UNIWERSYTET W BIAŁYMSTOKU
WYDZIAŁ MATEMATYKI I INFORMATYKI
INSTYTUT INFORMATYKI

Grzegorz SZYMAŃSKI

Wyświetlanie zawartości webowej w mobilnym systemie operacyjnym iOS

Promotor:
dr inż. Dominik TOMASZUK

BIAŁYSTOK 2015

Karta dyplomowa

UNIWERSYTET W BIAŁYMSTOKU Wydział Matematyki i Informatyki Zakład / Katedra Zakład	Studia Stacjonarne I stopnia	Nr albumu studenta 00000 Rok akademicki 2011/2012 Kierunek studiów Informatyka Ścieżka kształcenia Informatyka teoretyczna

Grzegorz Szymański
. . . .
Imię i nazwisko studenta

TEMAT PRACY DYPLOMOWEJ:

Wyświetlanie zawartości webowej w mobilnym systemie operacyjnym iOS

. . . .
Ocena promotora

. . . .
Podpis promotora

. . . .
Imię i nazwisko recenzenta

. . . .
Ocena recenzenta

. . . .
Podpis recenzenta

Spis treści

Streszczenie

Wstęp

Rozdział 1

Serializacja i prezentacja danych

1.1 Standardy internetowe

Standardami internetowymi możemy nazwać zbiór norm, które opisują i definiują sieć *World Wide Web*. Określane są przez organizację *World Wide Web Consortium*, w skrócie *W3C*, powstałą w 1994 roku a jej założycielem jest Tim Berners-Lee. Zrzesza ona ponad 400 organizacji, firm, agencji rządowych i uczelni z całego świata.

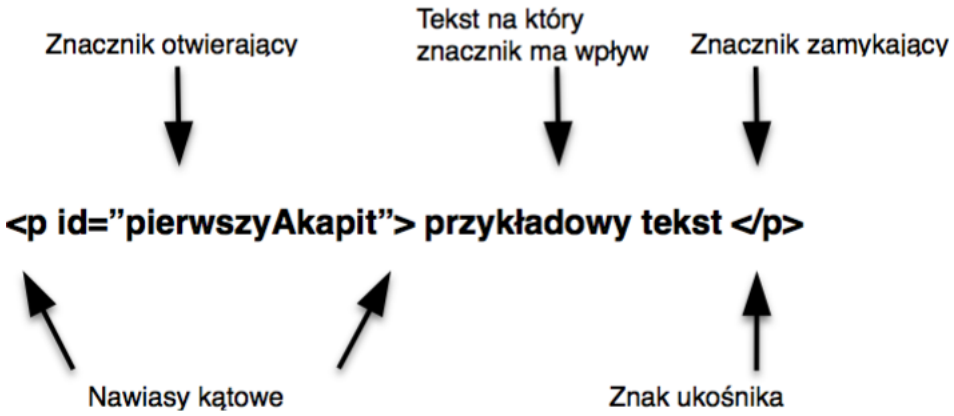
1.1.1 HTML

HTML [1] czyli Hipertekstowy język znaczników (ang. *HyperText Markup Language*) – początkowo został zaprojektowany jako język, którym opisywano dokumentację naukową. Z biegiem czasu dzięki swojej klarowności i adaptowalności reguł stał się podstawowym sposobem zapisania dokumentu w sieci Internet.

Język ten można scharakteryzować przez dwie główne cechy hipertekst oraz uniwersalność. Hipertekst można określić jako łączę w dokumencie, które jest w stanie przenieść osobę oglądającą stronę do dowolnego innego dokumentu lub zasobu Internetu. Prof. Tim Berners-Lee chciał, aby Sieć przypominała ludzki mózg, a nie statyczne źródło informacji, takie jak książka. Uniwersalność oznacza, że każdy komputer jest w stanie odczytać plik HTML ponieważ jest zapisany jako tekst w ASCII. Jest językiem znaczników zawierających instrukcje formatujące oraz szczegółowe informacje na temat wzajemnych relacji pomiędzy poszczególnymi fragmentami dokumentu. W *HTML* występują trzy zasadnicze typy języka znaczników – elementy, atrybuty oraz wartości.

Elementy są etykietami identyfikującymi i określającymi strukturę różnych części stron *WWW*. Niektóre elementy mają jeden lub większą ilość atrybutów, które bardziej szczegółowo opisują zawartość. Elementy mogą zawierać tekst oraz inne elementy; mogą także być puste. Elementy, które nie są puste, składają się ze znacznika otwierającego (który zawiera nazwę, atrybuty zapisane wewnątrz pary nawiasów kątowych), zawartości oraz znacznika zamykającego (znak ukośnika oraz nazwę elementu) [4] – rysunek 1.1 przedstawia strukturę elementu na przykładzie akapitu.

Atrybuty zawierają szczegółowe informacje o danych umieszczonych na stronie, natomiast same nie są tymi informacjami. Wartości atrybutów zapisywane są po-



Rysunek 1.1: Struktura znacznika

między znakami apostrofu lub cudzysłowu. Prawdopodobnie najpopularniejsze są atrybuty przybierające wartości wyliczeniowe lub predefiniowane, które należą do standardowego zbioru (Listing 1.1)

```
1 <link rel="stylesheet" type="text/css"
2   media="screen" href="style.css">
```

Listing 1.1: Wartość predefiniowana

Niektóre atrybuty zapisujemy jako wartości liczbowe bądź procentowe. Przede wszystkim atrybuty, które określają wartość lub długość. Jeśli wartościom atrybutów da się przyporządkować jakąś jednostkę, jak w przypadku wysokości tekstu lub szerokości obrazka, to zakłada się, że wartości te są wyrażone w pikselach. Wartościami atrybutów określających kolory mogą być nazwy kolorów lub wartość zapisana szesnastkowo składająca się z koloru czerwonego, zielonego oraz niebieskiego. Niektóre atrybuty odwołują się do innych plików i dlatego ich wartości muszą przybierać formę adresów URL (*Uniform Resource Locator*) – patrz listing 1.1 atrybut *href* w linii numer 8.

```
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <title>Sample page</title>
5   </head>
6   <body>
7     <h1>Sample page</h1>
8     <p>This is a <a href="demo.html">simple</a> sample.</p>
9     <!-- this is a comment -->
10  </body>
11 </html>
12
```

Listing 1.2: Przykładowa składnia dokumentu HTML

Każdy dokument *HTML* zaczyna się deklaracją w jakim języku napisana jest strona. Przykład przedstawiony na listingu 1.2 w linii numer 1 używa najnowszej 5

wersji specyfikacji języka *HTML*. Następna część dokumentu ma charakter drzewiasty, gdzie korzeniem drzewa jest znacznik *<html>* a jego potomkami *<head>* oraz *<body>*.

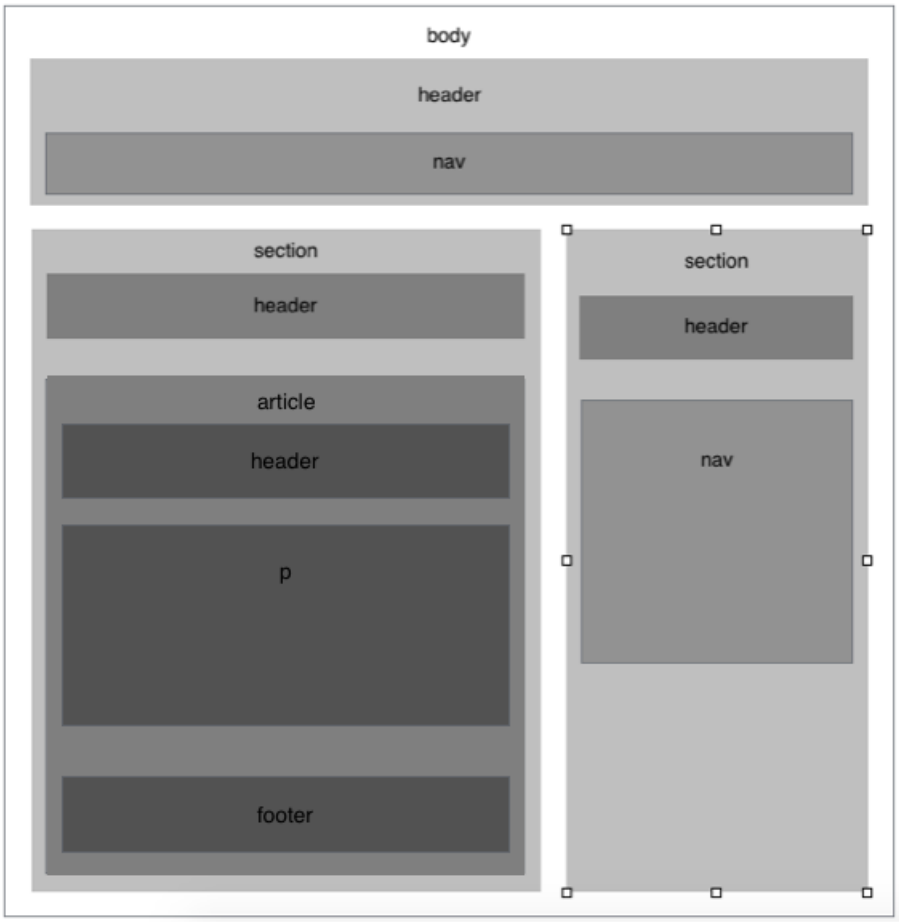
W sekcji *<head>* nagłówku dokumentu określamy tytuł strony, który umieszczany jest między znacznikami *<title></title>* i wyświetlany jest w pasku przeglądarki. Metadane, których zadaniem jest dostarczenie dodatkowych informacji na temat witryny i pozwalają ustalić:

- kodowanie znaków.
- wskazanie języka w jakim dokument został przygotowany.
- słowa kluczowe dla wyszukiwarki.

Znacznik *<body>* odpowiada za treść, która wyświetlana jest w przeglądarce (lub innym medium, np. drukarce). Wyróżniamy trzy najważniejsze komponenty treści: zawartości tekstowej, czyli faktycznych nagłówków i akapitów, które są widoczne dla osoby przeglądającej stronę; odwołań do bardziej złożonych elementów, czyli linków, obrazków, a nawet animacji; oraz języka znaczników. Najprostszym elementem blokowym do umieszczenia zawartości tekstowej jest znacznik akapitów* *– widoczny na rysunku 1.1. Dzięki tym znacznikom przeglądarka będzie w stanie odpowiednio sformatować akapit. Znacznik akapitu może zawierać inne elementy lecz nie inne akapity. Przydatnym elementem przy formatowaniu przejścia do nowego wiersza, bez kończenia akapitu może okazać się znacznik podziału wiersza: *
*. Aby określić unikalny identyfikator dla akapitu umieszczamy wewnątrz znacznika otwierającego atrybut *id*. W przypadku gdy chcemy aby znacznik należał do grupy elementów używamy atrybutu *class*. Do danej klasy może należeć dowolna ilość elementów.

Do grupowania powiązanych ze sobą obiektów służą sekcje. Pozwalają także, by grupowane wewnątrz nich obiekty korzystały ze wspólnych sposobów formatowania; wystarczy w tym celu określić sposób formatowania dla samej sekcji. Umieszczone wewnątrz sekcji obiekty dziedziczą określone cechy wyglądu, dzięki czemu nie trzeba określać formatowania osobno dla każdego z tych obiektów.

Do budowy struktury strony wraz z wejściem standardu *HTML5*, oprócz znacznika blokowego * *, do dyspozycji mamy elementy takie jak *<section>*, *<article>*, *<header>* czy *<footer>*. Rysunek 1.2 przedstawia strukturę strony stworzonej przy użyciu nowych znaczników. Element *<section>* reprezentuje ogólną sekcję dokumentu, bądź aplikacji. Używany jest do grupowania podobnie tematycznej treści. Za pomocą tego elementu możemy również tworzyć podsekcje, zagnieżdżając je w sobie. Przykładem takiej sekcji jest część przechowująca wpisy zamieszczone na blogu. Znacznik *<article>* pozwala oznaczyć sekcję dokumentu będącą oddzielną, niezależna jego częścią. Jest wprost stworzony do opisywania właściwej treści strony internetowej. W trakcie definiowania wielu różnorodnych elementów na stronie, takich jak nagłówki, stopki, czy elementy nawigacyjne, łatwo jest zapomnieć, że najważniejsza jest treść publikowana na stronie internetowej. Treść tę opisuję właśnie znacznik *<article>*. Nagłówek *header*, którego nie należy mylić z nagłówkami *h1, h2, h3* mogą zawierać dowolne treści od loga, a na wyszukiwarce kończąc. Na danej stronie może znajdować się jeden lub więcej nagłówków. Każda sekcja strony lub artykuł także może posiadać swój własny nagłówek, dlatego warto definiować dla każdego atrybut ID [6].



Rysunek 1.2: Struktura strony opisana przy użyciu nowych znaczników obecnych w HTML5

1.1.2 CSS

Kaskadowe arkusze stylów (ang. *Cascading Style Sheets*, w skrócie *CSS* [5]) to język używany do określenia formatowania elementów witryny internetowej, takich jak kolory, rysunki tła, czcionki, marginesy i akapity. By zwiększyć czytelność kodu *HTML* i ułatwić wprowadzenie zmian do prezentacji wielu stron używających ten sam arkusz rozdzielono strukturę dokumentu od jego wyglądu. Rozdzielenie takie to również lżejsza strona, co dla większych portali informacyjnych i użytkowników z wolniejszym łączem internetowym ma znaczenie.

CSS zamiast wersji posiada poziomy, każdy poziom budowany jest na podstawie poprzedniego poprzez udoskonalenia definicji i dodawania nowych funkcjonalności. Obecnie posiada trzy poziomy dwa z nich to obowiązujące specyfikacje, trzeci natomiast ma status rekomendacji. Konsorcjum *W3C* by przyśpieszyć prace nad wdrażaniem standardu *CSS3* w przeglądarkach, postanowiło podzielić go na moduły. Każdy z tych modułów odpowiada za oddzielne części, takie jak transformacja 2D, tła czy obramowania. Status oficjalnego standardu na tą chwilę otrzymało tylko dwa moduły: CSS3 Colors oraz CSS3 Selectors.

CSS by spełniał swoje zadanie, musi być przypisany do dokumentu strukturalnego. Pierwszym sposobem o najmniejszym zasięgu jest styl lokalny. Metoda ta pozwala na ustawienie wyglądu tylko dla jednego elementu, w którym zostanie

osadzony. Fragment kodu w listingu 1.3 odniesie skutek tylko do nagłówka *h1*.

```
1 <h1 style = "deklaracja;deklaracja;deklaracja" ></h1>
```

Listing 1.3: Styl lokalny

Styl ten zapisuję się jako atrybut znacznika *HTML*. Deklaracje wymienione są w cudzysłowie, którego pominąć nie można tak jak w przypadku innych atrybutów *HTML*. Pominięcie cudzysłowia spowoduje unieważnienie stylu i nie zadziała on w żadnej przeglądarce. Innym sposobem na umieszczenie *CSS* w kodzie *HTML* jest wewnętrzny arkusz stylów. Podobnie jak w przypadku stylu lokalnego ma on ograniczony zasięg, jednak reguła ta będzie się odnosić nie do jednego elementu lecz całego dokumentu. W listingu 1.4 przedstawiony został arkusz dla wersji *HTML5*.

```
1 <head>
2  <style>
3   selektor { deklaracje }
4  </style>
5 </head>
```

Listing 1.4: Wewnętrzny arkusz stylów

Najczęściej stosowanym sposobem i zarazem najlepszą metodą na zmianę wyglądu całej struktury za jednym zamachem jest dołączenie zewnętrznego arkusza stylów. Aby to zrobić w sekcji *<head>* każdego dokumentu *HTML* dodajemy wiersz kodu widoczny na listingu 1.5 w linii 2.

```
1 <head>
2 <link href="style.css" rel="stylesheet" >
3 </head>
```

Listing 1.5: Zewnętrzny arkusz stylów

Atrybut *href* odpowiada za ścieżkę dostępu do zewnętrznego arkusza stylów. Drugi atrybut musi mieć wartość *stylesheet* i nie możemy go pominąć. Jego brak sprawi, że arkusz nie zadziała. W *HTML* nie musimy podawać atrybutu *type* o wartości *text/css*, bowiem dla znacznika *<link>* został on ustawiony atrybutem domyślnym. Do jednego dokumentu *HTML* można jednocześnie dołączyć kilka zewnętrznych arkuszy stylów, jednak najwyższy priorytet zawsze będzie miał arkusz dołączony na samym końcu. Innym ważnym atrybutem odpowiadającym za przełączanie pomiędzy alternatywnymi arkuszami jest *title* a jego wartość jest jego nazwa.

Ważnym pojęciem stosowanym w kaskadowych arkuszach stylów, jest kaskadowość. Funkcja ta odpowiada za określenie hierarchii stosowanych stylów w dokumencie. Zasada kaskadowości przyjęta przez twórców polega tym, że najpierw ładowane i uwzględnianie są zewnętrzne arkusze, następnie style wpisane do nagłówka *<head></head>*, a na samym końcu style wpisane bezpośrednio do znacznika. Takie rozwiązanie umożliwia pełną kontrolę nad dokumentem, a w przypadku sprzeczności zdefiniowanych stylów użyty zostanie ten, który jest najbliżej formatowanego dokumentu[1].

[1] `http://webmaster.helion.pl/index.php/pcss-dziedziczenie-i-kaskadowosc` czas dostępu:2016-04-21

Za pomocą dziedziczenia polegającego na przejmowaniu przez elementy podrzędne właściwości przodków *CSS* oszczędza twórcy strony wprowadzania tych samych cech dla każdego elementu z osobna.

```
1 <!DOCTYPE html>
2 <html>
3  <head>
4   <title>Dziedziczenie w CSS</title>
5   <style type="text/css">
6    table {
7     font-family: Verdana;
8     }
9    </head>
10  <body>
11   <table>
12   <tr>
13    <th>Kolumna</th>
14   </tr>
15   <tr>
16    <td>Komorka</td>
17   </tr>
18   </table>
19  </body>
20 </html>
```

Listing 1.6: Dziedziczenie w CSS

Widoczny na listingu 1.6 przykład przedstawia dziedziczenie przez elementy potomne *<tr>*, *<th>* i *<td>*, właściwości *font-family*. Ciało dokumentu *<body>* (element nadrzędny) nie dziedziczy czcionki, bowiem jest przodkiem tabeli, a nie jej potomkiem. Jednak nie wszystkie cechy przekazywane są z pokolenia na pokolenia. Marginesy i dopełnienia są przykładami cech, będącymi wyjątkiem od tej reguły. Jednak jeśli jakaś cecha nie jest dziedziczona, dzięki *inherit* jesteśmy w stanie wymusić nadanie tej właściwości.

Podstawowa konstrukcja stylu *CSS* posiada selektor oraz deklaracje stylu. Selektor jest elementem *HTML* do którego będą stosowane deklaracje. Deklaracja stylu umieszczana jest zawsze w nawiasie klamrowym i określa właściwości umożliwiające modyfikację wglądu lub zachowania elementu. Najprostszym selektorem jest zwykły selektor elementu przedstawiony na listingu 1.7. Przypisanie znacznikowi *p* właściwości *color: red;* spowoduję, że wszystkie elementy o tej samej nazwie zostaną sformatowane zgodnie z definicją. Selektor uniwersalny widoczny na listingu 1.8,

```
1 p{
2    color:red;
3 }
```

Listing 1.7: Selektor elementu

przypisuję właściwość *color: green;* do wszystkich elementów języka *HTML*.

```
1 *{
2  color: green;
3  }
```

Listing 1.8: Selektor uniwersalny

Selektor klasy pozwala nam nadać identyczne formatowanie pogrupowanym przez nas elementom. Listing 1.9 przedstawia dwie zasady dotyczące klas, pierwsza z nich mówi, że ta sama klasa może być użyta w wielu elementach oraz więcej niż jedna klasa może opisywać element.

```
1 p.klasa1{
2  font-size: 10px;
3 }
4
5 h1.klasa2{
6 color: yellow;
7 }
8
9 <p class="klasa1">tekst akapitu</p>
10 <h1 class="klasa1 klasa2">tekst naglowka</p>
```

Listing 1.9: Selektor klasy

Selektorem pozwalającym odniesienie się do konkretnego elementu bez naruszenie pozostałych jest identyfikator widoczny na listingu 1.10. Możemy go nadać dla dowolnego elementu języka *HTML*. W przeciwieństwie do klas, każdy element może mieć tylko jeden identyfikator, który jest unikalnym atrybutem i może wystąpić tylko raz w danym dokumencie.

```
1 #identyfikator {
2  font-size: 50px;
3  }
4
5  <p id="identyfikator">tekst akapitu</p>
6
```

Listing 1.10: Selektor identyfikatora

1.1.3 JavaScript

JavaScript to język programowania pozwalający stworzyć program, który może być włączony do strony *HTML*. Zamknięty jest w znacznikach *<script>*, dlatego jego tekst nie pojawia się na ekranie, a przeglądarka „wie", że zamiast wyświetlać tekst skryptu, ma go wykonać. Znacznik *<script>* najczęściej można znaleźć w sekcji *<head>* strony *HTML*, chociaż skrypty można też umieszczać w sekcji *<body>*. W tym miejscu najlepiej umieszczać skrypty, które wypisują tekst na ekranie lub tworzą kod *HTML* [12].

Język *JavaScript* został opracowany przez *Netscape Communications Corporation* – firmę, która stworzyła przeglądarkę *WWW*. *JavaScript* był pierwszym językiem skryptowym obsługiwanym przez przeglądarki i nadal pozostaje zdecydowanie najpopularniejszym z nich. Według ECMA – która standaryzuje ECMAScript, podzbiór JavaScriptu – językiem skryptowym nazywamy język programowania stworzony do manipulowania, dostosowywania i automatyzowania obiektów istniejącego już systemu. Poprzez interfejs użytkownika w systemie dostępna jest użyteczna funkcjonalność a język skryptowy jest mechanizmem, który uwidacznia tą funkcjonalność programując kontrole. Tym sposobem komunikujemy istniejącemu systemowi by zapewnił środowisko obiektów i udogodnień co uzupełnia możliwości języka skryptowego.

Dzięki językowi *JavaScript* jesteśmy w stanie wyświetlać wszelkiego rodzaju komunikaty dla użytkownika w obrębie strony *WWW*, jak i na pasku stanu przeglądarki oraz w ramkach ostrzegawczych. Kontrolować czy użytkownik poprawnie wypełnia dane formularza lub przeprowadzać obliczenia. Tworzyć paski reklamowe pozwalające na interakcje z użytkownikiem zamiast wyświetlanie zwykłych obrazków. Wykrywać typy przeglądarek oraz wykonywać zaawansowane funkcje w tylko tych przeglądarkach, które je obsługują. Sprawdzać zainstalowane moduły dodatkowe i powiadamiać użytkownika, gdy jest potrzeba zainstalowania dodatkowego modułu. Użytkownik nie jest zmuszany do przeładowania strony gdy modyfikujemy jej część.

```
1 <html>
2  <head>
3   <title>Data i godzina</title>
4   <meta http-equiv="Content-Type" content="text/html">
5  </head>
6  <body>
7   <h1>Data i czas</h1>
8   <p>
9    <script language="JavaScript" type="text/javascript">
10    var teraz = new Date();
11    var czasLokalny = teraz.toString();
12    document.write("<b>Czas lokalny:</b>" + czasLokalny);
13    </script>
14   </p>
15  </body>
16 </html>
17
```

Listing 1.11: Skrypt w JavaScript

Widoczny na listingu 1.11 przykładowy skrypt umieszczony w znaczniku *HTML* wyświetli bieżący czas w przeglądarce użytkownika. Za pomocą instrukcji *teraz = new Date();* do zmiennej *teraz* przypiszemy obiekt typu *Date()*. Zmienne w *JavaScripcie* są kontenerami, które przechowują różnego typu dane (np. liczbę, łańcuch tekstu lub obiekt). Nazwa zmiennej może zawierać litery duże jak i małe alfabetu oraz cyfry od 0 do 9. Istnieją dwa typy zmiennych: globalna i lokalna. Zmienna globalna, która ma zasięg we wszystkich skryptach dokumentu *HTML*, powinna

być zadeklarowana w głównym skrypcie, na zewnątrz wszelkich funkcji. Natomiast zmienna lokalna może być używana tylko w funkcji, w której została utworzona. Metoda *toString()* obiektu *teraz* skonwertowała datę do zmiennej typu *String*, dzięki czemu funkcja *document.write()* wyświetliła aktualny czas użytkownika.

Komentarze w JavaScript umożliwiają umieszczenie dokumentacji kodu wewnątrz skryptu. Pozwala to innym zrozumieć działanie skryptu lub zapobiega egzekucji kodu gdy testujemy alternatywne rozwiązania. Aby zawrzeć komentarz w programie JavaScript do wyboru mamy dwa typy komentarzy. Liniowy – widoczny na listingu 1.12 – zaczyna się od dwóch ukośników a kończy przy przejściu do następnej linii. Drugim typem komentarzy obsługiwanym przez JavaScript jest styl języka C, nazywany blokowym. Rozpoczyna się od sekwencji znaków /* a kończy tymi samymi znakami tylko w odwrotnej kolejności. Komentarze takie mogą zawierać więcej niż jeden wiersz kodu, jak w listingu 1.12 wierszu 2 i 3. Instrukcje JavaScript w komentarzu są ignorowane więc przy usuwaniu błędów w skrypcie gdy potrzebujemy chwilowo wyłączyć część sekcji programu możemy dodać /* na początku i */ na końcu komentując fragment kodu.

```
1 //Komentarz liniowy
2 /*Komentarz blokowy wiersz pierwszy
3 Komentarz blokowy wiersz drugi*/
```

Listing 1.12: Komentarz liniowy i blokowy

1.2 Biblioteki języka JavaScript

Biblioteki *JavaScript* to fragmenty kodu napisanego w tym języku, które zawierają rozwiązania wielu prozaicznych zadań wykonywanych każdego dnia przez programistów. Można je sobie wyobrazić jako kolekcje gotowych funkcji *JavaScriptu*, które wystarczy dodać do strony. Funkcje te ułatwiają wykonywanie najczęściej spotykanych zadań. Często zdarza się, że bardzo wiele wierszy naszego własnego kodu (oraz sporo godzin koniecznych do jego napisania) można zastąpić wywołaniem jednej funkcji takiej biblioteki. Istnieje bardzo dużo takich bibliotek, a wielu z nich używano podczas tworzenia największych i najbardziej znanych stron *WWW*, takich jak Yahoo, Amazon, CNN, Apple oraz Twitter [11]. Kilka najbardziej znanych bibliotek:

- Yahoo User Interface Library (*http://developer.yahoo.com/yui/*) – projekt stworzony i używany przez Yahoo. Twórcy rozwijają i poprawiają tą biblioteka, posiada także dobrą dokumentacje.
- Dojo Toolkit (*http://dojotoolkit.org/*) – kolekcja bardzo wielu plików *JavaScript* mogących rozwiązać nurtujący nas problem związany z tworzeniem skryptów.
- Mootools (*/http://mootools.net/*) – biblioteka pomagająca tworzyć płynne animacje i innych efektów wizualnych.
- Prototype (*http://www.prototypejs.org/*) – jedna z pierwszych bibliotek *JavaScript* jaka się pojawiła. Zawiera przydatne rozszerzenia do środowiska skryptowego przeglądarki oraz zapewnia eleganckie *API* dla interfejsu *Ajax*.

1.2.1 jQuery

jQuery jest biblioteką języka *JavaScript*, która przyśpiesza i ułatwia programowanie ponieważ samoczynnie wykonuję złożone zadania. Pozwala łatwo manipulować stroną *HTML* po jej wyświetleniu przez przeglądarkę. Dostarcza narzędzi, które pomagają nam nasłuchiwać co użytkownik wykonuję na stronie, do tworzenia animacji, oraz funkcje pozwalające komunikowanie się z serwerem bez przeładowywania strony.

Podobnie jak w przypadku zewnętrznych plików *JavaScript* skrypty *jQuery*, by zaczęły działać muszą być dołączone do swojej strony. Istnieje kilka sposobów pozwalające nam to zrobić: można pobrać bibliotekę udostępnioną przez *Google*, *Microsoft* lub *jQuery.com* albo skopiować plik biblioteki na własny komputer i umieścić go na stronie. Korzystając z sieci dystrybucji treści (ang. *content distribution network*) – innej strony przechowującej i udostępniającej bibliotekę–obniżamy obciążenie własnego serwera, ponieważ to serwery *Google* czy *Microsoftu* użyczają bibliotekę użytkownikom przeglądającym witrynę. Inną korzyścią tego rozwiązania jest, że serwery tych firm rozmieszczone są na całym świecie, więc osoba przeglądająca stronę pobierze plik biblioteki z serwera położonego znacznie bliżej swojego miejsca zamieszkania niż nasz, co przyspieszy transfer i nasza witryna szybciej zadziała. Takie rozwiązanie ma też ważną wadę, użytkownik przeglądający naszą witrynę musi być podłączony do internetu by ściągnąć plik pozwalający poprawne działanie strony. Listing 1.13 przedstawia pobranie biblioteki z serwera Google.

```
1 <script src="https://ajax.googleapis.com/ajax/libs/jquery/2.1.3/jquery.
    min.js"></script>
```

Listing 1.13: Pobranie biblioteki z serwera Google

Drugim sposobem dodania biblioteki *jQuery* jest pobranie i umieszczenie pliku na stronie. Plik ten jest zewnętrznym plikiem *JavaScriptu* i dodaje się go do strony poprzez komendę widoczną na listingu 1.14.

```
1 <script src="jquery-2.1.3.min.js"></script>
```

Listing 1.14: Dodanie zewnętrznego pliku jQuery

Funkcje są potrzebnym narzędziem, dzięki którym w wydajny sposób możemy wielokrotnie powtarzać te same operacje. W *jQuery* funkcje mogą być zapisywane w skrócie jako *$()* i wykorzystywane na jeden z czterech sposobów. Pierwszy sposób – widoczny na listingu 1.15 – jako pierwszy parametr przyjmuje inną funkcje o nazwie *powitanieAplikacji()* w wierszu 11.

```
1  <!DOCTYPE html>
2  <html>
3   <head>
4    <title>Funkcja jako parametr</title>
5    <script type="text/javascript" src="jquery-2.1.3.min.js"></script>
6    <script type="text/javascript">
7     function powitanieAplikacji()
8     {
9      alert('Hello');
10    }
```

```
11     $(powitanieAplikacji);
12    </script>
13   </head>
14   <body>
15     ...
16    </body>
17 </html>
```

Listing 1.15: Wywołanie funkcji przyjmującej inną funkcje jako parametr

Drugim sposobem wywołania funkcji *jQuery* – pokazany na listingu 1.16 – jest przekazanie kodu XHTML jako pierwszego parametru. W wierszu 8 utworzyliśmy nowy węzeł DOM (ang. *Document Object Model*) i dowiązaliśmy go do bieżącego dokumentu wywołując metodę *.appendTo()*. Należy pamiętać by instrukcja widoczna w wierszu 8 była zapisana w jednym wierszu, ponieważ napisy w języku *JavaScript* nie mogą wydłużać się na wiele wierszy. Gdy napis rozciągnie się na więcej niż jeden wiersz otrzymamy błędy kod *JavaScriptu*.

```
1 <!DOCTYPE html>
2 <html>
3  <head>
4   <title>Kod XHTML jako parametr</title>
5   <script type="text/javascript" src="jquery-2.1.3.min.js"></script>
6   <script type="text/javascript">
7   $(function(){
8     $('<div id="nowyElement"><p>Hello</p></div>').appendTo('body');
9   });
10  </script>
11 </head>
12 <body>
13   ...
14  </body>
15 </html>
```

Listing 1.16: Wywołanie funkcji przyjmującej jako pierwszy parametr kod XHTML

Trzecim sposobem przedstawionym na listingu 1.17 jest wywołanie funkcji z selektorem jako pierwszym parametrem. W wierszu 10 wywołanie przyjmuję jeszcze jeden opcjonalny parametr, nazywany *kontekstem*. Kontekst jest obiektem *DOM* lub *jQuery*, ograniczający zasięg selektora. Kod w wierszu 8 i 9 utworzy zmienną *g*, która będzie obiektem *jQuery* zawierający identyfikator *#gora*.

```
1 <!DOCTYPE html>
2 <html>
3  <head>
4   <title>Selektor jako parametr</title>
5   <script type="text/javascript" src="jquery-2.1.3.min.js"></script>
6   <script type="text/javascript">
7   $(function(){
```

```
8   var g = $('#gora');
9   $('h1' , g).css('color' , 'green');
10  }):
11  </head>
12  <body>
13   <div>
14    <h1>Naglowek</h1>
15   </div>
16   <div id="gora">
17   <h1>Naglowek2</h1>
18   </div>
19  </body>
20 </html>
```

Listing 1.17: Wywołanie funkcji przyjmującej jako parametr selektor

Kolejnym sposobem wywołania funkcji *$()*, jest przyjęcie jako pierwszy parametr element drzewa *DOM* lub tablice takich elementów. Widoczna na listingu 1.18 w wierszu 8 i 9, zmienna *element* przyjmuję wynik działania metody *getElementById()* i wywołana zostaje funkcja z tym elementem drzewa *DOM*.

```
1 <!DOCTYPE html>
2 <html>
3  <head>
4   <title>Element DOM jako parametr</title>
5   <script type="text/javascript" src="jquery-2.1.3.min.js"></script>
6   <script type="text/javascript">
7   $(function(){
8   var element = document.getElementById('gora');
9   $(element).text('...');
10    }):
11  </head>
12  <body>
13   <div id="gora">
14   </div>
15 </body>
16 </html>
```

Listing 1.18: Wywołanie funkcji przyjmującej jako parametr element drzewa DOM

1.2.2 jQuery Mobile

jQuery Mobile jest zbiorem wtyczek i kontrolek mających na celu tworzenie mobilnych aplikacji internetowych na różne platformy. W odniesieniu do implementacji kodu, *jQuery Mobile* przypomina bardzo bibliotekę *jQuery UI*, z tym że *jQuery UI* skoncentrowana jest na tworzenie aplikacji desktopowych, zaś *jQuery Mobile* przeznaczona jest do aplikacji mobilnych. W bibliotece tej aby uzyskać kon-

sekwentny efekt dla różnych platform mobilnych użyto atrybutów *HTML5* i właściwości *CSS3* by wzbogacić podstawowe znaczniki *HTML*. Dzięki skorzystaniu ze specyfikacji *HTML5* w zakresie spersonalizowanych atrybutów obsługi danych możliwe jest osadzenie danych w kodzie znaczników *HTML5*. Po załadowaniu biblioteki *jQuery Mobile* wybiera ona elementy na postawie ich atrybutów obsługi danych, po czym poprawia je poprzez wprowadzenie nowych znaczników, dodanie procedur obsługi zdarzeń, oraz stworzenie nowych klas *CSS*. Dzięki temu można szybko stworzyć podstawowy kod złożony z podstawowych znaczników, a *jQueryMobile* przekształci to na złożone elementy interfejsu użytkownika.

Widoczna na listingu 1.19 podstawowa struktura strony korzystająca z biblioteki *jQuery Mobile* jest przygotowana w specyficzny sposób.

```
1 <!DOCTYPE html>
2 <html>
3  <head>
4  <title>Podstawowa strona</title>
5  <meta name="viewport" content="width=device-width, initial-scale=1">
6  <link rel="stylesheet" href= "http://code.jquery.com/mobile/1.4.5/
       jquery.mobile-1.4.5.min.css">
7  <script type="http://code.jquery.com/jquery-1.11.1.min.js"> </script>
8  <script type="http://code.jquery.com/mobile/1.4.5/jquery.mobile-1.4.5.
       min.css"> </script>
9  </head>
10 <body>
11 <div data-role="page">
12  <div data-role="header"> <h2>Tytul strony</h2>
13 </div>
14 <div data-role="content"> <p>Hello world!</p>
15 </div>
16 </div>
17 </body>
18 </html>
```

Listing 1.19: Prosta strona HTML5 dla aplikacji jQuery Mobile

W wierszu 4 jest znacznik *viewport*, który prosi urządzenie w ustawienie określonego poziomu powiększenia oraz wielkość strony i dopasowanie ich do aktualnie wyświetlanej zawartości. Ma to kluczowe znaczenie przy tworzeniu witryn przeznaczonych dla urządzeń mobilnych. Domyślna wartość tego znacznika jest zależna od przeglądarki ale przeważnie wynosi około 980 pikseli. W przypadku większej lub mniejszej rozdzielczości urządzenia od domyślnej, układ strony może wydawać się zbyt mały lub powiększenie zostanie ustawione w taki sposób, że tekst będzie zbyt mały do wygodnego czytania. Dzięki określeniu atrybutów *width* oraz *initial–scale* rozmiar ekranu urządzenia możemy dostosować do wielkości zawartości.

Określamy używany arkusz stylów *CSS* za pomocą instrukcji widocznej w wierszu 6. Przy jego użyciu można zmienić całkowity wygląd strony. Trzeba jednak wziąć pod uwagę liczbę urządzeń, na jakich witryna zostanie przetestowana. Domyślnie biblioteka *jQuery Mobile* testowana jest na dużej liczbie urządzeń.

W wierszach 8 i 7 znajdują się znaczniki pobierające biblioteki *jQuery Core* oraz *jQuery Mobile*. Sama zawartość strony umieszczona została w wierszach 11–16. W większości jest to zwyczajny kod *HTML*. Jednak każdy znacznik *div* zawiera atrybut *data-role* z przypisaną wartością: *page*, *header*, *content*. Biblioteka *jQuery Mobile* przypisuje za pomocą tych znaczników o określonych rolach, poszczególnym elementom witryny pasujący styl, temat czy działanie.

Biblioteka *jQuery Mobile* dzięki atrybutowi *data-role* oznacza strony oraz elementy na podstawie ich atrybutów, po czym ulepsza wprowadzając klasy *CSS*, potrzebne znaczniki oraz obsługę zdarzeń. Stosowanie takiej metody przez *jQuery Mobile* prowadzi do płynnych przejść między stronami, automatycznego obsługiwania nawigacji oraz lepszej wydajności ponieważ zasoby zapisane są w jednym pliku. Na listingu 1.20 przedstawiona została metoda, która za pomocą atrybutu *data-role="page"* widocznego w wierszu 12 i 21 oznaczyła oddzielne sekcje treści jako strony. Nie jest możliwe zagnieżdżanie stron wewnątrz siebie, sekcje odpowiadające za strony muszą być rodzeństwem najwyższego poziomu drzewa dokumentu.

Biblioteka *jQuery Mobile* jest w stanie zarządzać także stronami zewnętrznymi. Gdy łącze zostanie zdefiniowane do osobnej strony zamiast do ID elementu *data-role="page"* będącego w bieżącym dokumencie *jQuery Mobile* asynchronicznie pobiera żądaną stronę. Następnie przeszukuję ją w by znaleźć pierwszy element oznaczony atrybutem *data-role="page"* i wstawia go do dokumentu źródłowego. Reszta treści, razem z kolejnymi elementami z atrybutami *data-role="page"* jest ignorowana. Gdyby ściąganie strony zakończyło się niepomyślnie lub *jQuery Mobile* nie znajdzie elementu oznaczonego szukanym atrybutem, wyświetli komunikat o błędzie.

W przypadku gdy chcemy załadować stronę rezygnując by biblioteka *jQuery Mobile* pobrała ją asynchronicznie i scaliła z bieżącym modelem DOM możemy załadować stronę z wykorzystaniem technologii *AJAX* określając atrybut *target* łącza (na przykład na "blank") lub ustawiając atrybut *rel="externa"*[7].

```
1 <!DOCTYPE html>
2 <html>
3  <head>
4  <title>Strony wewnetrzne</title>
5  <meta name="viewport" content="width=device-width, initial-scale=1">
6  <link rel="stylesheet" href= "http://code.jquery.com/mobile/1.4.5/
      jquery.mobile-1.4.5.min.css">
7  <script type="http://code.jquery.com/jquery-1.11.1.min.js"> </script>
8  <script type="http://code.jquery.com/mobile/1.4.5/jquery.mobile-1.4.5.
      min.css"> </script>
9  </head>
10  <body>
11  <!--Strona pierwsza-->
12 <section id="strona1" data-role="page">
13 <header data-role="header"><h1>Przyklad strony</h1></header>
14 <div data-role="content" class="content">
15 <p>Strona Pierwsza</p>
16 <p><a href="#strona2">Przejscie do strony drugiej</a></p>
17 </div>
18 </section>
19
20 <!--Strona druga-->
21 <section id="strona2" data-role="page">
22 <header data-role="header"><h1>Przyklad stron wewnetrznych</h1></header
      >
23 <div data-role="content" class="content">
24 <p>Strona druga</p>
25 <p><a href="#strona1">Przejscie spowrotem do strony pierwszej</a></p>
26 </div>
27 </section>
28  </body>
29 </html>
```

Listing 1.20: W pojedynczym dokumencie HTML zawarte są dwie wewnętrzne strony

1.3 Reprezentacja danych

Wraz z rozpowszechnieniem internetu oraz jego zasobów informacyjnych w postaci stron *WWW*, pojawił się pomysł do wykorzystania ich jako baz danych. Jednak dane przechowywane w postaci stron *HTML* w przeciwieństwie do relacyjnych baz danych nie mają tak regularnej struktury. Do opisanie takich danych używa się określenia dane semistrukturalne – czyli model danych różniący się od formalnej struktury danych powiązanych z relacyjnymi bazami danych ale zawierający tagi pozwalające rozdzielić oraz wymusić hierarchie rekordów. Podstawą matematyczną

takiego modelu jest graf skierowany, co czynni reprezentacje danych na bardziej elastyczną niż w relacyjnych bazach danych. W semistrukturalnych danych wpisy należące to tej samej klasy mogą posiadać różne atrybuty nawet gdy są pogrupowane razem oraz kolejność atrybutów nie jest istotna.

1.3.1 JSON

JSON (ang. *Java Script Object Notation*) [2] czyli prosty format wymiany danych oparty na języku programowania *JavaScript*. Choć jest całkowicie niezależny od innych języków programowania jest formatem tekstowym używającym zwyczajów praktyk z innych języków jak *C*, *C++*, *Java*, *Perl* czy *Python*. *JSON* ukształtował się na bazie dwóch charakteryzujących go struktur. Pierwszą z nich jest zbiór par nazwa i wartość. W innych językach może występować jako rekord, struktura czy tabela asocjacyjna. Drugą strukturą jest uporządkowana lista wartości. Implementowana za pomocą tabeli czy listy w innych językach. Większość nowoczesnych języków programowania używa tych uniwersalnych struktur danych, dlatego też format danych opiera swoją budowę na przytoczonych strukturach.

```
1 {
2  "uczelnia": {
3   "wyklad": {
4     "typ": "seminarium",
5     "wyklad_tytul": "Podstawy JSON",
6     "wyklad_numer": "9.10",
7     "data_rozpoczecia": "12-12-2015",
8     "studenci": {
9       "student": [
10      {
11      "status":"obecny",
12      "imie": "Edward",
13      "nazwisko": "Nozycoreki"
14      },
15      {
16      "status":"nieobecny",
17      "imie": "Leon",
18      "nazwisko": "Zawodowiec"
19      }
20      ]}
21   }
22 }
```

Listing 1.21: Przykładowa struktura danych w formacie JSON

1.3.2 XML

XML (ang. *Extensible Markup Language*) [3] jest rozszerzalnym językiem znaczników opartym na *Standardowym uogólnionym językiem znaczników* (SGML). Jak

sama nazwa wskazuję *SGML* jest językiem bardzo ogólnym o ogromnych możliwościach, niestety ceną za te możliwości jest złożoność, co utrudnia jego naukę. *XML* służy do definiowania pozostałych języków zwany metajęzykiem. W przeciwieństwie jednak to swego przodka jest prostszy i bardziej praktyczny. W języku tym nie sprecyzowano ani zestawu znaczników, a gramatyki. Zestaw znaczników określa jakie znaczniki są ważne dla parsera znaczników. *XML* nie ma zdefiniowanego ani zestawu znaczników, ani gramatyki. Posiada więc nieograniczone możliwości rozbudowy. Choć elastyczność jest jedną największych zalet *XML*, to jest też jednak jedną z jego wad. Dokumenty *XML* możemy przedstawiać na tyle różnych sposobów, że powstała bardzo duża liczba standardów opisujących tłumaczenia formatów danych i same formaty.

XML pozwala także określić strukturę i sposób umieszczenia jednych elementów w innych. Listing 1.22 przedstawia dokument *XML* opisujący seminarium oraz obecność dwóch studentów. W dokumencie takim możemy wymusić by każdy element *<student>* zawierał dokładnie po jednym elemencie *<imie>* i *<nazwisko>* lub nie zawierał elementu *<wyklad_numer>*.

```
1 <?xml version="1.0" encoding="UTF-8">
2 <uczelnia>
3  <wyklad typ="seminarium">
4   <wyklad_tytul>Podstawy XML</wyklad_tytul>
5   <wyklad_numer>9.10</wyklad_numer>
6   <data_rozpoczenia>12-12-2015</data_rozpoczenia>
7   <studenci>
8    <student status="obecny">
9     <imie>Edward</imie>
10    <nazwisko>Nozycoreki</nazwisko>
11    </STUDENT>
12   <STUDENT status="nieobecny">
13    <imie>Leon</imie>
14    <nazwisko>Zawodowiec</nazwisko>
15   </student>
16  </studenci>
17 </wyklad>
18 </uczelnia>
```

Listing 1.22: Przykładowy dokument XML

Aby przeglądarka uznała dokument *XML* przeprowadza dwie kontrole. Pierwsza polega na sprawdzeniu czy dokument jest poprawnie sformułowany czyli spełnia ustawione przez *W3C* specyfikację *XML* 1.0 dotyczącej składni. Oznacza to istnienie przynajmniej jednego elementu w dokumencie oraz wystąpienie elementu głównego, który zawiera wszystkie inne elementy występujące w dokumencie. Każdy element musi koniecznie być zamknięty całkowicie w elementach nadrzędnych względem niego. Przykład na listingu 1.23 przedstawia niepoprawnie sformułowany dokument ponieważ widoczny w wierszu 6 znacznik końcowy *</naglowek>* występuje po znaczniku otwierającym następnego elementu *<temat>*.

```
1 <?xml version="1.0" encoding="UTF-8">
```

```
2 <dokument>
3  <naglowek>
4    Hello
5  <temat>
6  </naglowek>
7    Zle ustawione znaczniki
8    </temat>
9 </dokument>
```

Listing 1.23: Niepoprawnie sformułowany dokument XML

Druga kontrola przeprowadzana przez przeglądarkę jest walidacja dokumentu, która jest opcjonalna *XML*. Dokument *XML* można walidować, gdy definicja typu dokumentu jest z nim związana i dokument jest z nią zgodny. Definicja typu dokumentu (*DTD*) ustala jego prawidłową składnię. *DTD* można przechowywać na dwa sposoby: w osobnym pliku lub w tym samym dokumencie jak przestawia listing 1.24.

```
1 <?xml version="1.0" encoding="UTF-8">
2 <?xml-stylesheet type="text/css" href="first.css"?>
3 <!DOCTYPE DOKUMENT [
4    <!ELEMENT DOKUMENT (NAGLOWEK, TEMAT)>
5    <!ELEMENT NAGLOWEK (#PCDATA)>
6    <!ELEMENT TEMAT (#PCDATA)>
7    ]>
8 <dokument>
9  <naglowek>
10   Hello
11 <temat>
12 </naglowek>
13   Zle ustawione znaczniki
14   </temat>
15 </dokument>
```

Listing 1.24: Definicja typu dokumenty umieszczona w tym samym dokumencie

1.3.3 Atom

Atom [9] jest formatem dokumentu opartym na *XML*, który opisuje listę powiązanych ze sobą informacji znanych jako kanał. *Kanały* są zbudowane z pewnej liczby elementów przedstawionych jako *wpisy*, każdy *wpis* zawiera rozszerzalny zestaw dołączonych metadanych. Dla przykładu w przedstawionym listingu 1.25 **wpis** posiada metadane *<title></title>* w wierszu 13. Standard *Atom* stworzony został z myślą rozwiązania problemu z błędami specyfikacji równoległych standardów *RSS*. Unika on niejasności wyżej wymienionej specyfikacji, poprawia zgodność z *XML* i innym standardami oraz określa protokół publikacji kanałów.

```
1
```

```
2  <?xml version="1.0" encoding="utf-8"?>
3    <feed xmlns="http://www.w3.org/2005/Atom">
4   <link href="http://www.example.com" rel="alternate" type="text/html"
      />
5   <updated>2016-01-11T18:30:02Z</updated>
6       <title>Przyklad wpisu</title>
7       <author>
8         <name>Jan Kowalski </name>
9       </author>
10      <id>urn:uuid:60a76c80-d399-11d9-b93C-0003939e0af6</id>
11
12      <entry>
13        <title>Przykladowy tytul hasla</title>
14        <link href="http://example.org/2003/12/13/atom03"/>
15        <id>urn:uuid:1225c695-cfb8-4ebb-aaaa-80da344efa6a</id>
16        <updated>2016-01-13T18:30:02Z</updated>
17        <summary>Jakis tekst.</summary>
18      </entry>
19
20    </feed>
```

Listing 1.25: Przykład wpisu w Atom

Wiele z elementów *Atom* posiadają kilka podobnych struktur, takie jak struktura tekstu, daty czy osoby. Kiedy element jest zdefiniowany jako określona typ struktury, dziedziczy on wyszczególnione wymogi z definicji struktury tej sekcji. Struktura tekstu *Atom* zawiera zazwyczaj małej zawartości tekst czytelny dla człowieka. Przedstawiona na listingu 1.26 struktura zawiera atrybut *type* w wierszu 4 oznacza to, że wartość tej struktury musi być typu "textlub "html". Jeżeli nie zdefiniujemy tego atrybutu, *Atom* domyślnie będzie się zachowywał jakby atrybut ten był ustawiony na "text".

```
1
2  atomPlainTextConstruct =
3        atomCommonAttributes,
4        attribute type { "text" | "html" }?,
5        text
```

Listing 1.26: Struktura tekstu w Atom

Struktura osoby jest elementem opisującym osobę, korporację lub inne podobne hasło. W specyfikacji tej kolejność pojawiania się elementów pochodnych nie ma znaczenia. Element *name* jako wartość przyjmuje czytelną dla człowieka nazwę oraz opisana osoba może posiadać tylko jeden taki element. Ta sama reguła obowiązuję dla pozostałych elementów *uri* oraz *email*, z wyjątkiem, tego że *uri* zawiera adres *IRI* do tej osoby a element *email* jego adres e-mail.

Struktura daty odpowiada za przedstawienie daty zgodnej ze standardem internetowym. W dodatku wielka litera ”T” musi być użyta w celu oddzielenia daty od czasu, oraz litera ”Z” gdy nie podamy numerycznie strefy czasowej. Powinna być

także dokładna jak to tylko możliwe, czyli w złym zwyczaju było by ustawienie dla systemu publikacji zastosować takie same znaczniki czasu określające kilka wpisów, które miały miejsce tego samego dnia.

Rozdział 2

Platforma iOS

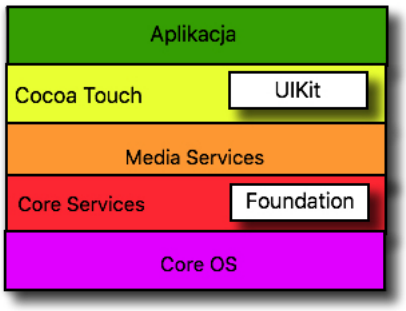
2.1 Mobilny system operacyjny

Mobilny system operacyjny jest oprogramowaniem pozwalającym innym programom zwanym aplikacjami uruchamiać się na urządzeniach przenośnych takich jak telefony komórkowe, smartfony czy tablety. Przez ostatnie lata projekt mobilnych systemów operacyjnych przeszedł trzy fazową ewolucję: z systemu opartego na komputerach PC do systemów komputerowych specjalnego przeznaczenia, aby zakończyć na obecnie obowiązującym systemie zorientowanym na smartfony. W całym procesie przemiany zmiany nastąpiły także w architekturze mobilnych systemów ze skomplikowanej przez mniej złożoną do czegoś pośrodku. Naturalnie cały proces ewolucyjny napędzany jest przez postęp technologiczny w trzech dziedzinach. Sprzęcie komputerowym, gdzie przemysł zmniejszył rozmiar mikroprocesorów i urządzeń peryferyjnych pozwalający na stworzenie dzisiejszych urządzeń mobilnych. Przed tymi zmianami nie było możliwe osiągnięcie tak wysokiej wydajności przy tak małych gabarytach. Drugą dziedziną jest oprogramowanie, które w komputerach przenośnych jest skoncentrowane na produktywności użytkownika, dlatego też wsparcie dla myszki i klawiatury było sprawą kluczową. Oprogramowanie na urządzenia mobilne pozwalały jedynie na zarządzanie kontaktami, wysyłanie wiadomości tekstowych itd. Przy czym korzystanie z tych niewielu funkcjonalności bez obsługi ekranu dotykowego zawierało stosunkowo dużo czasu i ograniczało produktywność. Ostatnią dziedziną, która miała wpływ na rozwój mobilnego systemu operacyjnego jest Internet. Wraz z rozwojem Internetu w szczególności po Web 2.0, istnieje masa informacji w sieci, którą trzeba wydobyć, przeszukać, zorganizować i dostarczyć do użytkownika. Ludzie coraz częściej żyją w internecie zamiast po prostu przeglądać zawarte w nim informacje. Powyższy rozwój technologiczny zaowocował wieloma konkurencyjnymi systemami operacyjnymi dostarczone przez różne firmy, takie jak *Android* od *Google*, *iOS Appla*, *Windows Phone Microsoftu* czy *BlackBerry OS* od *RIM*.

2.1.1 Charakterystyka iOS

Mobilny system operacyjny *iOS* został stworzony w 2007 roku jeszcze pod nazwą iPhone OS przez firmę *Apple Inc.* wyłącznie na urządzenia tejże firmy. Jest

komercyjnym system opartym na ogólnie dostępnym rdzeniu systemu *Darwin*. Jak większość obecnych systemów operacyjnych, *iOS* używa interfejsu graficznego jednakże jest to system przeznaczony na urządzenia mobilne i zaprojektowany pod kątem sterowania dotykiem takim jak przeciąganie palcem po ekranie, czy pojedyncze stuknięcie. Gesty takie pozwalają na intuicyjną kontrole nad elementami interfejsu czy przeprowadzenia różnorakich operacji. Wbudowane w urządzenie akcelerometry obsługują dodatkowo gesty typu potrząsanie czy obracanie urządzenia.



Rysunek 2.1: Architektura systemu iOS

Architektura systemu *iOS* widoczna na rysunku 2.1 wywodzi się z *Mac OS X* oraz na przenośnym interfejsie systemu operacyjnego *UNIX*. Stworzony jest na bazie czterech warstw abstrakcji: *Core OS*, *Core Services*, *Media Services*, *Cocoa Touch*.

Core OS jest jądrem systemu operacyjnego, czyli bazowa warstwa systemu obsługująca bezpośrednio sprzęt. Odpowiedzialna jest między innymi za przetwarzanie obrazów, bluetooth, dźwięk, DNS, prywatne/publiczne klucze, system obsługujący wątki, zarządzanie pamięcią.

Core Services - podstawowe usługi systemowe, które są podzielone na rożne biblioteki napisane w języku C i *Objective-C*. Najważniejszymi bibliotekami przekazanymi w *Core Services* są:

- **Foundation Framework** – Podstawowa biblioteka stosowana w *Objective-C* zawierająca zestawy klas przyśpieszające prace z językiem. Klasy tej biblioteki zaczynają się od liter *NS* co jest skrótem od nazwy systemu *NextStep*.
- **Core Foundation Framework** - biblioteka oparta na języku programowania C i pozwala na pracę z typami danych, kolekcjami oraz obsługę tekstowego typu znaków.
- **Core Data Framework** – biblioteka, która pozwala na zarządzanie modelem przechowywania danych. Zapewnia ogólne i automatyczne rozwiązanie typowych zadań związanych cyklem życia obiektów, zarządzanie wykresem obiek-

tów, zapewnia sprawdzanie czy dane mają poprawne typy, sprawdza spójność między relacjami.

- **Core Location Framework** – rejestruje informacje o położeniu przy pomocy GPS czy sieci WiFi.

Media Services jest warstwą odpowiedzialną za renderowania grafiki, tekstu czy obsługę dźwięku. Do najważniejszych bibliotek tej warstwy można zaliczyć *Core Audio*, *Core Animation*, *Core Text* oraz wydajnym środowiskiem pozwalającym renderowanie grafiki 2D *Core Image*.

Główną biblioteką warstwy *Cocoa Touch* jest *UIKit*, czyli framework napisany w języku *Objective-C* dostarczający różne funkcjonalności potrzebne do stworzenia interfejsu użytkownika dla aplikacji *iOS*. Zawiera także zestaw instrukcji przydatnych w tworzeniu wielowątkowych aplikacji, obsługi dotyku, notyfikacji, czy dostęp do danych urządzenia.

2.1.2 Możliwości developerskie w iOS 9.0

iOS 9 jest dziewiątym z kolei większym uaktualnieniem mobilnego systemu operacyjnego na urządzenia firmy *Apple*. Został zaprezentowany pierwszy raz na konferencji *Apple Worldwide Developers Conference* 8 czerwca 2015 roku, zaś dostępny na urządzeniach od 16 września tego samego roku. W porównaniu ze swoim poprzednikiem *iOS 8* skoncentrowany był bardziej na optymalizacji działania urządzenia takiego jak wydłużenie pracy baterii aniżeli dodanie nowych funkcjonalności do systemu.

Wraz z nowym system *iPad* zyskał możliwość obsługiwania dwóch aplikacji jednocześnie. Funkcja *Slide Over* pozwala użytkownikowi wybrać inna aplikacje od tej, którą w danej chwili używa oraz szybką interakcję z nią. Zaś *Split View* daję możliwość uruchomienia dwóch aplikacji jednocześnie obok siebie na urządzeniu *iPad Air 2*. Funkcja *Picture in Pricture* pozwala użytkownikowi wyświetlanie okienka z treścią wideo podczas wykonywania innych czynności na urządzeniu. Choć to użytkownik decyduje kiedy chce korzystać z dwóch aplikacji w tym samym czasie, deweloper nie ma kontroli kiedy to następuję, musi on jednak pamiętać o dwóch rzeczach przy tworzeniu aplikacji by zapewnić użytkownikowi jak najlepsze doświadczenie z wielozadaniowością. Pierwszą jest by aplikacja zarządzała w efektywny sposób zasobami pamięci urządzenia kiedy dzieli system z drugą aplikacją. Gdy występuję ostrzeżenie o braku pamięci system prewencyjnie zabija aplikacje, która zużywa najwięcej pamięci. Drugą sprawą jest zaadoptowanie odpowiednich wielkości dla interfejsu by wszystko wyglądało poprawnie gdy użytkownik zdecyduje współdzielić ekran z inną aplikacją.

3D Touch daję użytkownikom *iOS 9* dodatkową interakcje z urządzeniem poprzez siłę nacisku dotyku. Na urządzeniach obsługujących tę funkcjonalność możemy wybrać odpowiednie zadania aplikacji takie jak podgląd elementu, otworzenie w innym oddzielnym widoku, poprzez naciśnięcie ikonki programu w głównym ekranie telefonu. Do obsługi tej funkcjonalności dostarczone zostały *API* takie jak:

- **UIApplicationShortcut** – pozwala na dodanie skrótu aplikacji na ekran główny telefonu wraz dodatkowymi funkcjonalnościami.

- **UIKit peek and pop** – to *API* pozwalające w łatwy sposób uzyskać dostęp do dodatkowych treści w obrębie aplikacji.

- **WebView peek and pop** – umożliwia pogląd strony *HTML* poprzez link prowadzący do tej strony.

- **UITouch** – dla określonej siły nacisku można przypisać rożne interakcje z użytkownikiem.

Obojętnie które z tych nowych funkcjonalności użyjemy, developer piszący aplikacje musi sprawdzić czy urządzenie jest kompatybilne z funkcją *3D Touch*,na ten moment jedynie modele 6s i 6s Plus obsługują ten dodatek.

App Thining jest nowym rozwiązaniem firmy *Apple* by użytkownicy ściągali tylko takie komponenty aplikacji potrzebne do jej uruchomienia jakie odpowiadają konkretnemu typowi urządzenia. Np. urządzenia typu IPhone (4s, 5, 5s, 6) ściągać będą tylko grafiki odpowiadające temu typowi zamiast pobierać dodatkowe elementy potrzebne dla urządzeń typu *HD* np. IPad, co skutkować będzie mniejszymi aplikacjami, szybszym ściąganiem i zadowolonymi użytkownikami. Wraz z środowiskiem Xcode 7 mamy możliwość przesłania aplikacji do *iTunes* w pośrednim stanie zwanym Bitcode. W stanie takim kiedy użytkownik będzie chciał pobrać aplikacje, App Store automatycznie stworzy optymalną wersje tej aplikacji(jeśli potrzebna wersje 64–bitową) do pasującego urządzenia[1].

2.2 Środowisko programistyczne Xcode

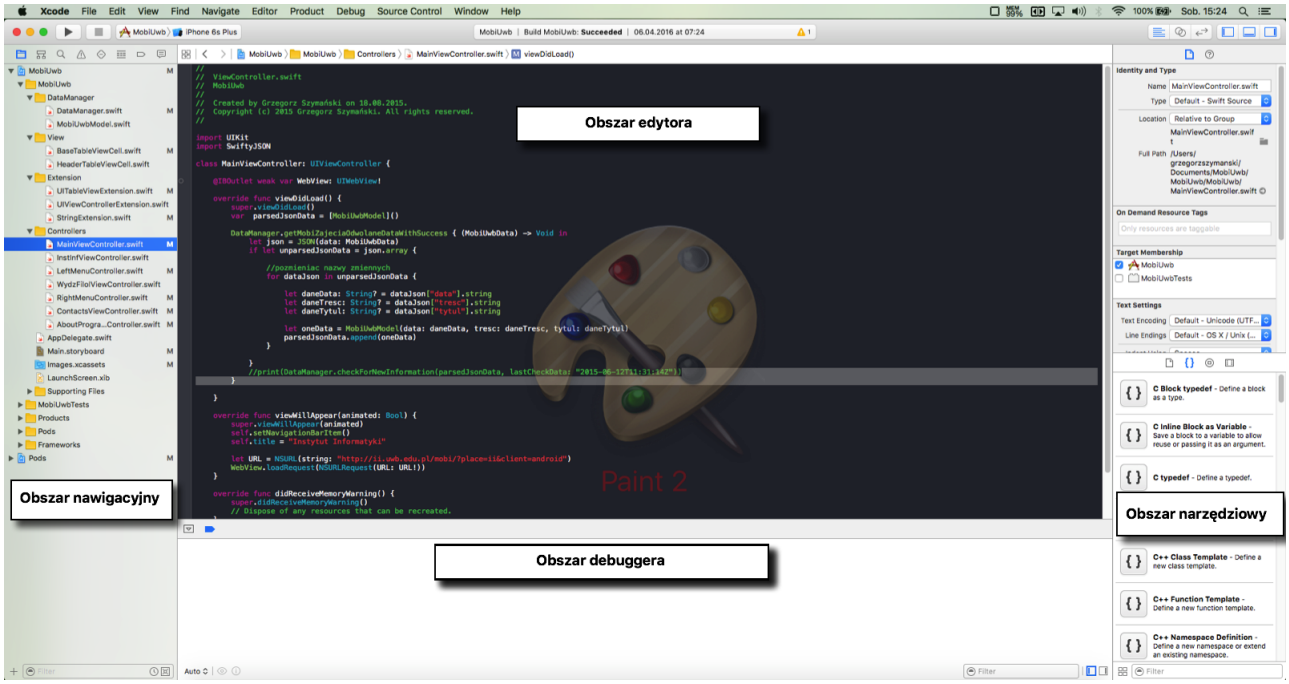
Xcode jest zintegrowanym środowiskiem programistycznym (ang. *Integrated Development Environment*, IDE), inaczej zestawem narzędzi developerskich potrzebnych do stworzenia aplikacji (w szczególności edytor tekstu, kompilator, narzędzie do debugowania, emulator urządzeń iOS), łączy wszystko w jeden pakiet oprogramowania zamiast kilka odrębnym narzędzi połączonych ze sobą skryptem. *Xcode* jest też oficjalnym IDE *Apple'a* do programowania aplikacji na Mac oraz iOS. Pierwotnie znany był jako *Project Builder* w czasach kiedy jeszcze firma Apple istniała pod nazwą *NeXT*, wraz z systemem operacyjnym Mac OS X 10.3 zmieniono nazwę na *Xcode* . Zapewnia wsparcie dla języków programowania takich jak C, C++, Objective–C, Objective–C++, Java, AppleScript, Python, Ruby, Rez oraz Swift, dopuszcza także kompilację w modelach programowania *Cocoa*, *Carbon* i Java.

Okno widoczne na rysunku 2.3 jest pierwszym oknem pojawiającym się przy każdym uruchomieniu *Xcode* przy użyciu ikony programu. Po prawej stronie okna wyszczególniona została lista ostatnio utworzonych projektów oraz ich ścieżkę dostępu. Po lewej stronie oprócz nazwy programu wraz z jego wersją dostępne są trzy opcje. Pierwszą z nich jest rozpoczęcie pracy z plikiem typy *playground*, jest to funkcja pozwalająca w szybki i łatwy sposób przetestować kod bez konieczności tworzenia projektu oraz poświęcaniu uwagi kompilacji. Drugą opcją jest utworzenie nowego projektu *Xcode* na urządzeniu typu iPhone, iPad czy Mac. Trzecią opcją

[1] `https://developer.apple.com/library/ios/releasenotes/General/WhatsNewIniOS/Articles/iOS9.html` czas dostępu:2016-04-21

Rysunek 2.2: Okno powitalne przy uruchomieniu Xcode

jest rozpoczęcie pracy nad istniejącym projekcie poprzez skopiowanie go z repozytorium po uprzednim podaniu wymagających danych typu login, hasło czy link do repozytorium.



Rysunek 2.3: Okno programu Xcode po uruchomieniu projektu

Rysunek 2.3 przedstawia cztery obszary umożliwiające sprawną pracę nad tworzonym projektem. Obszar nawigacyjny daje dostęp do plików, klas, testów czy diagnostyki. Pasek nawigacyjny widoczny na rysunku 2.4 umożliwia nam:

- **Projekt navigator** – Dodaje, usuwa, grupuje i zarządza plikami projektu.

Pozwala na zaznaczeniu interesującego nas pliku i wyświetlenie w obszarze edytora.

- **Symbol navigator** – Wyświetla wszystkie klasy projektu jako listę lub hierarchie.
- **Find navigator** – Pozwala na wyszukanie ciągu znaków w projekcie.
- **Issue navigator** – Pokazuje diagnostykę, ostrzeżenia, błędy, które pojawiają się po zbudowaniu projektu.
- **Test navigator** – Tworzy, zarządza i uruchamia test aplikacji.
- **Debug navigator** – Daje możliwość obsługiwania podstawowych narzędzi do debugowania.
- **Breakpoint navigator** – Wyświetla wszystkie ustawione punkty przerwania oraz wbudowane opcje np. wyłapujące wszystkie wyjątki.
- **Report navigator** – Umożliwia przeglądanie logów tworzonych podczas działania programu jak również uruchamiania aplikacji.

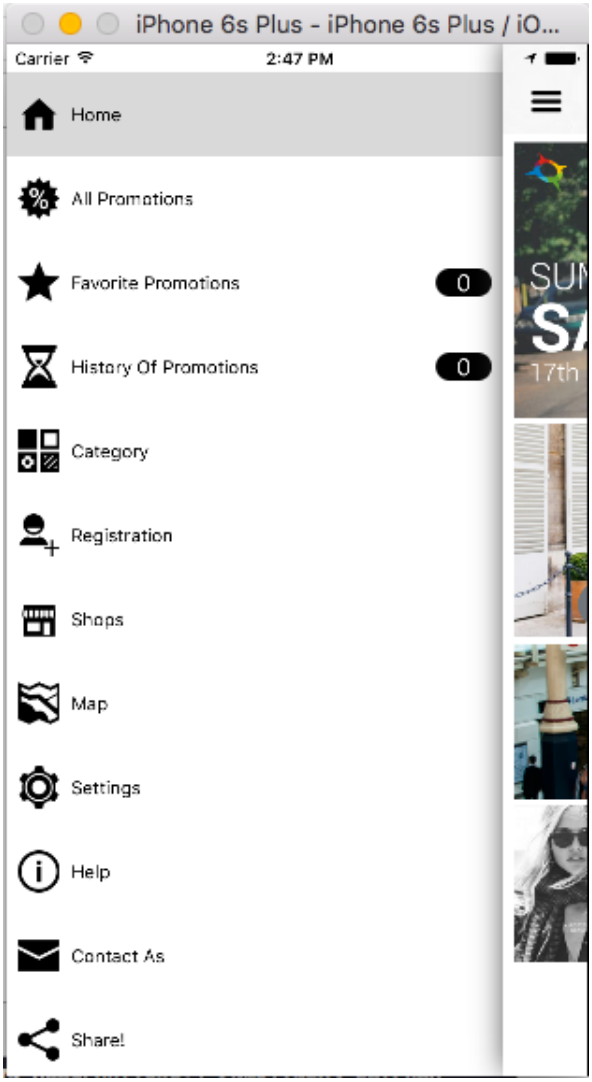
Rysunek 2.4: Pasek nawigacyjny

Zawsze widocznym obszarem okna *Xcode* jest edytor kodu przyśpieszający prace poprzez udostępnienie wielu funkcjonalnych możliwości, takich jak rozpoznanie i wyróżnienie składni, generowanie wcięć, sugerowanie i autouzupełnianie kodu. Umieszczony w prawym górnych rogu przycisk uruchamiający *Assistant Editor* dzieli edytor na dwie części, które pozwalają otworzyć dwa oddzielne dokumenty jednocześnie. Na przykład gdy edytujemy plik *MojaKlasa.m* w nadrzędnym edytorze, *Assistant Editor* automatycznie otworzy plik związany z naszą klasa *MojaKlasa.h*.

Obszar narzędziowy możemy podzielić na dwa odrębne panele. Pierwszy z nich umiejscowiony na górze nazywa się panelem Inspektora i zawiera informację takie jak ścieżka dostępu do pliku, kodowanie tekstu czy danych o systemie kontroli wersji. Panel dolny nazywany panelem Bibliotek dzieli się na cztery elementy zawierające wszystkie pliki multimedialne użyte w projekcie, funkcje i kontrolki biblioteki *Cocoa Touch*, gotowe schematy klas, widoków i kontrolerów oraz zakładkę *Code Snippets* z kawałkami kodów, który można użyć wielokrotnie.

Ostatnim z obszarów z których zbudowane jest okno programu *Xcode* jest obszar debuggera. Kontroluje on wykonanie naszego kodu, wyświetla utworzone zmienne czy wyjściowe parametry w konsoli. Składa się z trzech głównych komponentów:

- **Pasek debuggera** – Kontroluje wykonywanie się aplikacji i zapewnia nawigację po kodzie źródłowym.
- **Sektor widoku zmiennych** – Wyświetla informacje na temat elementów zaznaczonych na pasku debuggera.
- **Sektor konsoli** – Zawiera interaktywny terminal.



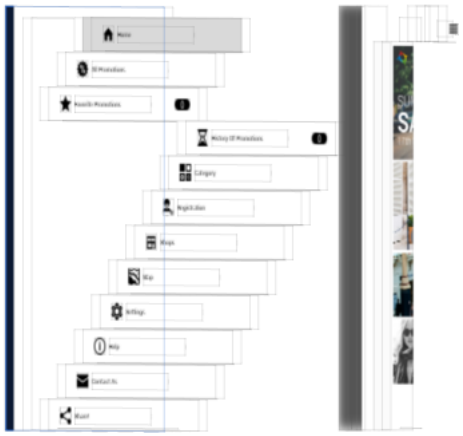
Rysunek 2.5: Symulator urządzenia programu Xcode

Widoczny na rysunku 2.5 zrzut ekranu przedstawia symulator urządzeń programu *Xcode*. Pozwala on w szybki sposób zbudować i przetestować aplikacje w trakcie jej pisania. Zainstalowany jest jako narzędzie *Xcode*, uruchamia się na naszym komputerze jako aplikacja ale działa jak standardowe urządzenie mobilne typu iPhone, iPad, Apple Watch[2]. Umożliwia symulowanie iOS, watchOS czy tvOS w

[2]iPhone – telefon firmy Apple, iPad – tablet firmy Apple oraz Apple Watch – zegarek

wybranej przez wersji systemów. Każda wersja i rodzaj urządzenia na którym symulujemy aplikacje traktowane jest jak oddzielne środowisko z niezależnymi ustawieniami i plikami systemowymi. Dzięki symulacji poprzez te narzędzie jesteśmy w stanie znaleźć błędy w aplikacji jeszcze na etapie konstruowania i wczesnych testów.

Przy tworzeniu aplikacji może pojawić się problem z widokiem czy ustawieniami *auto layout constraints*, który trudno znaleźć poprzez przeglądanie kodu. Zamiast próbować wizualizować sobie układ obiektów *Xcode* dostarcza nam narzędzie pokazujące całą hierarchie widoku widoczną na rysunku 2.6. *Xcode* robi zrzut ekranu i prezentuje w trójwymiarze interfejs użytkownika czyli pozycje,kolejność i wielkość każdego z elementów widoku oraz jak na siebie wzajemnie oddziałują.



Rysunek 2.6: Hierarchia widoku aplikacji

2.3 Języki programowania

Język programowania, informatyczne narzędzie do formułowania programów dla komputerów; jest językiem formalnym, którego składnia określa zasady zapisu programów (w sposób jednoznaczny i łatwy do analizy), a semantyka przypisuje programom ich interpretację (określa efekty działania programu zapisanego w języku programowania).

Składnia języka programowania jest najczęściej definiowana za pomocą gramatyki bezkontekstowej (gramatyka formalna), w której można wygodnie opisać zagnieżdżone struktury (np. instrukcje złożone, wyrażenia arytmetyczne z nawiasami); dodatkowe warunki składniowe, których nie można zdefiniować w ten sposób (np. wymaganie, aby funkcje użyte w programie były w nim zdefiniowane), są najczęściej zapisywane w języku naturalnym. Semantyka jest zwykle opisywana tekstem w języku naturalnym, choć można ją określić w sposób w pełni precyzyjny i formalny, np. podając reguły tłumaczenia konstrukcji języka programowania na ciąg operacji pewnego hipotetycznego komputera (semantyka operacyjna) albo określając sposób konstruowania funkcji przekształcającej dane wejściowe programu w jego wyniki (semantyka denotacyjna)[3].

[3] `http://encyklopedia.pwn.pl/haslo/jezyk-programowania;3917948.html` czas dostępu:2016-04-21

2.3.1 Objective-C

Objective–C jest jednym z dwóch języków programowania używanym do tworzenia oprogramowania na platformy OS X, iOS czy watchOS. Pochodzi on od języka C i dziedziczy takie elementy jak np. składania, typy prymitywne oraz instrukcje sterujące, co więcej kod poprawnie napisany języku C jest kompatybilny i zgodny z językiem *Objective–C*, choć rozszerzony jest o możliwości programowania obiektowego pochodzące od *Smaltalka*. W 1988 firma *NeXT Software* wykupiła licencje do tego języka i wypuściła bibliotekę opartą o język *Objective–C* pod nazwą *NeXTSTEP*. Kiedy firma *Apple* została właścicielem *NeXT*, bibliotekę wbudowano w jądro systemów operacyjnych *Mac OS X*, zapewniając tym sposobem nowoczesny fundament OS, którego *Apple* nie było w stanie zapewnić we własnym zakresie.

W *Objective–C* wymaga się by deklaracja i definicja klasy odbywała się w dwóch odrębnych plikach. Przyjęło się by deklarację umieszczać w pliku nagłówkowym z rozszerzeniem .h i między słowem kluczowym *@interface* oraz zamykającym *@end* widoczne na listingu **??**. Zadeklarowana klasa posiada dwa parametry typu NSObject(klasy bazowej większości klas języka *Objective–C*), przedrostek NS wymienionych typów parametrów pochodzi od biblioteki *NeXTSTEP*, która została wspomniana wcześniej. Od wiersza 6 do 12 widoczne są metody klasy oraz metody instancji, *"+"* oznacza metodę klasy, dzięki niej nie potrzebujemy tworzyć instancji klasy by wywołać tą metodę. Drugi rodzaj metody operuje na stworzonym obiekcie danej klasy.

```
1 @interface NazwaKLasy: klasaNadrzedna {
2   NSObject *parametr1Klasy
3   NSObject *parametr2Klasy
4 }
5
6 + (void)metoda1Klasy;
7 + (typParametruWyjsciowego) metoda2Klasy;
8 + (typParamteruWyjsciowego) metoda3Klasy:(typParametruWejsciowego)
    nazwaParametruWejsciowego;
9
10 - (void)metoda1Instancji;
11 - (typParametruWyjsciowego) metoda2Instancji;
12 - (typParamteruWyjsciowego) metoda3Instancji:(typParametruWejsciowego)
    nazwaParametruWejsciowego;
13
14 @end
```

Listing 2.1: Deklaracje klasy w Objective-C umieszczamy w pliku z rozszerzeniem .h

W sekcji drugiej nazywanej definicją klasy, kod umieszczony w pliku z rozszerzeniem *.m* odpowiedzialny jest za rzeczywiste działanie klasy. Definicje metod umieszczamy między słowem kluczowym *@implementaion* a zamykającym *@end*. W przypadku metody widocznej na listingu**??** by nie wyskoczył błąd przy kompilacji typ parametru wyjściowego musi być taki sam jak w przypadku parametru wejściowego.

```
1 @implementation
2 -(typParametruWyjsciowego) metoda2instancji:(typParamteruWejsciowego)
      nazwaParametruWejsciowego {
3   return ParamertWejsciowy
4 }
5 @end
```

Listing 2.2: Definicje klasy w Objective-C umieszczamy w pliku z rozszerzeniem .m

Zazwyczaj do stworzenie obiektu danej klasy potrzebujemy dwóch metod. Pierwsza z nich widoczna w listingu 2.3 nazywa się *allloc* i odpowiada za przydzielenie odpowiedniej wielkości na obiekt w pamięci naszego komputera, zaś druga *init* jest domyślnym konstruktorem gdy nie nadpiszemy go sami dziedziczony jest z klasy nadrzędnej *NSObject*.

```
1 NaszaKlasa * nazwaKlasy = [[NaszaKlasa alloc] init];
```

Listing 2.3: Instrukcja stworzenia obiektu.

Obiekty klas powinny mieć wyraźnie zdefiniowane zadania, pozwalające na wykonywanie określonych czynności takich jak modelowanie konkretnej informacji, wyświetlenia treści, kontrole przepływu informacji. Interfejs klasy definiuje sposoby dzięki którym poprzez spodziewaną interakcje na obiekcie jesteśmy w stanie osiągnąć te czynności. Niekiedy jednak potrzebujemy rozszerzyć funkcjonalność klasy poprzez dodanie nowej metody czy parametru by bardziej odpowiadała naszym potrzebom. Dla przykładu, możemy potrzebować by nasza aplikacja wyświetlała konkretny ciąg znaków kilkanaście razy w różnych miejscach, zamiast tworzyć obiekty za każdym razem gdy będziemy potrzebować tej funkcjonalności możemy wziąć pod uwagę czy nie dodać jej do klasy *NSString* by to ona mogła to wyświetlić ten napis. Stworzenie podklasy dziedziczącej z *NSString* ogranicza tą funkcjonalność i użycie jej type *NSMutableString* będzie już niemożliwe. Zamiast tego *Objective–C* daje nam możliwość dodanie własnych metod do istniejących klas poprzez kategorie i rozszerzenia klas.

Jeśli potrzebujemy dodać metody do już istniejącej klasy najprostszym sposobem jest dodanie kategorii. Zaczynamy od deklaracji nagłówka wraz z zainicjowaniem metody widocznej w wierszu 5,oraz nazwy kategorii wyszczególnionej w nawiasach obok nazwy klasy widocznej w wierszu 3 listingu2.4.

```
1 #import "Osoba.h"
2
3 @interface Osoba (OsobaWypiszDaneOsobowe)
4
5 -(NSString *) wypiszNazwiskoImie;
6 }
7
8 @end
```

Listing 2.4: Deklaracja nowej metody poprzez kategorie

Następnym krokiem jest zaimplementowanie funkcjonalności naszej metodzie. Przyjęło się by plik kategorii nazywać poprzez połączenie przez "+ńazwę klasy do której

dodajemy kategorie oraz nazwę opisującą kategorie widoczną na listingu 2.5 w wierszu 1. Teraz wystarczy tylko zaimportować tę klasę gdy będziemy potrzebowali skorzystać z nowej funkcjonalności.

```
1 #import "Osoba+OsobaWypiszDaneOsobowe.h"
2
3 @interface Osoba (OsobaWypiszDaneOsobowe)
4
5 -(NSString *) wypiszNazwiskoImie {
6   return [NSString stringWithFormat:@"%@, %@", self.nazwisko, self.imie
      ];
7 }
8
9 @end
```

Listing 2.5: Definicja metody kategorii

Rozszerzenia klasy działają podobnie jak kategorie, z tą różnicą że dodać je możemy tylko do klas których posiadamy kod źródłowy (kompilacja klasy odbywa się w tym samym czasie co rozszerzenie tej klasy). Składania pozwalająca na deklaracje rozszerzenia klasy widoczna na listingu 2.6 jest bardzo podobno do składni kategorii. Ponieważ nie podana jest żadna nazwa w nawiasach w wierszu 1 rozszerzenia nawiązują to kategorii anonimowej. W przeciwieństwie do zwykłych kategorii, rozszerzenie daje nam możliwość dodania *@property* czy zmiennych instancji widocznej w wierszu 3. Kompilator automatycznie syntetyzuje zmienną instancji do implementacji klasy podstawowej.

```
1 @interface Osoba ()
2
3 @property NSString  *adres;
4
5 @end
```

Listing 2.6: Deklaracja nowego parametru poprzez rozszerzenie klasy bazowej

Protokoły [8] to lista metod wspólnie użytkowane przez klasy. Metody znajdujące się na takiej liście nie mają implementacji, ponieważ są przeznaczone do zaimplementowania przez innych programistów. Określany jest jako sposób definicji zestawu metod, które mają jakiś związek z określoną nazwą. Metody te zazwyczaj są dobrze udokumentowane, dzięki czemu wiadomo, jak mają działać i można je w razie potrzeby zaimplementować w swojej klasie. Może zawierać listę metod, których implementacja jest opcjonalna, oraz zestaw takich, które trzeba zaimplementować obowiązkowo. Jeśli zaimplementuje się wszystkie wymagane metody, mówi się, że klasa jest zgodna z danym protokołem lub go adaptuje. Można zdefiniować zarówno protokół zawierający same wymagane, jak i same opcjonalne metody. Składnia definicji protokołu widoczna na listingu 2.7 jest dosyć prosta, należy użyć dyrektywy *@protocol*, po której wpisuje się dowolną nazwę definiowanego protokołu. Następnie deklaruje się metody w taki sam sposób jak w sekcji interfejsu. W skład protokołu wchodzą wszystkie metody zadeklarowane między słowami *@protocol* i *@end*.

```
1 @protocol NSCopying
```

```
2
3 - (id)copyWithZone: (NSZone *)zone;
4
5 @end
```

Listing 2.7: Protokół *NSCopying* zdefiniowany w pliku nagłówkowym *NSObject.h*

Czasami zachodzi potrzeba by parametr naszej klasy raz zainicjowany zachował swoją wartość pomimo iż został wywoływany kilkakrotnie lub ograniczyć widoczność funkcji tylko do pliku gdzie została zdefiniowana. W obu przypadkach słowem kluczowym odpowiedzialnym za te funkcjonalności jest komenda *static*.

```
1 int liczCoTrzy() {
2   static int licznik = 0;
3   licznik += 3;
4   return licznik
5 }
6
7 int main() {
8   liczCoTrzy(); // 3
9   liczCoTrzy(); // 6
10  liczCoTrzy(); // 9
11  return 0;
12 }
```

Zmienna statyczna *licznik*, która została trzy razy zainicjowana zachowała swoją wartość z poprzednich wywołań co pozwoliło osiągnąć upragniony efekt. Domyślnie wszystkie funkcje widoczność globalną, co oznacza że raz zdefiniowana funkcja w jednym pliku jest od razu widoczna we wszystkich innych. Słowo kluczowe *static* pozwala ograniczyć widoczność funkcji do konkretnego pliku, dzięki temu możemy stworzyć funkcje prywatną i uniknąć konfliktów w nazwach.

```
1 //deklaracja funkcji statycznej
2 static int dodajLiczby(int, int);
3
4 //Implemetacja funkcji statycznej
5 static int dodajLiczby(int pierwsza, int druga) {
6   return pierwsza + druga;
7 }
```

Trzeba pamiętać by słowo kluczowe *static* było użyte przy deklaracji i implementacji.

2.3.2 Swift

Swift [10] to opracowany przez *Apple'a* nowy język programowania, który ma być przyjazny i łatwy w użyciu. Został zaprojektowany z uwzględnieniem filozofii łatwości użycia i szybkości działania języków skryptowych, a jednocześnie oferuje potężne możliwości charakterystyczne dla języków kompilowanych. Opracowany przez *Apple'a* język *Swift* został zaprezentowany w roku 2014, od tamtej chwili jest zin-

tegrowany z *Xcode* i pozwala na tworzenie aplikacji na platformy OS X, iOS oraz watchOS.

W chwili pisania tej pracy *Swift* dostępny jest w wersji 2.2 i wraz z nowym wydaniem wprowadzono kilka istotnych zmian poprawiających prace. Zdecydowano się na nowy model obsługi błędów widoczny na listingu 2.8, który ma dostarczyć jasną i przejrzystą składnie obsługującą błędy. Nowy model błędów, który pozwala na stworzenie własnych typów błędu, został zaprojektowany do płynnej współpracy z *NSError* i biblioteką *Cocoa*.

```
1 func wczytajDane() throws {}
2
3 func test() {
4   do {
5     try wczytajDane()
6     } catch {
7       print(error)
8     }
9 }
```

Listing 2.8: Nowy model obsługi błedów w Swift

Dodano nowe funkcje składniowe, które pomagają pisać bardziej przejrzysty kod, poprawiając jednocześnie jego spójność. Zastosowano w *SDK* nowe funkcjonalności z *Objective–C* takie jak typy generyczne czy wartości *null* adnotacji poprawiające bezpieczeństwo kodu. Niektóre z pozostałych ulepszeń to:

- lepsza kontrola przepływu działania przy pomocy *do*,*guard*, *defer* i *repeat*.
- rozszerzenia protokołów i domyślna implementacja.
- rozszerzone dopasowanie do wzorca do pracy klauzulą if i pętlą for.

Wraz z nową wersją pojawiła się informacja, że *Swift* stał się oprogramowaniem *open source*. W tym samym czasie otwarta została strona *Swift.org* z dostępnym kodem, rejestratorem błędów i systematycznymi wersjami developerskimi dla każdego. Na razie *Swift* obsługuje platformy *Apple* oraz *Linux* ale pracują na resztą platform.

Częścią wspólną dla zmiennych i stałych jest nazwa oraz wartość przechowywane określonego typu. Wartość stałej raz zainicjowanej nie może zostać zmieniona, w przeciwieństwie do zmiennych gdzie wartość może zostać zmieniona wielokrotnie. Podstawową zasadą jest by zadeklarować zmienne lub stałe przed próbą ich użycia. Słowem kluczowym za zadeklarowania stałej w języku *Swift* odpowiada *let*, zaś zmiennej *var* widoczne na listingu 2.9. Dobrym zwyczajem jest by właściwie deklarować stałe i zmienne, co nie tylko sprawia, że nasz kod jest bezpieczniejszy ale świadczy o znajomości pisanego przez nas kodu.

```
1 let stalaLiczbaPi = 3.14159
2
3 var zmiennaX = 10
```

Listing 2.9: Deklaracja stałych i zmiennych w Swift

W języku *Swift* bardzo rzadko jesteśmy zmuszeni to definiowania typu po zainicjowaniu wartości dla zmiennej lub stałej. Prawie zawsze *Swift* jest w stanie sam wywnioskować typ po wartości.

Funkcje są samodzielnymi kawałkami, które odpowiadają za wykonanie konkretnego zadania. Nazywamy funkcje zgodnie z jej wykonywanym zadaniem i wywołujemy tą nazwę w miejscu gdzie te konkretne zadanie ma być wykonane. Składnia funkcji w *Swift* jest na tyle elastyczna, aby wyrazić wszystko, od prostych funkcji w stylu języka *C* bez nazw parametrów wejściowych do kompleksowych metod *Objective–C* z lokalnymi i zewnętrznymi nazwami dla każdego parametru. Możemy podać dla parametrów wejściowych wartości domyślne w celu uproszczenia wywołań funkcji lub użyć parametrów *in–out*, które mogą być przekazane poza funkcje ze zmodyfikowanymi wartościami.

```
1 func przywitajSieZ(imieOsoby: String) -> String {
2
3   let powitanie = "Witaj, " + imieOsoby + "!"
4
5   return powitanie
6
7 }
```

Listing 2.10: Definicja funkcji w języku Swift

Funkcja widoczna na listingu 2.10 nosi nazwę *przywitajSieZ(_:)*, ponieważ za to właśnie odpowiada – bierze imię osoby i zwraca powitanie z tą osobą. By to osiągnąć zdefiniowaliśmy jeden parametr wejściowy typu *String* i zwrócimy typ *String* zawierający powitanie wraz z imieniem. Rezultat widoczny jest poniżej.

```
print(przywitajSieZ("Kamil")) // "Witaj, Kamil!"
```

Swift dostarcza trzy główne typy kolekcji, tablice, słownik oraz zbiór do przechowywania elementów. Tablice są uporządkowaną kolekcja danych, zbiór natomiast nieuporządkowana kolekcją unikatowych danych, zaś słownik nieuporządkowaną kolekcją typu klucz–wartość. W żadnej z tych kolekcji nie możemy wstawić wartości o różnych typach, daje nam to pewność, że wartość którą chcemy odzyskać z kolekcji jest określonego typu. W *Objective–C* jeśli zależało nam na strukturze, która raz zainicjowana elementami nie zostanie zmieniona byliśmy zmuszeni do użycia kolekcji o innej nazwie i funkcjonalności od struktury, która pozwalała na taką zmianę. W języku *Swift* słowami kluczowymi oznaczającymi czy kolekcje możemy modyfikować lub nie, odpowiadają tak jak w przypadku zmiennych i stałych, *let* i *var*. Pełna nazwa typu tablicy w *Swift* piszemy *Array<Element>*, gdzie *Element* oznacza typ wartości dozwolony do przechowania w tej tablicy. Możemy także skorzystać ze skróconej wersji pisząc tylko *[Element]*. Obydwie formy działają identycznie, choć wersja skrócona jest preferowana.

```
var tablicaIntow = [Int]()
```

Komenda widoczna powyżej odpowiedzialna jest za stworzenie pustej tablicy przechowującej wartości typu całkowitego. Jak już wcześniej wspomniałem *Swift* jest w stanie wywnioskować samemu jakiego typu wartości będą przechowywane w tablicy, dlatego też inicjalizacja bez podania typu jest poprawna.

var tablicaStringow = ["kot","pies","kaczka",łeniwiec"]

Przy korzystaniu z tablicy *Swift* zapewnia kilkanaście przydatnych funkcji pozwalających na łatwą obsługę. Najczęściej używane funkcje to:

- **array.count()** – zwraca liczbę elementów w tablicy w postaci liczby całkowitej.
- **array.isEmpty()** – sprawdza czy tablica jest pusta, jeśli tak zwraca wartość boolowską *true* w przeciwnym przypadku *false*.
- **array.append(_:)** – dodaje element na koniec tablicy.
- **array.insert(_:atIndex:)** – dodaje element pod wskazany przez nas index.
- **array.removeAtIndex(_:)** – usuwa element ze wskazanego indexu.
- **array.removeLast()** – usuwa ostatni element z tablicy.

Zbiór(ang. *Set*) przechowuje unikatowe wartości tego samego typu w kolekcji w niezdefiniowanej kolejności. Możemy użyć zbioru zamiast tablicy jeśli kolejność jest dla nas bez znaczenia lub gdy potrzebujemy mieć pewność, że element pojawi się tylko raz. Innym wymogiem używania tej struktury jest typ przechowywanych wartości a dokładniej musi posiadać funkcje haszującą, która zwraca wartość *hash*. Wartość *hash* jest liczbą całkowitą i jest jednakowa dla wszystkich elementów, które są sobie równe. Wszystkie podstawowe typy w *Swift*(takie jak *String*, *Int*, *Double* czy *Bool*) posiadają taką funkcje domyślnie i mogą być przechowywane w *Set*. Składnia wygląda podobnie jak w przypadku tablicy, *Set<Element>* gdzie *Element* odpowiada z przechowywany typ. Jednak w przeciwieństwie do tablicy nie posiada skróconej wersji.

var litery = Set<Characters>()

Nawiasy bez argumentów oznaczają wywołanie konstruktora, który stworzy pusty *Set* typu *Character*. Większość funkcji dostępnych dla tablicy możemy użyć na *Set*, jednak posiada on unikatowe warte uwagi funkcje:

- **set.intersect(_:)** – porównuje dwa zbiory i zwraca nowy zawierający elementy wspólne dla obu.
- **set.exclusiveOr(_:)** – nowy zbiór bez elementów wspólnych.
- **set.union(_:)** – połączenie dwóch zbiorów razem z elementami wspólnymi.
- **set.substract(_:)** – pozostawia tylko te elementy, które nie pojawiły się w drugim zbiorze.

Ostatnią kolekcją dostępną w *Swift* jest słownik przechowujący związki pomiędzy kluczami a wartościami tych samych typów kolekcji z niezdefiniowanym porządkiem. Każda wartość jest połączona unikatowym kluczem, który zachowuje się jak identyfikator dla wartości w słowniku. Powinniśmy wykorzystywać słownik kiedy potrzebujemy wyszukiwać wartość po identyfikatorze, podobnie jak w rzeczywistym słowniku wyszukujemy definicje określonego słowa. Pełna deklaracja typu dla słownika wygląda podobnie jak w pozostałych przypadkach lecz posiada dwa argumenty,

Dictionary<Key, Value> gdzie *Key* odpowiada za identyfikator dla wartości. Jak w przypadku tablicy, słownik posiada skróconą wersje zapisu *[Key: Value]*, i tak jak w przypadku tablicy skrócona wersja jest wersją preferowaną.

var kodyPocztoweMiejscowosci = [String: String]()

Definicja tworzy pusty słownik przechowujący jako klucz kod pocztowy dla miasta jako wartości. Przykładowe przypisane do istniejącego już słownika wyglądało by następująco:

kodyPocztoweMiejscowosci = ["18-300": Żambrów", "16-010": Ńowodworce",
"12524": "Berlin", "20355": "Hamburg"]

Do iteracji po słowniku możemy wykorzystać strukturę zwaną *tuple* oraz pętle for widoczną za listing 2.11

```
1 for (kodPoczotwy, nazwaMiejscowosci) in kodyPocztoweMiejscowosci {
2     print(kodPocztowy + " " +  nazwaMiejscowosci)
3
4 }
```

Listing 2.11: Iteracja po słowniku z wykorzystaniem *tuple* i pętli for.

Rozdział 3

Implementacja aplikacji

3.1 Opis aplikacji MobiUwB

3.1.1 Zastosowania

3.1.2 Funkcje

3.1.3 Elementy

3.2 Architektura programu

3.3 Użyte biblioteki

3.4 Schematy Konfiguracji

3.4.1 Schematy XML

3.4.2 Schematy JSON

Bibliografia

[1] Robin Berjon, Steve Faulkner, Travis Leithead, Silvia Pfeiffer, Edward O'Connor, and Erika Doyle Navara. HTML5. Candidate recommendation, W3C, July 2014. http://www.w3.org/TR/2014/CR-html5-20140731/.

[2] T. Bray. The javascript object notation (json) data interchange format. Technical report, 2014. https://tools.ietf.org/html/rfc7159.

[3] Tim Bray. Extensible markup language (xml) 1.0 (fifth edition). W3c recommendation, W3C, 2008. http://www.w3.org/TR/2008/REC-xml-20081126/.

[4] Elizabeth Castro. *Po prostu HTML, XHTML i CSS*. Helion, vi edition, 2008.

[5] Elika Etemad. Cascading style sheets (CSS) snapshot 2010. W3C note, W3C, May 2011. http://www.w3.org/TR/2011/NOTE-css-2010-20110512/.

[6] Brian P. Hogan. *HTML5 i CSS3: Standardy przyszłości*. Helion, i edition, 2011.

[7] Steven Holzner. *Ajax. Biblia*. Helion, i edition, 2008.

[8] Stephen G. Kochan. *Objective-C Vademecum profesjonalisty*. Helion, iii edition, 2012.

[9] R. Sayre M. Nottingham. The atom syndication format. Technical report, 2005. https://tools.ietf.org/html/rfc4287.

[10] Tom Stachowitz Mark A. Lassoff. *Podstawy języka Swift*. Helion, iii edition, 2016.

[11] David Sawyer McFarland. *JavaScript i jQuery: nieoficjalny podręcznik*. Helion, i edition, 2012.

[12] Dori Smith Tom Negrino. *Po prostu JavaScript i Ajax*. Helion, vi edition, 2007.