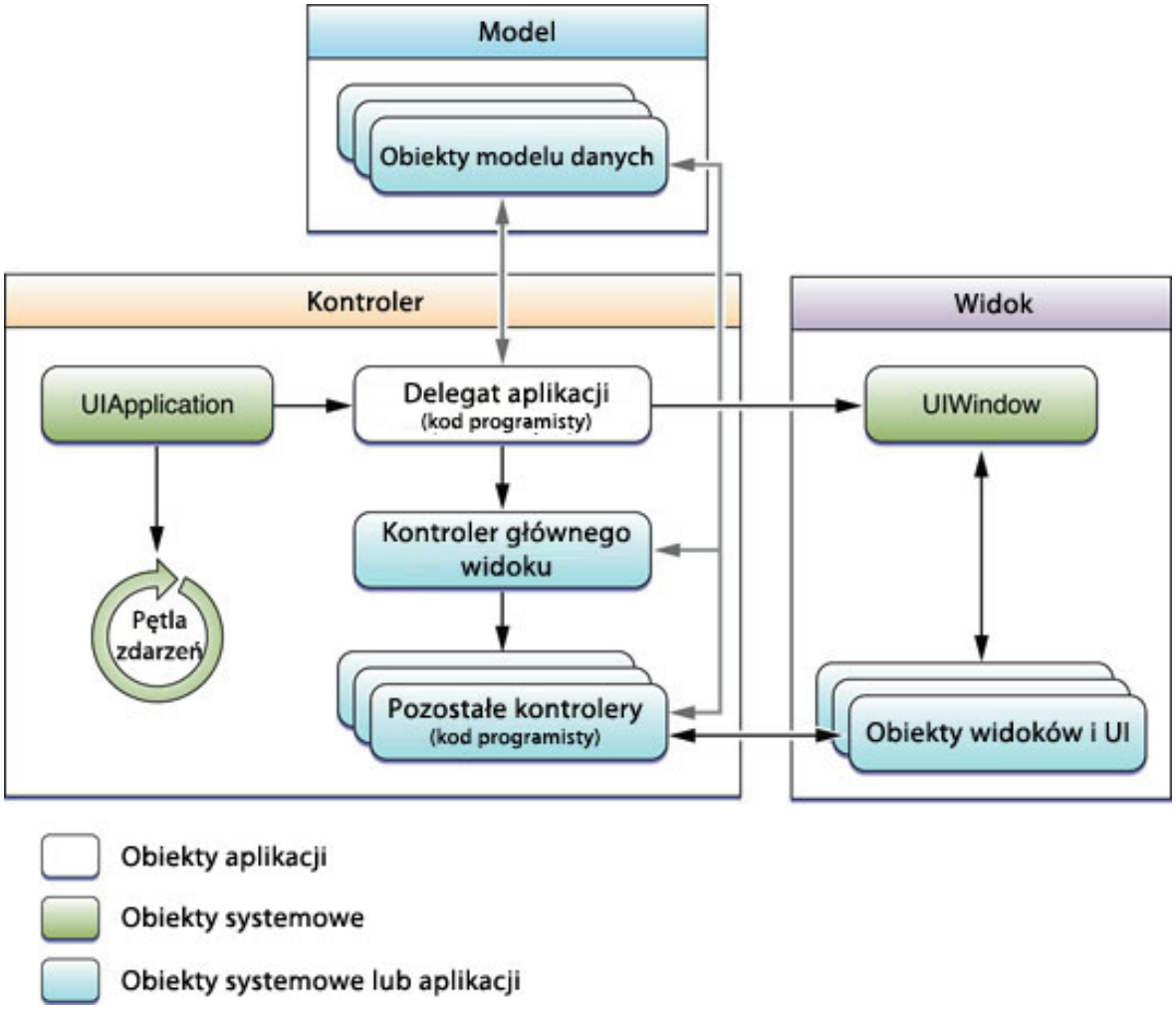
- Model-View-Controller,
- Delegation,
- Target-action,
- Managed memory model.

Można próbować tworzyć aplikacje na swój sposób, ale wygodniej jest korzystać z narzędzi tak, jak jest to sugerowane, dlatego starałem się budować aplikację zgodnie z sugestiami projektantów systemu.

4.2. Komponenty

Od samego startu aplikacja otrzymuje od systemu komunikaty. Odbiór komunikatów jest zadaniem opisanego dalej obiektu klasy `UIApplication`, a reakcja na nie to zadanie kodu programisty.

Na rysunku 4.2 przedstawiono podstawowe elementy aplikacji. Strzałki oznaczają przepływ komunikatów pomiędzy obiektami. Poszczególne obiekty opisałem dalej.



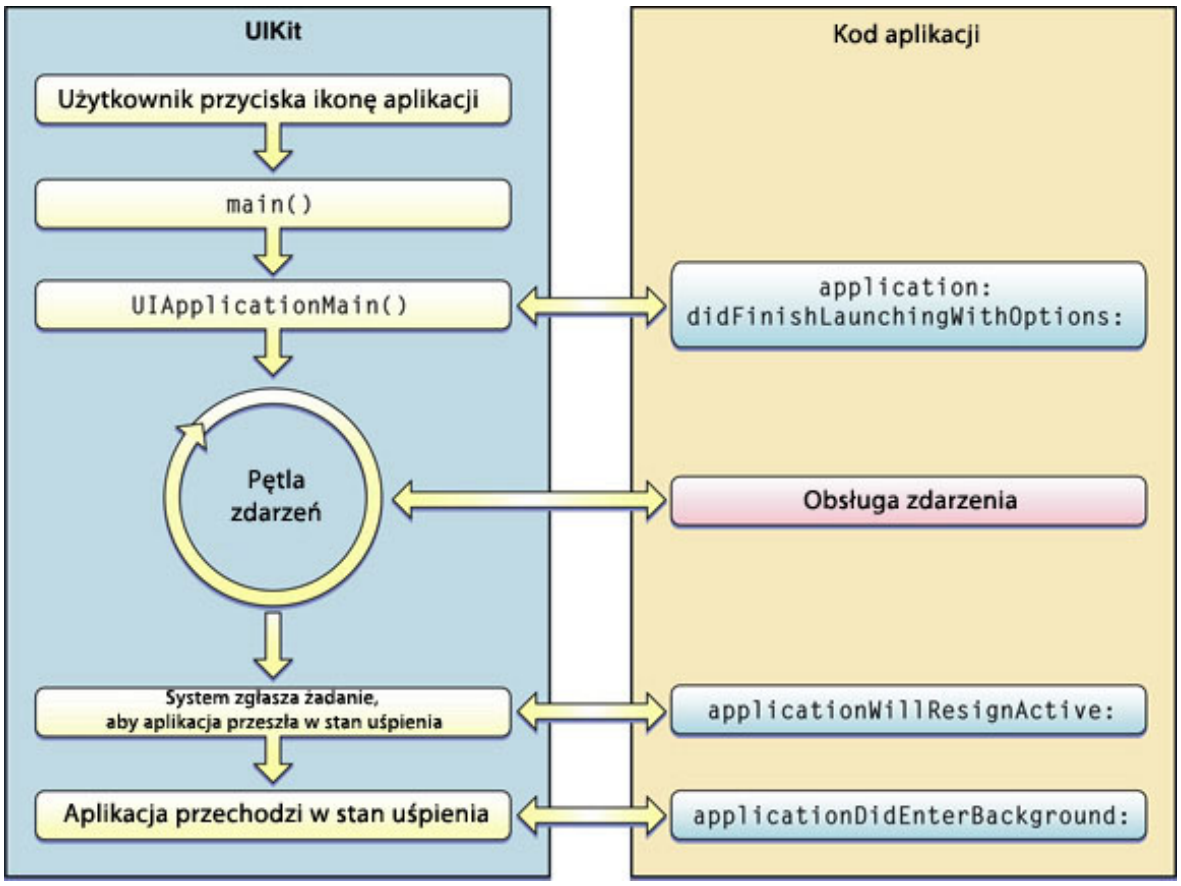
Rysunek 4.1: Kluczowe obiekty w aplikacji iPhoneOS [za Apple.com]

UIApplication	Zarządza główną pętlą aplikacji i koordynuje wysokopoziomowe zdarzenia. Programista nie ma kontroli nad tym obiektem i nie może go modyfikować - własny kod aplikacji umieszcza w IApplicationDelegate, który odbiera zdarzenia od UIApplication.
Delegat aplikacji	IApplicationDelegate to obiekt progamisty - najczęściej zdefiniowany w pliku konfiguracyjnym nib. Głównym zadaniem jest inicjalizacja aplikacji i pokazanie odpowiedniego okna. UIApplication przesyła również zdarzenia, kiedy aplikacja musi być wstrzymana (np. odebranie wiadomości sms) czy zawieszona (gdy użytkownik z niej wychodzi).
Model danych	Obiekty przechowujące dane - całkowicie zależne od programisty. Przechowujemy tutaj dowolne dane oprócz ustawień aplikacji.
Kontroler widoku	Kontroler zarządza wyświetlanym widokiem oraz odpowiada za wymianę danych pomiędzy widokiem a modelem. Klasa UIViewController jest bazowa dla wszystkich kontrolerów widoków aplikacji. Dostarcza standardowe funkcjonalności takie jak animacja i obrót (przy obróceniu urządzenia). Bardziej szczegółowe kontrolery odpowiadają za bardziej szczegółowe funkcje - np. kontroler widoku tabelki udostępnia komunikaty związane z przekazywaniem danych do poszczególnych komórek.
UIWindow	Obiekt UIWindow odpowiada za odpowiednie renderowanie aplikacji. Większość aplikacji iPhoneOS ma tylko jedno okno - aplikacja zmienia wygląd poprzez prezentowanie kolejnych widoków. Okno zarządza również przekazywaniem komunikatów do odpowiednich widoków i ich kontrolerów.
Widoki i kontrolki	Widoki i kontrolki umożliwiają reprezentację wizualną danych aplikacji. Widok jest obiektem, który rysuje treść w odpowiednim miejscu i przejmuje zdarzenia z tego obszaru. Kontrolki to widoki specjalnego rodzaju - odpowiednie do specyficznych funkcjonalności. Np. kontrolka suwaka prezentuje liczbę z danego zakresu i umożliwia jej zmianę. UIKit dostarcza wielu widoków i kontrolek. Można także tworzyć swoje przedefiniowuje UIView lub jej podklasy.

4.3. Cykl życia aplikacji

Na rys. 4.3 przedstawiono uproszczony cykl życia aplikacji. Diagram przedstawia przepływ zdarzeń od uruchomienia programu z menu, aż do wyjścia z niego. Po lewej stronie pokazane są elementy, za które odpowiada system operacyjny. Jak widać na diagramie, kod użytkownika ogranicza się do odbioru odpowiednich zdarzeń z systemu operacyjnego.

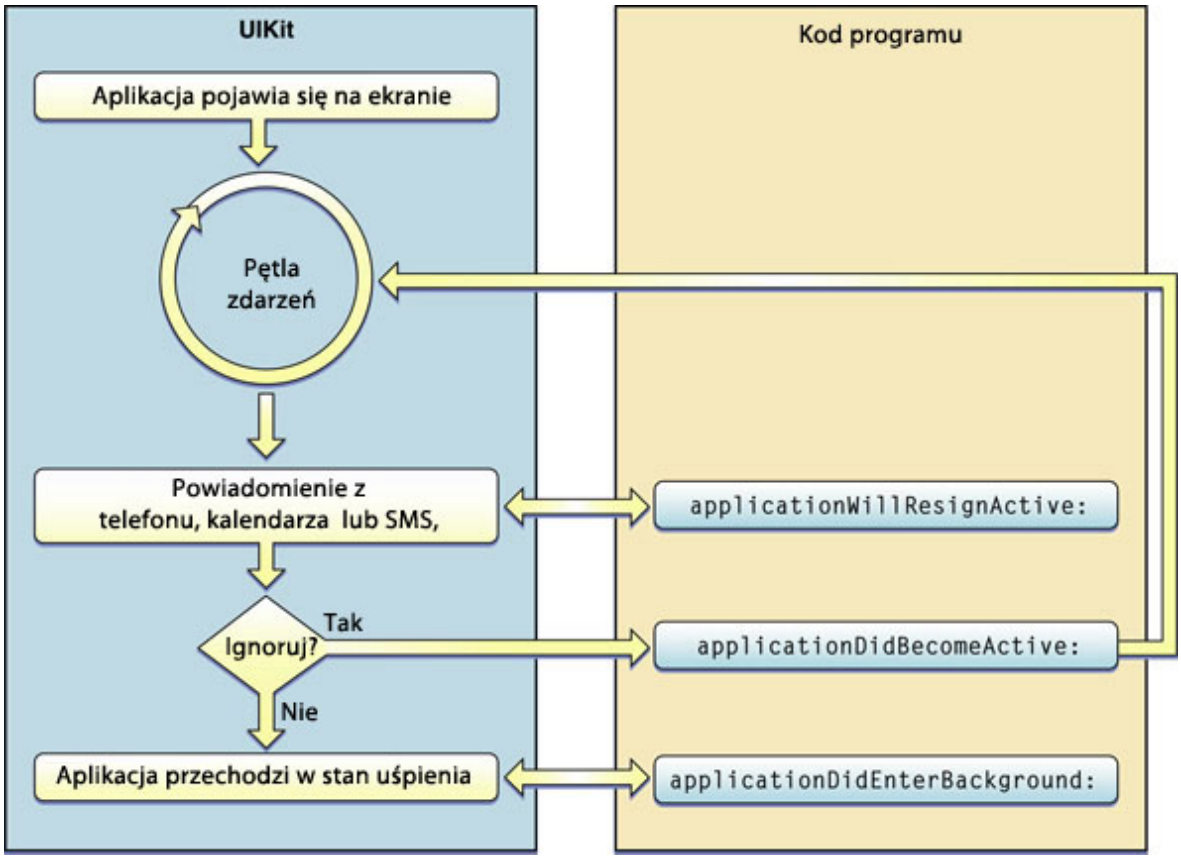
Warto tutaj zwrócić uwagę na zaletę wynikającą bezpośrednio ze struktury języka Objective-C i wzorca delegacji. Mianowicie zamiast w standardowy, znany z C++ i Javy sposób przedefiniowywania klas systemowych, zastosowanie delegacji daje nam możliwość odseparowania kodu od struktur systemu operacyjnego, skupiając się jedynie na odpowiednim przetwarzaniu komunikatów.



Rysunek 4.2: Cykl życia aplikacji [za Apple.com]

Obiekt, który odbiera komunikaty może być dowolnego typu, ale musi umieć zareagować na podstawowe zdarzenia systemowe, tj. musi implementować protokół UIApplicationDelegate.

W takim modelu stosunkowo łatwo zrealizować obsługę przerwań, co ilustruje rys 4.3.



Rysunek 4.3: Obsługa przerwań [za Apple.com]

Przepływ zdarzeń od użytkonika (np. dotknięcia ekranu) przebiega w następujący sposób:

1. System operacyjny odbiera zdarzenie od sprzętu.
2. System operacyjny dodaje zdarzenie jako obiekt IEvent do kolejki zdarzeń związanych z uruchomioną aplikacją.
3. Aplikacja w głównej pętli (w części kodu niedostępnej dla programisty) pobiera zdarzenie z kolejki i przekazuje je do UIWindow.
4. UIWindow przekazuje komunikat do „pierwszego respondera" stworzonego przez programistę.
5. Responder, który nie obsługuje zdarzenia, przekazuje je do "następnego respondera", którym zazwyczaj jest nadrzędny widok.
6. Responder, który umie obsłużyć zdarzenie przekazuje komunikat do swojego kontrolera i tam uruchamiany jest odpowiedni fragment kodu.

Powyższy proces dobrze obrazuje przepływ komunikatów, co jest kolejnym argumentem „za" językiem Objective-C.

Wysyłanie komunikatów pomaga również we współpracy pomiędzy aplikacjami. Przykładem może być wykonanie telefonu pod dany numer, które realizujemy poprzez wywołanie:

```
[[UIApplication sharedApplication] openURL:[NSURL URLWithString:@"tel:8006927753"]];
```

`SharedApplication` przekazuje nam specjalny obiekt aplikacji, w którym możemy otwierać URLe. Takie aplikacje systemu iPhone jak przeglądarka www, telefon, usługa wiadomości SMS i MMS, klient poczty czy odtwarzacz muzyki mają zadeklarowane swoje schematy URL i po uruchomieniu komendy analogicznej do powyższej, aplikacje otrzymują odpowiedni komunikat.

Możemy też zadeklarować własny schemat URLi. W aplikacji iDood zadeklarowałem schemat:

```
dood:zapytanie
```

gdzie `zapytanie` to fraza, którą chcemy wyszukać. Tzn. jeśli inna aplikacja chce umożliwić użytkownikowi znalezienie pizzerii w pobliużu to wystarczy, że wyśle następujący komunikat

```
[[UIApplication sharedApplication] openURL:[NSURL URLWithString:@"dood:pizza"]];
```

Rozdział 5

Implementacja i wydanie aplikacji iDood

Starałem się zrealizować prosty bazodanowy program wykorzystujący jak najszerszy wachlarz możliwości sprzętowych, a jednocześnie związany z portalem dood.pl.

Zaimplementowałem aplikację umożliwiającą wyszukiwanie firm detalicznych „w okolicy" miejsca, w którym znajduje się użytkownik. Program umożliwia również recenzowanie miejsc i przeglądanie recenzji znajomych z portalu.

5.1. Przypadki użycia

Głównym celem aplikacji jest umożliwienie znalezienia odpowiedniego lokalu. Mamy dwa przypadki związane z wyszukiwaniem:

1. Użytkownik wie konkretnie jaki lokal go interesuje - szuka jedynie danych teleadresowych lub recenzji.

2. Użytkownik wie tylko jaka branża go interesuje (np. sklep komputerowy) i na podstawie jego lokalizacji chcemy pomóc mu w wyborze.

Kolejnymi przypadkami są cele rozrywkowe - podczas wykorzystywania aplikacji w wolnej chwili lub w celu przeglądania bazy lokali nie oczekując konkretnych rezultatów:

3. Użytkownika interesują aktywności znajomych.

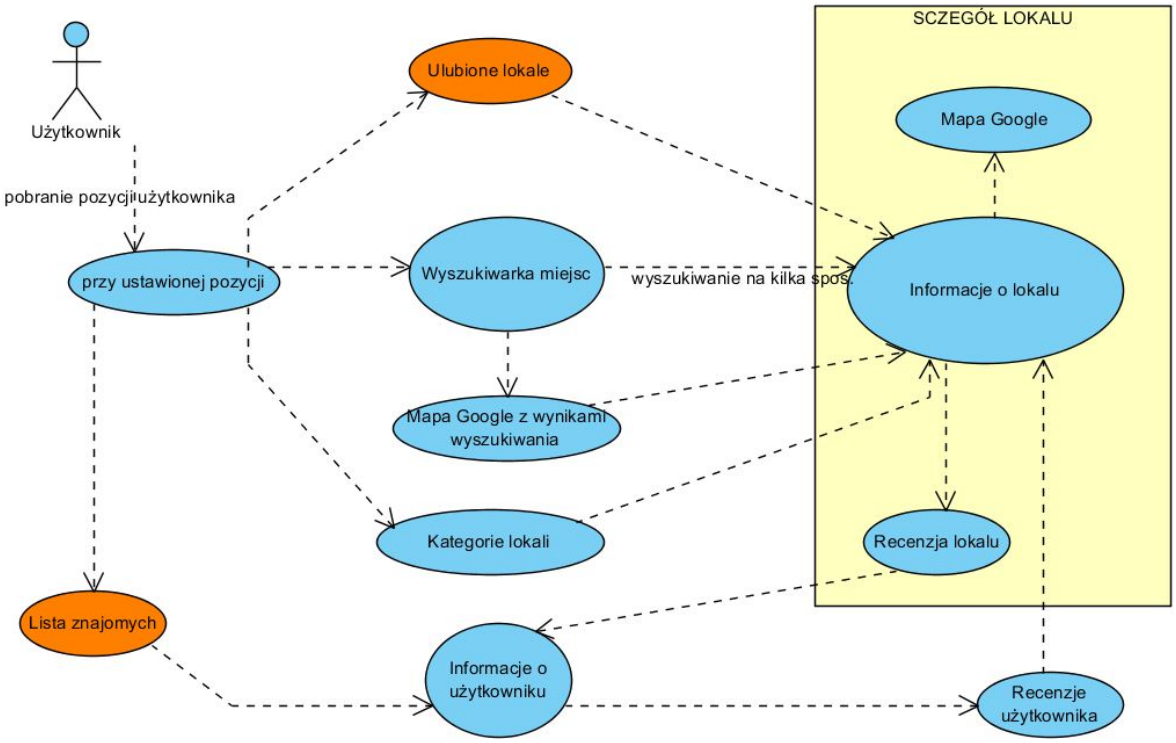
4. Użytkownik chce poznać więcej informacji o osobie recenzującej dany lokal.

Jak realizuję powyższe przypadki użycia?

Podzieliłem aplikację na dwie części: biznesową (przypadki 1 i 2) oraz społecznościową (przypadki 3 i 4).

Przypadek 1 realizujemy za pomocą wyszukiwarki, przypadek 2 realizujemy za pomocą wyszukiwarki lub katalogu kategorii.

Zarówno wyszukiwarka jak i katalog przekierowują do wyników wyszukiwania, z których użytkownik może wybrać lokal, którego szczegóły chce obejrzeć. Na widoku informacji o firmie zawarte są następujące informacje: recenzje, dane teleadresowe, zdjęcia, pozycja na mapie i odległość od aktualnego położenia. Dane, które otrzymuje, są tożsame z danymi na stronie dood.pl.



Rysunek 5.1: Stany aplikacji. Strzałki oznaczają przejścia pomiędzy widokami (bez uwzględnienia możliwości powrotu z każdego widoku do poprzedniego). Wyróżnione widoki (Lista znajomych i Ulubione lokale) wymagają zalogowania.

Natomiast, w części społecznościowej użytkownik może przeglądać profile znajomych - w szczególności również ich recenzje (przypadek 3). Do tej części zaliczyłem również przeglądanie recenzji i aktywności osób, na które użytkownik trafia przeglądając treści biznesowe (np. gdy chce dowiedzieć się więcej o recenzencie któregoś z lokali - przypadek 4).

Warto zwrócić tutaj uwagę na problem związany z nawigacją. Standardowym sposobem umożliwiania użytkownikowi intuicyjnego poruszania się po aplikacji mobilnej jest budowanie stosu widoków. W przypadku tej aplikacji należy jednak uważać na pętle

```
lokal -> recenzje -> użytkownik -> recenzje -> inny lokal
```

które mogą znacznie utrudnić choćby powrót do strony tytułowej.

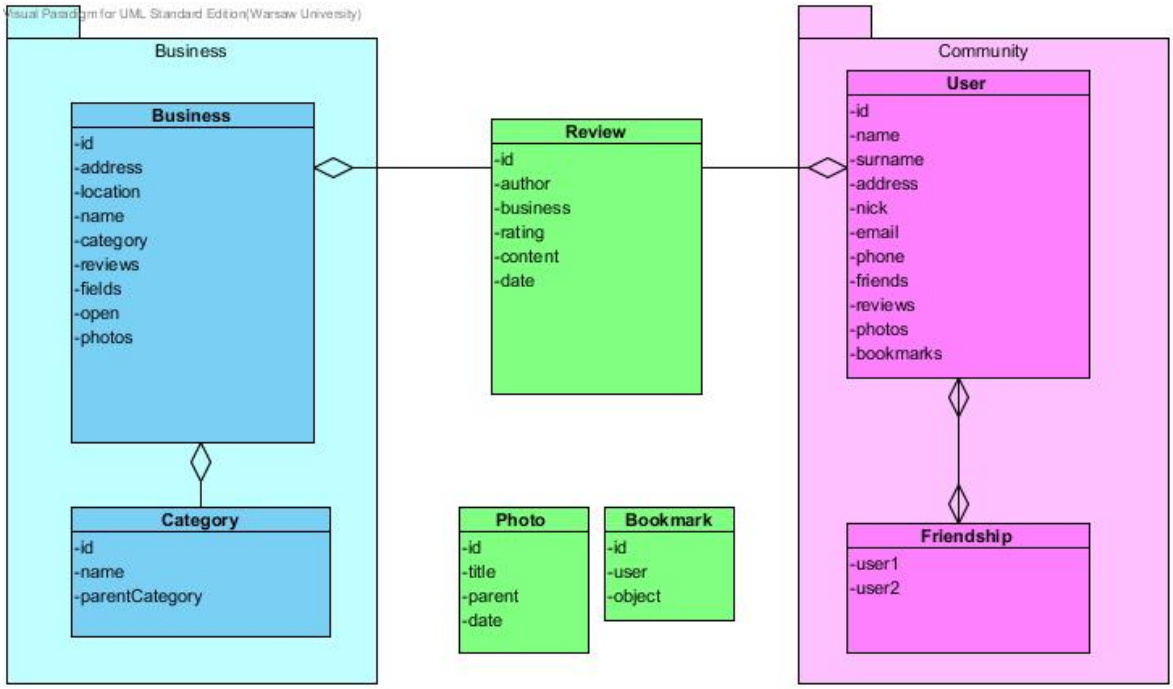
Ważnym zagadnieniem determinującym architekturę oprogramowania (poruszanym przez twórców iPhoneOS [PGU]) jest to w jaki sposób użytkownicy podchodzą do aplikacji mobilnych. Programy na telefon komórkowy nie są duże i najczęściej użytkownik oczekuje, że szybko się załadują, a czas pracy z aplikacją będzie krótki. Dlatego projektując aplikację starałem się by dojście do każdej strony docelowej nie wymagało przejścia przez więcej niż 2 widoki.

5.2. Model

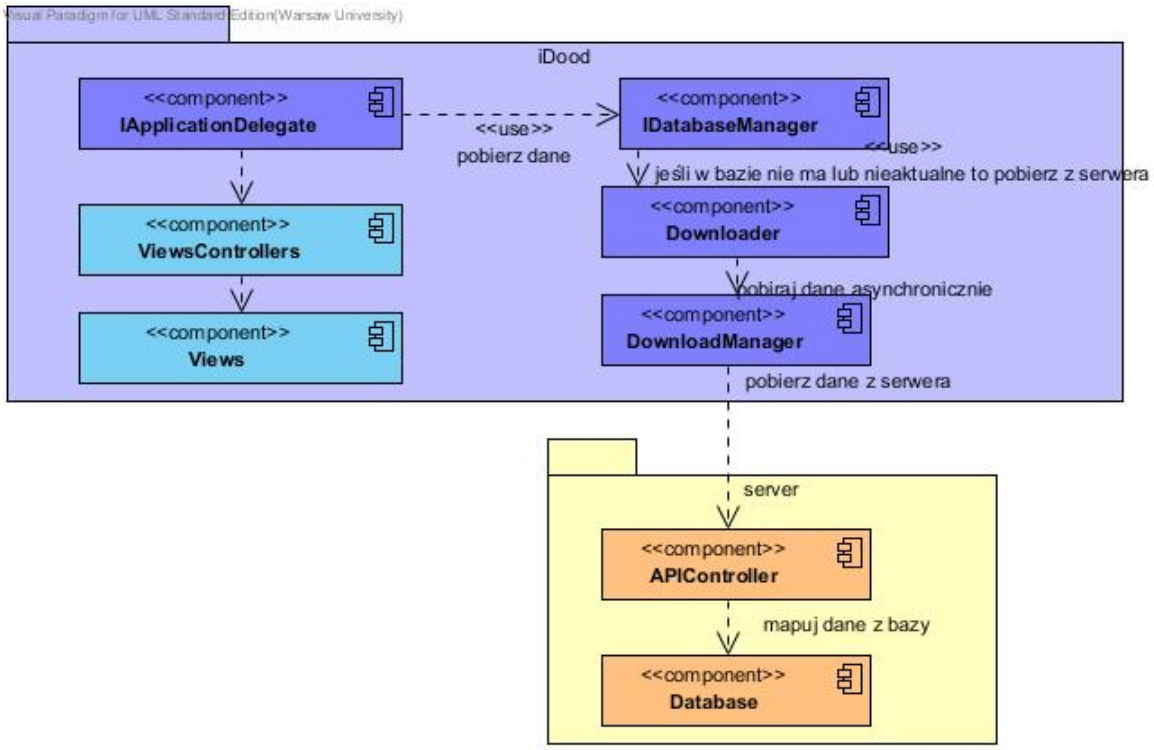
Model również podzieliłem na część biznesową i społecznościową, co obrazuje rysunek 5.2

5.3. Warstwy

Poniższa ilustracja przedstawia podział aplikacji na komponenty.



Rysunek 5.2: Model aplikacji



Rysunek 5.3: Zależności pomiędzy komponentami (MVC w którym model synchronicznie komunikuje się z bazą).

Aplikacja chcąc otrzymać dostęp do danych pobiera z ApplicationDelegate odpowiedniego managera i pyta go o informacje. Jeśli ten je posiada (niekoniecznie aktualne) to przekazuje, a sam sprawdza, czy nie pojawiły się na serwerze nowe. Jeśli tak, to pobiera je i informuje odpowiedni kontroler, który o te dane pytał. Jeśli kontroler już tych danych nie potrzebuje (bo np. został zamknięty), to komunikat jest po prostu ignorowany.

Dzięki temu użytkownik otrzymuje dane niekoniecznie aktualne tak szybko, jak tylko się da, a aktualizację dostaje kilka sekund później. W rezultacie nie musi czekać przy pustym oknie.

5.4. API

Do komunikacji z serwerem zaprojektowałem (wraz z Pawłem Bedyńskim, studentem IV roku MIMUW) specjalne API w konwencji REST. Wykonywane są zapytania HTTP, a wynik przekazywany jest w formacie JSON. Nazwy komunikatów staraliśmy się budować w konwencji

```
zasób/akcja?parametry
```

tzn. w celu uzyskania np. listy recenzji użytkownika nr 123, wywołujemy zapytanie

```
review/get?user=123
```

w celu otrzymania informacji o firmie nr 2352 wywołujemy

```
business/get?id=2352
```

a chcąc dodać recenzję wywołujemy

```
review/put?body=Bardzo\%20fajny\%20lokal&business_id=2352&rating=5
```

Zdecydowaliśmy się na format JSON, ze względu na jego prostotę, która w zupełności wystarcza do przesyłania danych nie wymagających uwierzytelniania i szyfrowania.

Pełen opis API znajduje się aktualnie pod adresem *http* : *//api.dood.pl* : 81/

5.5. Implementacja

Z każdym widokiem z rysunku 5.1 związana jest odpowiednia klasa. Są to: `CategoriesView`, `ResultsView`, `ReviewsView`, `BusinessDetailsView`, `ContactsView`, `BookmarksView`. Widoki odpowiadają za prezentację treści, są też pierwszymi odbiorcami komunikatów. Każdy komunikat interfejsu użytkownika przekazywany jest do kontrolera widoku (odpowiednio `CategoriesViewController`, `ResultsViewController`, `ReviewsViewController`, `BusinessDetailsViewController`, `ContactsViewController`, `BookmarksViewController`).

Zarządzanie pamięcią odbywa się poprzez obiekt `NSManagedObjectContext`, który tworzymy w `iAppDelegate`. Obiekt ten pozwala na tworzenie obiektów zaprojektowanego wcześniej modelu (5.2). Mechanizm dostarczony przez Apple automatycznie tworzy bazę danych sqlite - generuje ją na podstawie diagramu.

W rezultacie zadaniem programisty jest jedynie pisanie kodu kontrolerów - widoki bowiem przygotowujemy z pomocą narzędzia WYSIWYG o nazwie InterfaceBuilder, a model przygotowujemy graficznie w narzędziu o nazwie Core Data.

Do zadań kontrolerów należą:

- przygotowywanie danych do zaprezentowania,

- reagowanie na akcje użytkownika,
- zapis danych.

Dla widoków w postaci tabeli (`TableView`) przygotowywanie danych przebiega tak, że widok odpytuje kontroler o odpowiednie wartości. Np. dla widoku `ResultsView` z wynikami wyszukiwania:

1. kontroler otrzymuje komunikat `numberOfSectionsInTableView`,
2. w odpowiedzi deklaruje liczbę sekcji - w tej sytuacji będzie to 1,
3. kontroler otrzymuje komunikat `numberOfRowsInSection`,
4. w odpowiedzi deklaruje liczbę komórek - u nas jest to liczba wyników wyszukiwania,
5. kontroler otrzymuje komunikat `cellForRowAtIndexPath`,
6. tworzymy komórkę wyniku wyszukiwania (`BusinessCell`), uzupełniamy dane i zwracamy ją.

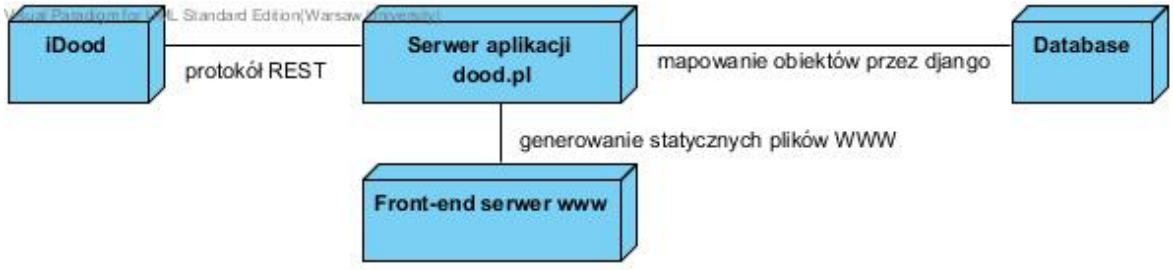
W przypadku innych widoków często korzystamy z tzw. outletów. Tzn. w InterfaceBuilderze łączymy odpowiedni widget na widoku z odpowiednią zmienną w kontrolerze. Np. Pole `Label` nazwy lokalu w widoku z odpowiednią zmienną typu `NSString name` w kontrolerze. Wtedy po odpowiednim zainicjowaniu zmiennych widok sam wie co ma wyświetlić i nie musimy przedefiniowywać komunikatów. Przykładem są `BusinessDetailsView` i `BusinessDetailsViewController`.

Do tworzenia ustawień programu znów korzystamy z narzędzia dostarczonego przez Apple. W specyfikacji projektu definiujemy zmienne i podajemy ich typy (np. napis `login`, napis `hasło`, liczba "ile wyników pokazywać na stronie" itp.), a system operacyjny automatycznie tworzy zakładkę w programie zarządzającym wszystkimi ustawieniami.

Na rysynku 5.5 przedstawiłem strukturę warstw i związane z nimi urządzenia. Aktualnie system składa się z trzech serwerów:

- database - baza danych MySQL z danymi serwisu,
- back-end - serwer aplikacji napisany w języku python, generujący podstrony serwisu oraz odpowiadający na zapytania API,
- front-end - serwujący wygenerowane podstrony.

Aplikacja komórkowa łączy się bezpośrednio z serwerem aplikacji.

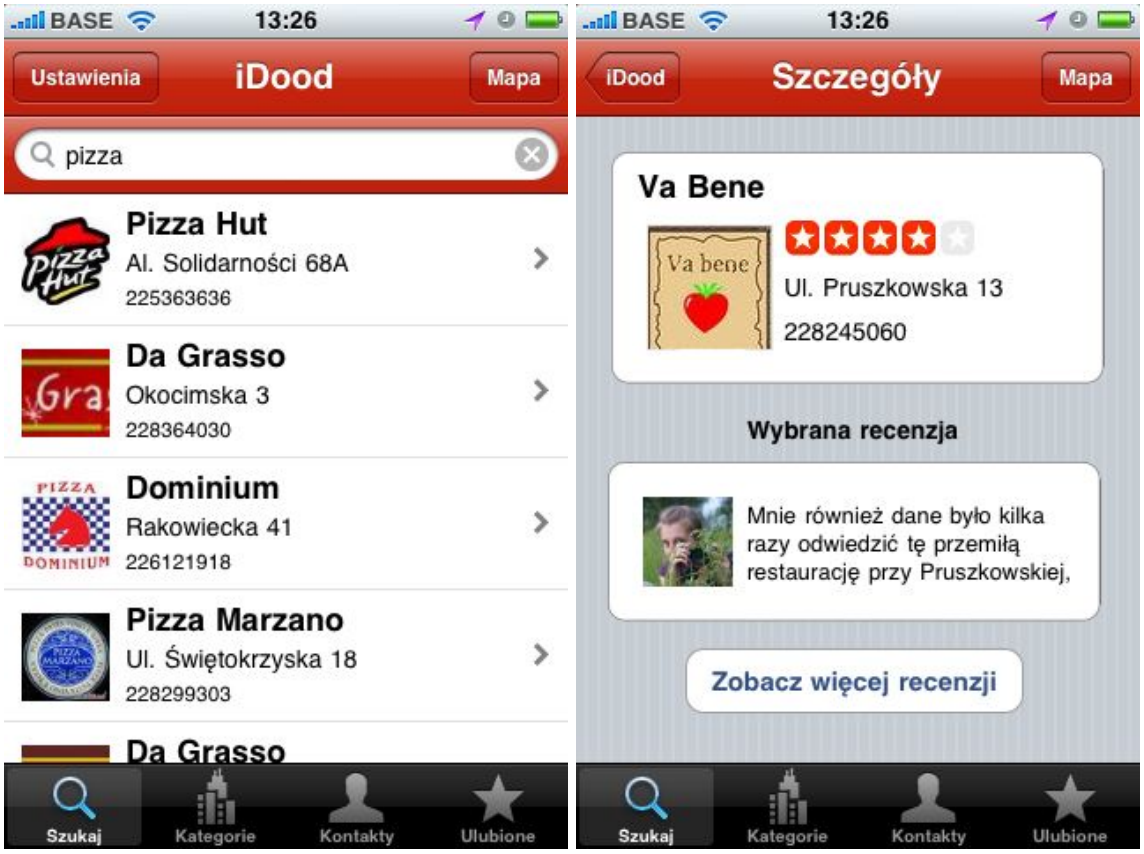


Rysunek 5.4: Komponenty portalu dood.pl

5.6. Wydanie aplikacji

Projektując aplikację starałem się jak najbardziej zbliżyć do istniejącej już na rynku amerykańskim aplikacji iPhone serwisu yelp.com - oferującego podobną funkcjonalność do serwisu dood.pl.

Ostateczny wygląd aplikacji iDood przedstawiają zrzuty ekranu na rys. 5.6.



Rysunek 5.5: Interfejs użytkownika aplikacji iDood

Jedynym legalnym kanałem dystrybucji aplikacji iPhone jest sklep App Store. Przed dodaniem programu do sklepu musi on zostać sprawdzony przez Apple. Weryfikacja dotyczy m.in.:

- poprawności działania,
- zgodności z konwencjami interfejsu przyjętymi przez Apple,
- poprawności używania znaków towarowych Apple oraz innych kwestii prawnych,
- poprawności treści - brak elementów pornograficznych, obraźliwych kontekstów religijnych, kulturowych itp.

Część warunków została spisana w dokumencie App Store Review Guidelines [ARG] opublikowanym 9 listopada 2010, choć firma Apple zastrzega sobie możliwość odrzucenia aplikacji z innych powodów:

„Odrzucamy aplikacje których treść lub działanie wykracza poza granicę. Jaką granicę, zapytacie? Cóż, tak powiedział kiedyś Sąd Najwyższy USA - "Będę wiedział, jak zobaczę.". Wydaje nam się, że Ty też będziesz wiedział, gdy ją przekroczysz."

Dnia 14 grudnia 2010 aplikacja iDood została zatwierdzona przez Apple i umieszczona w sklepe AppStore na zasadach licencji freeware.

Rozdział 6

Podsumowanie i wnioski

Mimo, że wszystkie języki programowania mają tę samą siłę wyrazu, składnia może znacząco przełożyć się na efektywność tworzenia kodu. W niniejszym rozdziale postaram się podsumować najważniejsze elementy wyróżniające Objective-C na tle innych języków, w kontekście tworzenia aplikacji mobilnych.

6.1. Objective-C

Zagadnienia, których implementacja jest bardzo naturalna w języku Objective-C, dzięki jego specyfice, to między innymi:

- Ustawianie wartości danego obiektu modelu na podstawie słownikowych danych otrzymanych z serwera, dzięki wykorzystaniu kategorii i refleksji (rozdział 3.1).
- Zmniejszenie liczby zależności pomiędzy klasami, dzięki kategoriom i przekazywaniu komunikatów (dobrą ilustracją jest rozwiązanie zaprezentowane w rozdziale 4.3).
- Możliwość korzystania ze źródeł bibliotek C bez ich modyfikacji - do iPhone SDK dołączone są np. biblioteki C takie jak OpenGL czy sqlite.
- Wzorzec obeserwatora możemy zrealizować wyjątkowo łatwo, tak by dało się obserwować dowolną zmienną klasy bez ingerowania w jej kod, dzięki wykorzystaniu kategorii i refleksji. Implementacja (nazwana Key-Value Observing) jest częścią iPhone SDK API. Więcej na ten temat można przeczytać w [KVO].

Ciekawym rozwiązaniem jest również możliwość dodawania implementacji komunikatów do istniejących już klas. Pozwoliło to np. na dodanie nowego komunikatu `JSONValue` do klasy `NSString`, który tłumaczy napis w formacie JSON na zmienną słownikową. Inne języki programowania dają nam możliwość zrealizowania takiej funkcjonalności za pomocą dziedziczenia, ale podejście w stylu Objective-C ma swoich zwolenników (argumentem znów jest między innymi zmniejszanie liczby zależności).

6.2. Sukces iPhone

Główne różnice projektowe języków dla platform mobilnych, choć są zasadnicze, same w sobie nie są raczej powodem, dla którego programiści zaczęli masowo tworzyć aplikacje iPhone.

Można się dopatrywać wielu czynników stojących za sukcesem tej platformy, od informatycznych po socjologiczne i psychologiczne. Za jeden z nich uważa się duży nacisk firmy Apple na doznania użytkownika (User Experience).

Na podstawie analizy źródeł, wypowiedzi programistów, jak i własnych doświadczeń wydaje mi się, że czynnikiem nieco pomijanym, a również istotnym i wspieranym przez Apple mogą być doświadczenia programisty - Developer Experience. Na platformę iPhone powstało już ponad 300 000 (20 października 2010) - większość z nich to proste małe programy, z których radość czerpie zarówno użytkownik, jak i programista.

Bibliografia

[ARC] Microsoft Patterns & Practices Team *Microsoft Application Architecture Guide (Patterns & Practices)*, Microsoft Press, October 2009, Chapter 19: Mobile Applications.

[ARG] App Store Resource Center, *App Store Review Guidelines*, http://stadium.weblogsinc.com/engadget/files/app-store-guidelines.pdf, pobrano 21 listopada 2010

[CDI] Steven S. Muchnick, *Advanced compiler design and implementation*, Morgan Kaufmann Publishers, 1997 p. 336

[COC] Matt Gallagher, *Cocoa with Love*, http://cocoawithlove.com/2009/04/what-does-it-mean-when-you-assign-super.html, pobrano 13 października 2010.

[COG] Cogniance company, *Cogniance blog*, http://blog.cogniance.com/65/, pobrano 14 października 2010.

[DPO] Ahlgren, Riikka and Markkula, Jouni *Design Patterns and Organisational Memory in Mobile Application Development*, Springer Berlin / Heidelberg 2005, str. 143-156.

[DPR] Gamma, Erich, Richard Helm, Ralph Johnson i John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software.*, Addison Wesley Professional, 1995.

[GCC] Free Software Foundation, Inc., *GCC 4.0.4 documentation*, http://gcc.gnu.org/onlinedocs/gcc-4.0.4/gcc/C-Dialect-Options.html, pobrano 13 października 2010

[KVO] Apple, *Introduction to Key-Value Observing Programming Guide* http://developer.apple.com/library/mac/#documentation/Cocoa/Conceptual /KeyValueObserving/KeyValueObserving.html, pobrano 14 listopada 2010

[LTR] Laurence Tratt, *Dynamically Typed Languages*, http://tratt.net/laurie/research/publications/papers/tratt__dynamically_typed_languages.pdf Bournemouth University, 2009

[MEP] Meijer, Erik and Peter Drayton, *Static Typing Where Possible, Dynamic Typing When Needed: The End of the Cold War Between Programming Languages*, http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.69.5966&rep=rep1&type=pdf Microsoft Corporation, 2005

[MEX] Dalrymple, Mark; Knaster, Scott, The Nielsen Company, *Learn Objective-C on the Mac*, p. 9. „The .m extension originally stood for messages when Objective-C was first introduced, referring to a central feature of Objective-C"

[MSH] Don Kellogg, Senior Manager, Research and Insights/Telecom Practice, The Nielsen Company, *iPhone vs. Android*, http://blog.nielsen.com/nielsenwire/online_mobile/iphone-vs-android/, pobrano 10 sierpnia 2010

[NXT] Apple, *Apple Computer, Inc. Agrees to Acquire NeXT Software Inc.*, http://web.archive.org/web/19970301172356/http://live.apple.com/next/961220.pr.rel.next.html, Apple 1997, pobrano 10 sierpnia 2010

[OJD] James Bucanek *Learn Objective-C for Java Developers*, apress, 2009, Chapter: Introduction

[OPG] Apple, *The Objective-C Programming Guide*, http://developer.apple.com/mac/library/documentation/Cocoa/Conceptual/ObjectiveC/Introduction/introObjectiveC.html, Objects, Classes, and Messaging, pobrano 27 lipca 2010

[OPL] Apple, *The Objective-C Programming Language*, http://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/ObjectiveC/ObjC.pdf, str. 9, pobrano 13 października 2010

[PER] Mike Ash, *Performance Comparisons of Common Operations*, http://www.mikeash.com/pyblog/performance-comparisons-of-common-operations.html, pobrano 27 lipca 2010

[PGU] Apple, *iOS Application Programming Guide*, http://developer.apple.com/iphone/library/documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/Introduction/Introduction.html, pobrano 27 lipca 2010

[PHY] Wikipedia, *Philosophical differences between Objective-C and C++*, http://en.wikipedia.org/wiki/Objective-C#Philosophical_differences_between_Objective-C_and_C.2B.2B, Wikipedia, pobrano 15 sierpnia 2010

[WPR] Wikipedia, *Objective-C - protocols*, http://en.wikipedia.org/wiki/Objective-C#Protocols, Wikipedia, pobrano 15 sierpnia 2010