Uniwersytet Jagielloński w Krakowie
Wydział Fizyki, Astronomii i Informatyki Stosowanej

WITOLD KAROL BOBROWSKI

NUMER ALBUMU: 1115454

CZYTNIK KSIĄŻEK ELEKTRONICZNYCH NA iOS

PRACA LICENCJACKA NA KIERUNKU INFORMATYKA

PRACA WYKONANA POD KIERUNKIEM:
dra Karola Przystalskiego
Zakład Technologii Informatycznych

Kraków 2017

Oświadczenie autora 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.

Kraków, dnia Podpis autora pracy

Oświadczenie kierującego pracą

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

Kraków, dnia Podpis kierującego pracą

podziękowania

i więcej podziękowań

Streszczenie

streszczenie

Abstract

abstract

Spis treści

1. Wstęp

Celem pracy jest opisanie stworzonego przeze mnie projektu jakim jest biblioteka **EPUBKit** której zadaniem jest obsługa (parsowanie oraz wyświetlanie) książek elektronicznych w formacie EPUB (Electronic Publication) a następnie wykorzystanie jej w aplikacji mobilnej na platformie iOS. Biblioteka ta została stworzona z myślą jej udostępnienia publicznie w formie open-source na portalu GitHub. EPUBKit jest lekkim narzędziem w języku Swift, które uprości pracę innym programistom tworzącym aplikacje na iOS rozwiązując problem jakim jest obsługa formatu EPUB. Na dzień dzisiejszy natywne biblioteki iOS nie zapewniają programistom takiego narzędzia, a publicznie istnieje niewiele rozwiązań, które cieszą się mniejszą lub większą popularnością. Problem parsowania oraz wyświetlenia publikacji w formacie EPUB nie jest trywialny. Parser musi sobie poradzić ze skomplikowaną strukturą i bardzo bogatą zawartością takiego dokumentu, który choć posiada szczegółową specyfikację, to ze względu na urozmaicenie technologii w nim wykorzystanych, czyni pracę parsera znacznie trudniejszą. EPUBKit pokonuje te trudności, analizując zawartość i kolektywizując informację dzięki czemu finalnie tworzy instancję klasy która reprezentuje dokument EPUB. Biblioteka dodatkowo dostarcza widok którzy można wykorzystać w celu podglądu dokumentu. Cały projekt jest stworzony tak, aby jak najbardziej wpasowywał się w konwencje programowania w Swifcie na platformę iOS i opiera się na wzorcu architektonicznym Model-Widok-Kontroler, który jest szeroko stosowanym przy programowaniu aplikacji na iOS (zgodnie z wytycznymi Apple). Biblioteka wykorzystuje w pełni język Swift i wszelkie jego nowinki, które sprawiają że jest językiem bardzo praktycznym a przy tym lekkim, szybkim i przyjaznym. Biblioteka EPUBKit choć nie idealna, a napewno nie będąca w swojej finalnej wersji, to już w aktualnej formie oferowane przez nią API (Application Programming Interface) spełnia założone wymagania. Ze względu na wciąż ewoluującą oraz szeroką specyfikację formatu EPUB, dostarczenie takiej biblioteki która gwarantowała by taką funkcjonalność oraz była by tak rozwinięta jak dla przykładu program iBook na iOS wymagało by ogromnego nakłady pracy ze strony całej drużyny programistów. Głęboko wierzę w słuszność ruchu wolnego oprogramowania, dlatego też postępuję zgodnie z jego duchem i decyduję się upublicznić moją pracę, która już w obecnej formie może zaoszczędzić wiele czasu komuś kto natknie się na problem obsługi dokumentu EPUB, a przy tym może zostać przez tą osobę ulepszona i rozwinięta.

Mobilny system od Apple, iOS niedługo będzie obchodził 10 lat od wprowadzenia go na rynek. W 2014 roku Apple ogłosiło, że jest ponad miliard aktywnych urządzeń z tym właśnie systemem a dziś jest ich z pewnością znacznie więcej. Każdy kolejny model telefonu komórkowego marki Apple, prezentowany z roczną częstotliwością cieszy się coraz większym powodzeniem. Oprócz nowych urządzeń dostajemy w pakiecie nową wersję systemu iOS która jest nie tylko udoskonaleniem poprzedniej wer-

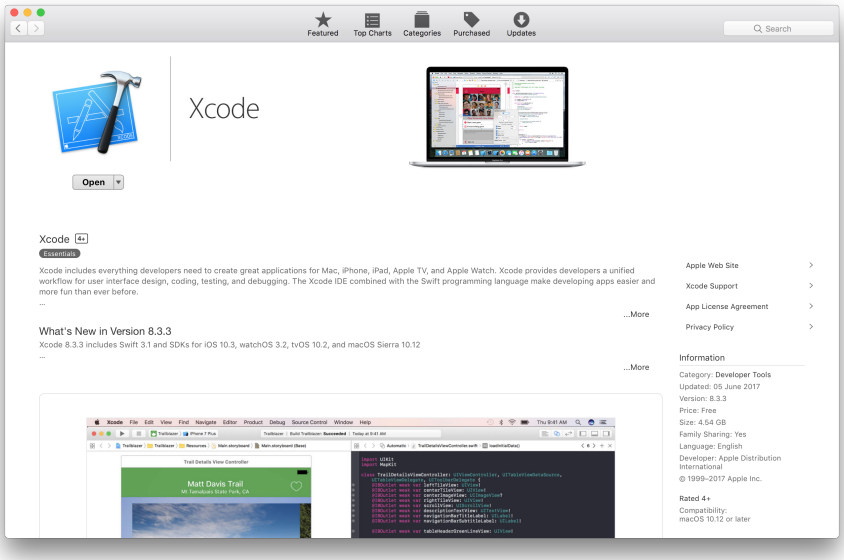
sji, ale również jej pełnoprawnym następcą wprowadzając nowy zbiór zarówno funkcjonalności jak i elementów wizualnych. Tak dynamicznie rozwijający się system jest bardzo atrakcyjny dla użytkownika, który dzięki darmowym aktualizacjom dla starszych urządzeń wciąż może cieszyć się najnowszym oprogramowaniem. iOS Development staje się coraz popularniejszy wśród programistów oraz przyciąga wielu młodych ludzi zainteresowanych technologią. Język Swift nad którym prace zostały rozpoczęte w 2010 roku przez Chrisa Lattnera oraz wielu innych programistów z Apple, a w 2014 roku miał swój debiut, dzisiaj jest już głównym językiem programowania mobilnych aplikacji na platformę iOS oraz aplikacji desktopowych na MacOS i wyparł dotychczas używany w tych celach język Objective-C, który swoją historię ma również ściśle związaną z Apple. Korzenie fundamentalnych frameworków z iOS, takich jak CocoaTouch, sięgają lat 80-tych poprzedniego stulecia, a przez ten czas były bardzo silnie rozwijane i wykorzystywane w desktopowym systemie MacOS. Nowoczesny język oraz potężne SDK (Software Development Kit) stanowią dziś podstawę pracy z aplikacjami na te platformy. Z roku na rok, wraz z nową wersją systemu, Apple uaktualnia istniejące API oraz dostarcza nowe biblioteki zapewniające dostęp do najnowszych elementów systemu.

Ta praca dokumentuje bibliotekę "EPUBKit", rozpoczynając od dokładnego opisu środowiska, wykorzystanych narzędzi, scharakteryzowano format EPUB i jego specyfikację techniczną, opisano proces tworzenia biblioteki, jej strukturę oraz możliwości dystrybucji biblioteki jako moduł gotowy do wykorzystania przez developerów. Następnie w celu demonstracji funkcjonalności biblioteki opisano proces tworzenia aplikacji z jej wykorzystaniem.

2. Wykorzystane technologie

2.1. Xcode i Developer Tools

Xcode jest IDE (Integrated development environment) stworzonym przez Apple i dostępnym za darmo do pobrania z App Store, sklepu z aplikacjami do którego dostęp mają wyłącznie użytkownicy komputerów z systemem MacOS. Jest wyposażony w pakiet wszystkich narzędzi (Developer Tools) potrzebnych dla developerów aby tworzyć aplikacje na iOS. Główną aplikacją pakietu jest Xcode IDE który wraz z wspomagającymi aplikacjami dostępnymi w pakiecie takimi jak Simulator czy Instruments czyni pracę przy tworzeniu aplikacji płynną i efektowną. W tym rozdziale przedstawię właśnie te narzędzia ze względu na ich rolę w procesie tworzenia aplikacji.

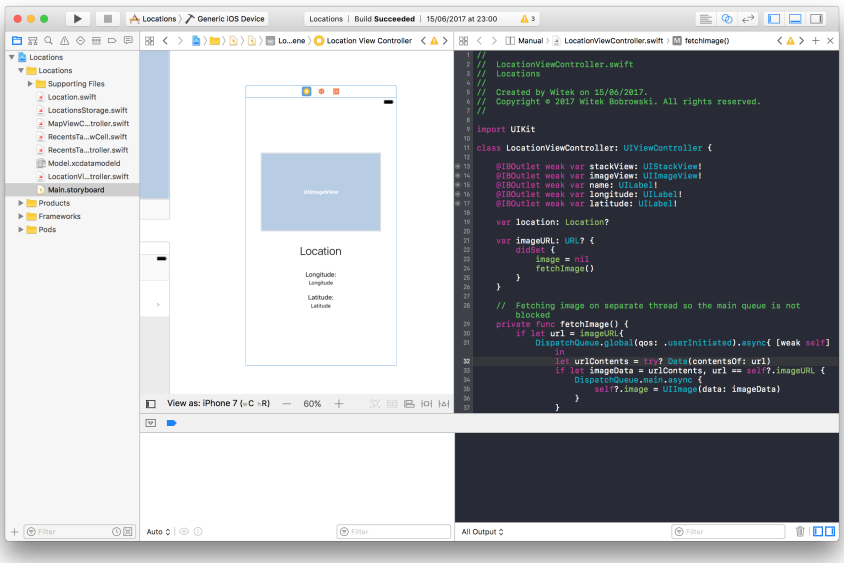


Rysunek 2.1: Xcode w App Store

2.1.1. Xcode IDE

Xcode jako nowoczesne, produktywne środowisko jest miejscem w którym programista aplikacji na iOS spędza większość swojego czasu. Całość prac wykonywanych przy produkcji aplikacji może zostać wykonana właśnie tutaj. Najbardziej podstawowy element jakim jest edytor tekstu dobrze współgra z takimi narzędziami jak Interface Buildier, który pozwala w prosty sposób zaprojektować stronę wizualną

aplikacji przy użyciu Storyboardów a następnie stworzyć referencję w kodzie do wybranych przez nas elementów przez proste przeciągnięcie myszką. Storyboardy są opcjonalnym aczkolwiek bardzo pożytecznym narzędziem szczególnie dla programistów stawiających swoje pierwsze kroki na tej platformie. Zapewniają one wizualne wyobrażenie interfejsu aplikacji nad którą wykonywana jest praca, a projektowanie dowolnego widoku który będzie wyglądał dobrze na każdym urządzeniu w dowolnej orientacji, jest relatywnie proste po zapoznaniu się z kilkoma elementarnymi zasadami.

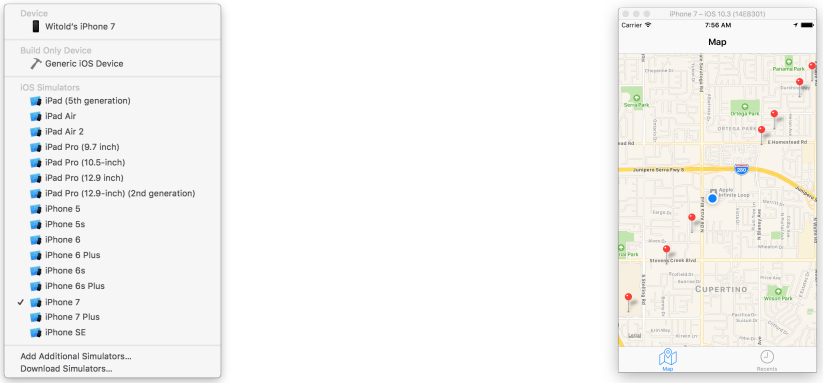


Rysunek 2.2: Xcode pokazujący "Assistant editor"

Ponieważ Storyboardy tworzy się w jednym pliku o formacie .storyboard, często w profesjonalnej produkcji rezygnuje się z nich ze względu na konflikty w systemach kontrolii wersji. Konflikty te powstają w wyniku pracy wielu programistów, a ponieważ plik .storyboard jest w rzeczywistości plikiem XML, który został wygenerowany automatycznie, rozwiązywanie konfilków bywa kłopotliwe, a przy dużych projektach problematyczne. Dlatego rezygnuje się z nich na rzecz tworzenia widoków tylko przy użyciu kodu, oraz niezależnych plików XIB. Pliki te pozwalają na ustawienie elementów w stylu znanym ze Storyboardów lecz w przeciwieństwie do nich reprezentują pojedynczy widok, dzięki czemu problem z konfliktami zostaje uniknięty a jednocześnie tworzenie bardziej skomplikowanych widoków pozostaje znacznie ułatwione. Xcode zapewnia wsparcie dla systemu kontroli wersji git. Przy tworzeniu nowego projektu, gdy jest o to poproszony, inicjalizuje nowe repozytorium. Dodatkowo w nawigatorze projektu, w którym widać strukturę projektu, Xcode oznaczy literą "M" pliki które git oznacza jako pliki w których dokonano zmian (modified) a literą "A" pliki które zostały dodane (new file) od czasu poprzedniego zachowania zmian. W najnowszej wersji 9.0, Xcode zyskał nową funkcjonalność - Source Control Navigator, który pozwala na eksplorowanie poszczególnych gałęzi repozytorium i podglądu dowolnego momentu w jego historii.

2.1.2. Simulator

Simulator pozwala na uruchomienie zbudowanej aplikacji na dowolnym urządzeniu z iOS które jest w stanie zasymulować. Gdy chce się przetestować aplikację wystarczy w pasku narzędzi Xcode wybrać dowolny model urządzenia (oprócz telefonów iPhone znajdują się tam również tablety iPad) które chcemy zasymulować (jeżeli podłączymy do komputera fizyczne urządzenie, Xcode również je wykryje i pozwoli na zainstalowanie i uruchomienie na nim aplikacji), a Xcode przejdzie to etapu budowania aplikacji w którym kompiluje pliki źródłowe, a następnie umieści aplikację w symulatorze wybranego przez nas urządzenia.



(a) Wybór docelowego urządzenia

(b) Uruchomiona aplikacja na iPhone 7

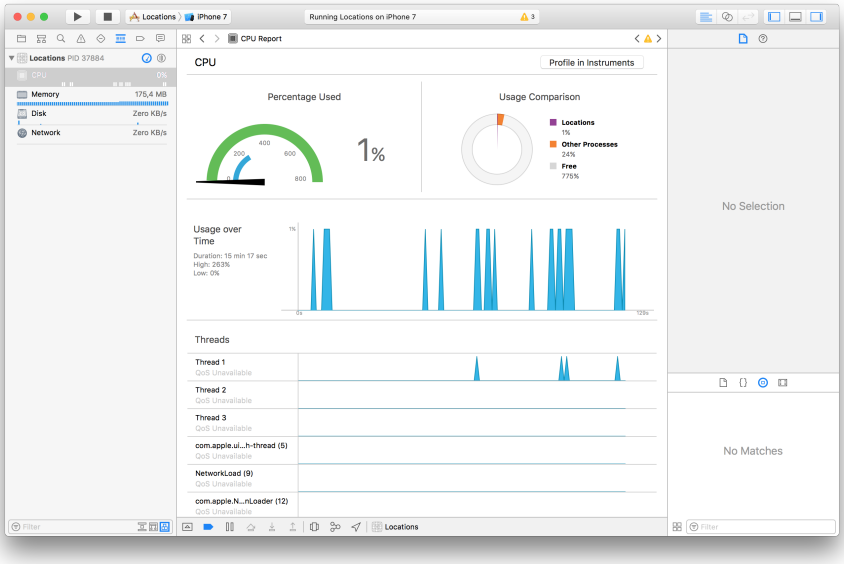
Rysunek 2.3: Po wybraniu urządzenia w Xcode, Simulator uruchamia na nim aplikację

Podczas gdy aplikacja jest uruchomiona i testowana na symulatorze, Xcode pozwala na podgląd użycia zasobów takich jak procesor, pamięć RAM, pamięć dyskowa oraz sieć. Po wybraniu dowolnego z wyżej wymienionych zasobów z nawigatora Debuggera ukazuje nam się bardziej szczegółowy podgląd na to w jakim stopniu aplikacja obciąża urządzenie co pozwala na szczegółowe testowanie.

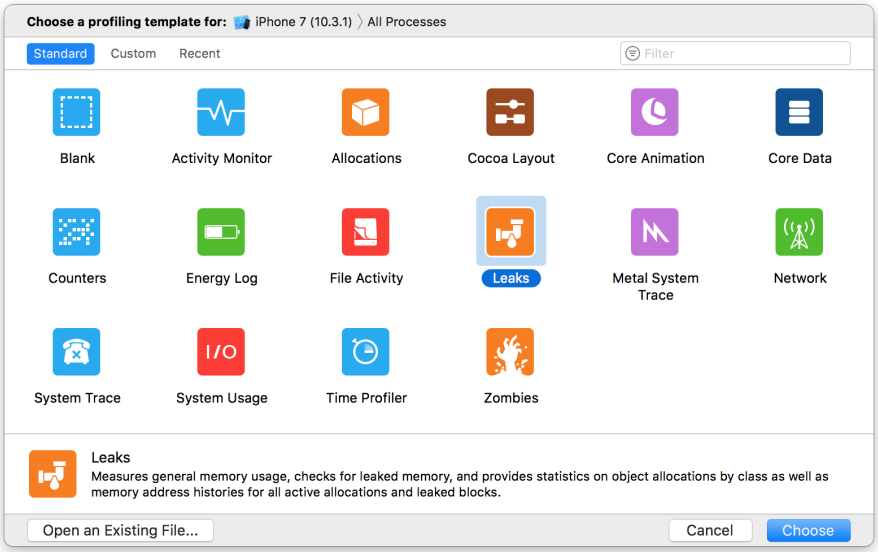
2.1.3. Instruments

Widok poboru zasobów w Xcode jest bardzo pomocny na poglądową ocenę wydajności naszej aplikacji, jeżeli jednak chcemy poddać ją prawdziwej próbie, musimy uruchomić kolejne narzędzie jakim jest Instruments. Instruments jest aplikacją dzięki której dokonamy pomiaru nie tylko każdego zasobu na urządzeniu, ale również dostajemy możliwość nadzoru takich aktywności jak alokowanie pamięci dla obiektów, zmiany layoutu widoków czy zmiany w Core Data, natywnej bazie danych dla iOS.

Niezależnie od tego jak szybkie i wydajne są w dzisiejszych czasach telefony, optymalny kod nadal jest podstawą prawidłowego działania aplikacji i niedbałe projektowanie architektury może być fatalne w skutkach. Cykle referencji mogą powodować niechciane wycieki pamięci a problemy powstające podczas przerysowywania się widoków spowodują nieczytelny interfejs. Jeżeli takie błędy nie zostaną wykryte na etapie produkcyjnym, a dogłębne testy aplikacji nie zostaną przeprowadzone przed oddaniem jej do recenzji (Aby nasza aplikacja mogła znaleźć się w sklepie AppStore, i być dostępna dla każdego, musi ona otrzymać pozytywną recenzje Apple) aplikacja najprawdopodobniej zostanie odrzucona, co



Rysunek 2.4: Użycie CPU na symulatorze odświeżane na bieżąco i wyświetlone w Xcode



Rysunek 2.5: Instruments, menu główne

wiąże się z dodatkowymi opóźnieniami. Pakiet narzędzi dostarczanych przez Apple spełnia swoje zadanie i dla większości developerów są one wystarczające. Istnieją rozwiązania firm trzecich, JetBrains dostarcza alternatywne IDE do produkcji aplikacji - AppCode, które cieszy się bardzo dobrą reputacją wśród użytkowników.

2.2. Swift

Hello, world!

Gdy w 1996 roku Apple przejęło NeXT wraz z ich oprogramowaniem, Objective-C stało się głównym językiem do tworzenia oprogramowania na najnowszy system operacyjny OS X. Wraz z OpenStepem Apple znalazło się w posiadaniu takich narzędzi jak Interface Buildier oraz Project Buildier które z biegiem czasu ewoluowały w Xcode który znamy dzisiaj. Objective-C służył programistom przez wiele lat, na początku tym tworzącym aplikacje desktopowe, a od ukazania się pierwszego iPhone'a, programistom mobilnym. Objective-C jest bardzo mocno rozwiniętym językiem, co zważając na jego wiek nie powinno nikogo dziwić. Lecz jako język którego początki sięgają wczesnym latom osiemdziesiątym dwudziestego wieku, z biegiem lat postarzał się, co bardzo upraszczając można by przedstawić stwierdzeniem, iż nie jest wystarczająco nowoczesny.

Twórca LLVM (Low Level Virtual Machine) Chris Lattner, w 2010 roku rozpoczął prace nad nowym językiem który z założenia miał czerpać inspirację z takich języków jak Objective-C, Haskell, Python czy Ruby. Cztery lata później, Apple na WWDC (Worldwide Developer Conference) zaprezentowało Swifta i wypuściło wersję beta. Swift jest bezpiecznym, szybkim językiem, który agreguje wiele paradygmatów znanych z innych nowoczesnych języków. Jego najważniejszymi cechami są automatycznie zarządzana pamięć, typ Opcjonalny który zapewnia wsparcie dla nullowych wskaźników, zaawansowana obsługa błędów która zapewnia kontrolowane wyjście z niespodziewanych niepowodzeń, zmienne, które zawsze są zainicjalizowane przed użyciem oraz silne typowanie które po połączeniu z nowoczesnym, lekkim syntaxem czyni go językiem zarówno potężnym jak i łatwo dostępnym. Dynamiczny rozwój zawdzięcza inżynierom z Apple oraz społeczności open-source, którzy bardzo intensywnie pracują dodając nowe elementy, naprawiając niedoskonałości oraz optymalizując jego architekturę. Swift dzisiaj jest jednym z najbardziej lubianych języków programowania. Jego popularność wciąż rośnie, a odkąd stał się projektem open-source'owym oprócz systemów Apple, zaczyna być używany do takich celów jak tworzenie serwerowych aplikacji. Zdominował platformę iOS i zdetronizował Objective-C, stając się językiem pierwszego wyboru. Wszystkie powstające aplikacje są tworzone przy jego użyciu, i wiele napisanych w Objective-C jest dzisiaj przepisywanych na nowy język. Warto wspomnieć, że kod w Objective-C i kod w Swifcie może być użyty w tej samej aplikacji, dzięki czemu może się tam znaleźć również C i C++. Jednak aby użyć kodu napisanego w C czy w C++ w Swifcie musimy go najpierw opakować aby był dla niego dostępny. W przypadku Objective-C, Swift ma dostęp do jego obiektów oraz może dziedziczyć z jego klas. A to zapewnia nam dostęp do wszystkich natywnych bibliotek, czyniąc Swift pełnoprawnym językiem i narzędziem gotowym do użytku już w dniu jego premiery.

Do projektu wykorzystano Swift w wersji 3.1.

2.3. iOS SDK

Jak już wspomniano wcześniej, iOS dziedziczy wiele po desktopowym systemie, MacOS. W skład iOS SDK wchodzą biblioteki znane już od wielu lat oraz biblioteki stworzone z myślą o urządzeniach

mobilnych. W tej sekcji omówiono po krótce framework CocoaTouch, który dostarcza elementy interfejsu będące podstawą każdej aplikacji i zapewniające wizualną spójność platformy.

2.3.1. CocoaTouch

CocoaTouch jest potomkiem frameworku Cocoa dostępnego na systemach MacOS który jest rozszerzony o interfejs obsługi narzędzi dostępnych w urządzeniu mobilnym takich jak rozpoznawanie gestów, serwis lokalizacji czy obsługa kamery. W skład CocoaTouch wchodzą między innymi biblioteki Foundation, UIKit, MapKit, EventKit i wiele innych. Dzięki temu pakietowi Apple zdefiniowało jak powinny być tworzone aplikacje na iOS. Dostarczony jest zbiór wielu elementów interfejsu użytkownika które można dowolnie rozszerzać i modyfikować aby stworzyć unikalny wygląd aplikacji trzymając się wytycznych wyznaczonych przez Apple. Dostęp do gestów zapewni naszej aplikacji lekkość obsługi oraz intuicyjność, a niezliczona ilość innych bibliotek wchodzących w skład CocoaTouch sprawi że aplikacja nabierze życia. Implementowanie funkcjonalności staje się bardzo proste dzięki wysoko poziomowym interfejsom dającym dostęp do poszczególnych elementów systemu oraz fizycznego urządzenia. CocoaTouch jest najbardziej elementarnym frameworkiem na iOS ponieważ to on zapewnia na poziomie podstawowym to co potrzebne do stworzenia funkcjonalnej aplikacji. Zakładając że aplikacja powinna agregować, w jakiś sposób przetwarzać a następnie wyświetlić w odpowiedniej formie zbiór pewnych danych, CocoaTouch w parze ze Swiftem to zagwarantują, oczywiście do pewnego stopnia. W większości przypadków gdy potrzebujemy wykonać jakieś specyficzne zadanie, lub chcemy je po prostu uprościć unikając implementacji własnego rozwiązania problemu co było by czasochłonne, będziemy skłaniać się ku zewnętrznym bilbiotekom.

2.4. Zewnętrzne biblioteki

Biblioteki te, tworzone przez środowiska open-sourc'owe, pojedynczych programistów czy wielkie korporacje dadzą nam to czego nie otrzymaliśmy wraz z natywnymi frameworkami. Przykładem jest projekt który opisuje ta praca. Rozwiązuje problem programisty tworzącego własną aplikację który nie chce tracić czasu na implementowanie funkcjonalności która zajęła by mu relatywnie dużo czasu, a która niekoniecznie jest głównym celem jego aplikacji. Istnieje wiele sposobów na rozszerzenie naszej aplikacji o wybrane moduły, które bardziej szczegółowo zostaną opisane pod koniec czwartego rozdziału podczas omawiania możliwościach dystrybucji frameworku. W tej sekcji przedstawiono framework'i wykorzystane do projektu EPUBKit, które nie wchodzą w skład iOS SDK a zostały udostępnione do publicznego użytku.

2.4.1. Zip

Zip jest swift'ową biblioteką open-source dostępną na licencji MIT której kod źródłowy znajdziemy na GitHubie[1]. Zip jest narzędziem do archiwizowania plików oraz rozpakowywania archiwów. Wspiera on formaty archiwów .zip, .cbz oraz daje możliwość dodania rozszerzenia pliku który chcemy rozpa-

[1] Adres url: github.com/marmelroy/Zip

kować. W przypadku tego projektu dodano format .epub, aby rozpakować książkę elektroniczną w tym właśnie formacie. Zip bazuje na narzędziu minizip napisanym w języku C, również na wolnej licencji, oraz dostępnym na portalu GitHub. Dzięki bibliotece zip, parser frameworku EPUBKit może w prostu sposób otrzymać dostęp do struktury plików publikacji elektronicznej. Szczegółowe opisanie wykorzystania frameworku Zip znajduje się w czwartym rozdziale podczas omawiania kodu źródłowego parseru.

2.4.2. AEXML

Kolejną wykorzystaną biblioteką jest AEXML, prosty i lekki pareser XML, dostępny publicznie na licencji MIT, również udostępniony na GitHubie[2]. Ze względu na strukturę elektronicznej publikacji EPUB, parser xml jest niezbędny do rozpoznania zawartości całej struktury dokumentu, identyfikacji elementów oraz określenia ich lokalizacji. AEXML jest swiftowym frameworkiem a jego API jest czytelne i proste w wykorzystaniu dzięki czemu idealnie spełnia swoje zadanie w projekcie który opisuje ta praca.

[2] Adres url: github.com/tadija/AEXML

3. Charakterystyka EPUB

3.1. Omówienie

EPUB jest standardem formatu dystrybucji cyfrowych publikacji i dokumentów opartych na standardach technologii webowej. EPUB definiuje formę reprezentacji, organizacji struktury oraz kodowania określonej zawartości webowej, na co składają się XHTML, CSS, SVG, obrazy i inne zasoby sprowadzone do formy pojedynczego pliku. EPUB daje wydawcom możliwość stworzenia cyfrowej publikacji a następnie dystrybuowania go, a odbiorcy łatwy dostęp do pliku niezależnie od urządzenia jakim operuje. Jako następca OEB (Open eBook Publication Structure), zaprezentowanego w 1999 roku, EPUB 2 został ustandaryzowany w roku 2007 a aktualną jego wersją jest EPUB 3.1 (styczeń 2017). Dzisiaj jest on standardem wykorzystywanym na szeroką skale przez wszystkich wydawców. Obok MOBI oraz PDF dominuje rynek, dzięki jego popularności wsród wydawców oraz wsparciu urządzeń. W przeciwieństwie do MOBI które zostało spopularyzowane przez Amazon, właściciela sklepu amazon.com, giganta dystrybucji książek elektronicznych oraz producenta czytników elektronicznych marki Kindle, EPUB jest standardem uniwersalnym, nieograniczonym do jednej platformy. EPUB zarówna jak i MOBI charakteryzuje się tym, że jego zawartość nie jest statyczna, co oznacza, że ilość która jest wyświetlana dopasowana jest do wielkości ekranu urządzenia dzięki czemu jest ona bardziej przyjazna dla odbiorcy (PDF natomiast jest już podzielony na strony których nie da się podzielić). To co najbardziej różni EPUB od MOBI, to wsparcie EPUB dla multimediów (od wersji 3.0) oraz CSS, który stylizuje cały dokument. Dzięki temu jest znacznie bardziej elastyczny i nowoczesny. Popularną praktyką wśród dystrybutorów książek elektronicznych, jest dostarczanie książki klientowi który ją zakupił we wszystkich trzech wcześniej wymienionych formatach. EPUB jako standard jest szeroko udokumentowany dzięki International Digital Publishing Forum[1], grupie która nadzoruje rozwój formatu. W następnej sekcji zostanie szczegółowo opisana struktura formatu EPUB.

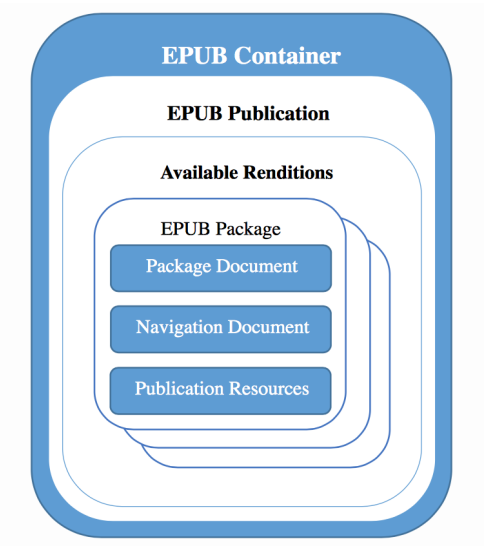
3.2. Specyfikacja

Poniższa specyfikacja jest podzbiorem najważniejszych informacji wyselekcjonowanych ze specyfikacji EPUB 3.1 z dnia 5 stycznia 2017 roku dostępnej na stronie International Digital Publishing Forum[2]. Przedstawiono tutaj najbardziej istotne elementy formatu EPUB w celu zrozumienia problemu jakiego dotyczy projekt EPUBKit.

[1] Adres url: idpf.org

[2] Adres url najnowszej wersji specyfikacji: http://www.idpf.org/epub3/latest

3.2.1. EPUB Open Container Format



Rysunek 3.1: Wizualna reprezentacja struktury formatu EPUB[2].

Format EPUB definiuje jego najwyższy abstrakcyjny model jakim jest EPUB Publication. Model ten składa się z interpretacji jego zawartości. Interpretacja która jest przedstawiona za pomocą EPUB Package zawiera już bezpośrednio zawartość dokumentu, oraz poboczne zasoby mające na celu wspomagać system czytający (z ang. ze specyfikacji "Reading System", jakim jest EPUBKit). Kluczowym elementem jest Package Document który zawiera wszystkie metadane które są używane następnie przez system czytające do zaprezentowania publikacji użytkownikowi. Zawiera on również kompletny manifest zasobów publikacji oraz "kręgosłup" (z ang. ze specyfikacji "Spine"), który reprezentuje sekwencję w jakiej system czytający ma wyświetlać poszczególne elementy. EPUB Package zawiera również Navigation Document pełniący rolę spisu treści, przeznaczony dla użytkownika do poruszania się po dokumencie. Wszystko to opakowane jest archiwum ZIP z rozszerzeniem ".epub". Rozszerzenie informuje o charakterze pliku, oraz dostarcza informację o archiwum w ZIPowskim stylu za pomocą pliku "mimetype", oraz zapewnia system o posiadaniu przez niego folderu "/META-INF" w którym dostępny jest plik container.xml, niezbędny systemowi do określenia lokalizacji zawartości publikacji.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <container version="1.0" xmlns="urn:oasis:names:tc:opendocument:xmlns:container">
3     <rootfiles>
4         <rootfile full-path="OEBPS/content.opf" media-type="application/oebps-
              package+xml"/>
5    </rootfiles>
6 </container
```

Listing 3.1: Przykładowy plik container.xml

To w jaki sposób zawartość publikacji jest zorganizowana określa standard EPUB Open Container Format (OPF), który definiuje reguły enkapsulacji zasobów w pojedynczym kontenerze abstrakcyjnym (EPUB Container) zawartym w archiwum ZIP. Struktura OPF to tylko jedna część składająca się na

EPUB Publication, druga część to zawartość przedstawiona użytkownikowi która jest oparta o XHTML oraz (EPUB Content Documents). Zawartość ta jest rozszerzona o wiele dodatkowych zasobów potrzebnych do prawidłowego wyświetlenia publikacji jakimi mogą być obrazy, pliki audio lub video, dodatkowe czcionki, skrypty oraz style nazywane w oficjalnej specyfikacji "EPUB Core Media Types".

3.2.2. EPUB Content Documents

Ta sekcja opisuje rolę jaką odgrywają standardów HTML, SVG i CSS w elektronicznej publikacji w formacie EPUB. Wizualna kompozycja publikacji w znacznej mierze oparta jest o pliki XHTML. Specyfikacja EPUB Content Documents 3.1, szczegółowo opisuje semantykę atrybutów wspieranych przez EPUB, które mają na celu wzbogacenie doświadczenia użytkownika. Atrybuty te nadają dodatkową naturę i znaczenie elementom XHTML, przy tym nie nadpisując ich pierwotnej funkcjonalności. Atrybuty te są przeznaczone wyłącznie dla systemów czytających i przekazują istotne informacje odnośnie struktury i zawartości dokumentu. Wsparcie standardu EPUB dla HTML nie odnosi się do konkretnej jego wersji, natomiast pozostawia tę kwestię twórcom systemów czytających. To w ich obowiązkach leży upewnienie się, że każda publikacja zostanie prawidłowo przez system obsłużona.

```
1 <html ... xmlns:epub="http://www.idpf.org/2007/ops">
2    ...
3    <p> ... <span epub:type="pagebreak" title="234" id="p234"/> ... </p>
4    ...
5 </html>
```

Listing 3.2: Przykładowe wykorzystanie atrybutu epub:type aby oznaczyć zakończenie linii.[4]

Kolejnym typem dokumentu ważnym dla formatu EPUB jest SVG (Scalable Vector Graphics). Co prawda uniwersalność XHTML przyczynia się do tego iż jest to w znacznej mierze dominujący format przedstawiania treści w publikacji, SVG oferuje udogodnienia dzięki czemu również znajduje zastosowanie. Format ten jest zazwyczaj wykorzystywany w specyficznych przypadkach, takich jak prezentacja zawartości publikacji z gatunku manga czy komiks. Użytkownik oczekiwał by od tego typu dokumentu aby zawartość (w tym przypadku głównie graficzna) prezentowała się dobrze na każdym urządzeniu, niezależnie od rozdzielczości jego ekranu, i to właśnie SVG może zagwarantować. Dodatkowo, treść tych dokumentów jest zazwyczaj z góry podzielona na strony, i narzuca systemom czytającym wyświetlenie ich w określonej formie.

CSS odgrywa ogromną rolę w praktyce programistycznej (jako technologia webowa) od wielu lat, i jego możliwości wciąż rosną. Nietrudno się domyślić że swoje miejsce CSS znajduje również w standardzie EPUB. Prezentacja publikacji jest sztywnie ustylizowana, chociaż nadpisywanie stylu przez systemy czytające nie jest zabronione. Specyfikacja EPUB sugeruje, aby systemy dawały możliwość zmian w stylu publikacji użytkownikowi, który dopasuje jej wygląd pod swoje upodobania. Zazwyczaj będzie to kolor tła, rozmiar czy typ czcionki. Specyfikacja również przestrzega twórców publikacji w formacie EPUB aby rozważnie stylizować dokument. Niektóre systemy czytające mogą nie wspierać pewnych elementów CSS co może być problematyczne[5].

To o czym również należy wspomnieć opisując EPUB Content Documents jest to, że EPUB wspiera skrypty w dokumentach HTML czy SVG. Opcjonalne jest wspieranie skryptowania w systemach czytających, natomiast te systemy które wspierają skryptowane dokumenty, muszą wziąć pod uwagę pewne niebezpieczeństwa z tym związane. Specyfikacja przestrzega, aby systemy czytające zapewniały izolację publikacji w celu uniknięcia niebezpieczeństw zagrażającym podatnej na ataki zawartości. System powinien monitorować wszelkie aktywności aby zawartość pozostała nienaruszona. Należy założyć, że może zostać przeprowadzony atak na zawartość innych plików w strukturze publikacji, atak na sam system czytający (np. próba zdobycia danych użytkownika), atak na sieć czy wszczepianie innych złośliwych skryptów w niezaszyfrowane fragmenty dokumentu. Ponadto systemy które zezwalają na stałe przechowywanie publikacji, muszę dostarczyć metody zezwalające podgląd i kasowanie danych publikacji. W sytuacji usunięcia publikacji przez użytkownika, system musi skasować wszelakie pliki z nią związane[6].

3.2.3. EPUB Core Media Types

Zaobserwować można, że EPUB jako format jest bardzo ściśle specyfikowany odnośnie jego struktury i wspieranych technologii. W przypadku samej zawartości nie jest inaczej. Specyfikacja EPUB 3 Core Media Types[3] szczegółowo opisuje jakie typy mediów mogą zostać załączone do publikacji, i określa dla nich unikalny typ (Media Type) który jest wykorzystywany w celach poinformowania systemu czytającego o typie danego elementu. Ta sekcja opisuje poszczególne typy mediów wspierane przez EPUB, nie zagłębiając się przy tym w techniczne szczegóły konkretnego typu.

Image Types

Wspierane typy obrazów (Image Types) to GIF, JPEG, PNG oraz SVG. Ich typy zadeklarowane w standardzie EPUB to kolejno "image/gif", "image/jpeg", "image/png", "image/svg+xml". Notacja ta wykorzystywana jest w atrybutach elementów nawiązujących (wskazujących na/deklarujących) konkretny obiekt odpowiedniego typu. Oznaczenia te możemy spotkać w manifeście publikacji.

```
1 <manifest>
2   ...
3   <item href="Images/image-1.jpg" id="id1" media-type="image/jpeg"/>
4   <item href="Images/image-2.jpg" id="id2" media-type="image/jpeg"/>
5   ...
6 </manifest>
```

Listing 3.3: Przykładowy fragment manifestu znajdującego się w pliku content.opf

Ten sposób oznaczenia elementów wykorzystywany jest w deklarowaniu każdego obiektu o wspierany przez EPUB typie.

Audio Types

EPUB wspiera pliki audio w formacie MP3("audio/mpeg") oraz MP4("audio/mp4"). Zapewne zdecydowano się na te dwa konkretne standardy ze względy na ich popularność oraz mały rozmiar dzięki

[3] Adres url: https://idpf.github.io/epub-cmt/v3/

stosunkowo wysokiej kompresji.

Video Types

Ze względu na rozbieżność preferencji wśród autorów oraz twórców specyfikacji, EPUB nie definiuje preferowanego typu wideo dla publikacji w swoim standardzie. IDPF proponuje dwa formaty kodowania, H.264 oraz WebM ale nie kładzie nacisku na jeden z nich, tę kwestię pozostawia na ten moment autorom. Spór dotyczący określenia konkretnego formatu plików wideo rozchodzi się o szerokie wsparcie H.264 przeciwko nieopatentowanemu, wolnemu formatowi WebM, który jednak wciąż nie jest spopularyzowanym standardem. IDPF deklaruje, że najprawdopodobniej w pełnym momencie preferowany standard zostanie określony, jednakże teraz jest jeszcze na to zbyt wcześnie[9].

Application, Text, Font Types

Pozostałe typy które wspiera EPUB zdecydowano się opisać razem w tej sekcji w skrócie ze względu na to, że wszystkie przysłużają się jednemu celowi jakim jest prezentacja treści. Są to czcionki w formacie WOFF("application/font-woff"), WOFF2("font/woff2") oraz OpenType/TrueType("application/font-sfnt"), pliki CSS("text/css"), skrypty("application/javascript"), dokumenty XHTML("application/xhtml+xml"), nawigacyjne pliki NCX("application/x-dtbncx+xml") oraz nakładki MediaOverlays3("application/smil+xml") wraz z plikami PLS("application/pls+xml"), które mają na celu odtworzenia tekstu w formie dźwiękowej[1].

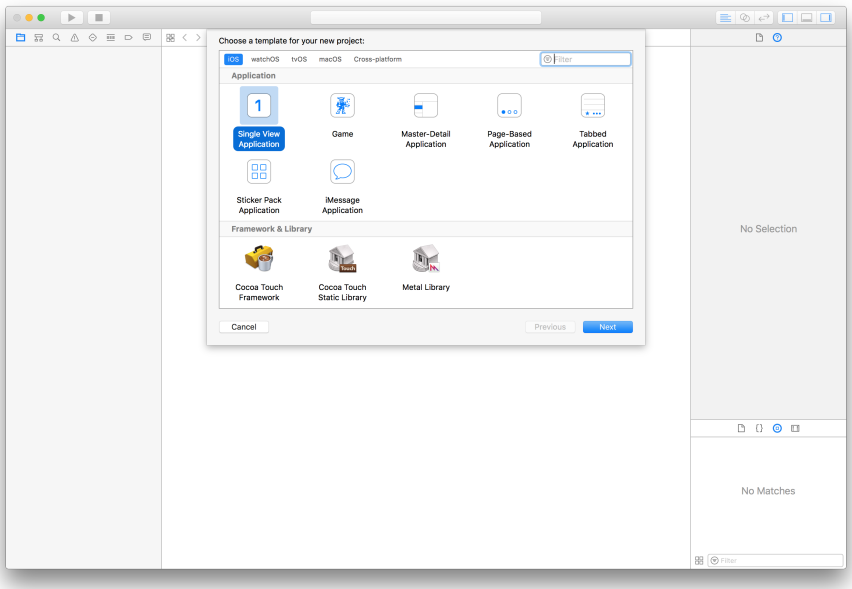
Foreign Resources

Należy wspomnieć iż EPUB zezwala na użycie niewspieranych formatów, jednakże nie gwarantuje, że zostaną prawidłowo obsłużone przez systemy czytające, od których wymaga jedynie wsparcia na wcześniej wymienionych formatów. W przypadku użycia przez autora publikacji niewspieranych formatów, specyfikacja EPUB wymaga od niego zadeklarowanie tak zwanego "Core Media Type fallback", czyli typu wspieranego który zastępczo będzie przypisany do obiektu jeżeli system czytający nie będzie wspierał pierwotnie wskazanego typu[3].

4. Framework EPUBKit

4.1. Tworzenie frameworku na iOS

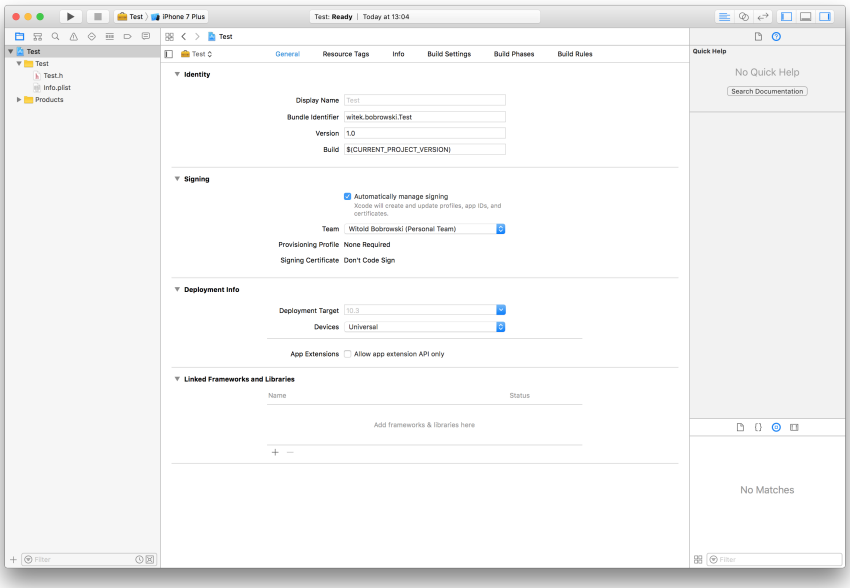
Tworzenie biblioteki którą zamierzamy następnie wykorzystać we własnej aplikacji lub udostępnić publiczne, jest stosunkowo prostym procesem. Wszystko sprowadza się do stworzenia nowego projektu w Xcode a następnie dołączenie go do przestrzeni roboczej (Xcode Workspace) w której znajdzie się projekt aplikacji nad którą pracujemy oraz projekt biblioteki. Aby stworzyć projekt biblopteki należy uruchomić Xcode IDE i na powitalnym ekranie wybrać opcję "Create a new Xcode project" która przeniesie nas do kolejnego ekranu z możliwością wybrania konktetnego szablonu projektu nad jakim chcemy pracować.



Rysunek 4.1: Szablny projektów które znajdują się w Xcode

Na naszym przypadku interesuje nas szablon "Cocoa Touch Framework", a po jego wybraniu jesteśmy proszeni o uzupełnienie formularza ze szczegółowymi informacjami na temat projektu który zamierzamy stworzyć, poczynając od nazwy, po język w którym będzie napisany (Swift lub Objective-C). Następnie Xcode poprosi o wskazanie lokalizacji na dysku w której chcemy zapisać projekt oraz zapyta nas czy chcemy stworzy repozytorium systemu kontroli wersji git. W tym momencie mamy projekt który można już w prosty sposób dołączyć do aplikacji (Co zostanie opisane w kolejnym rozdziale przy okazji

omawianie wykorzystania biblioteki EPUBKit w demonstracyjnej aplikacji). Teraz już możemy tworzyć klasy które mają składać sie na funkcjonalność biblioteki.



Rysunek 4.2: Szablony projektów które znajdują się w Xcode

Plik projektu pozwala nam za szczegółowy wgląd w preferencje oraz infromacje jego dotyczące, które można zmieniac w dowolnej chwili. Jest możliwość zmiany wersji bibioteki, docelowego urządzenia (iPhone lub iPad), preferowanej wersji systemu który ma wspierać bibltioteka, oraz co bardzo istotne, mamy możliwość użycia innych niezależnych bibliotek w naszym projekcie. Wystarczy wyeksportować plik projektu jako plik przestrzebi roboczej do której można dodać inne projekty a następnie połączyć je z naszym w menu "General" w polu "Linked Frameworks and Libraries" w pliku projektu. Dzięki tak funkcjonalnym i prostym w obsłudze narzędziom jak Xcode, po szybkiej konfiguracji swoją uwagę można skupić na samej logice którą chcemy zaimplementować.

W tym rozdziale zostanie opisana stworzna przez mnie biblioteka EPUBKit. Biblioteka ta jest oparta na architekturze MVC (Model View Controller) dlatego omówienie jest rozpocznę od opisania jej modelu oraz parsera, a następnie przejdę do zawartych w niej widokach które pozwalają na wyświetlenie danych zawartych w modelu. Zakończę rozdział przedstawiając możliwości dystrybuowania takiej biblioteki, przy pomocy szeroko stosowanych i popularnych narzędzi, które są podstawą iOS developmentu.

4.2. Model

Struktura klas modelu została zaprojektowana w ten sposób aby z jednej strony odzwierciedlała struktrę dokumnetu EPUB oraz format OPF który EPUB wykorzystuje, a z drugiej by trzymała się konwencji swiftowych i intuicyjnie reprezontowała obiekt który następnie będzie wykorzystywany w kolejnych klasach biblioteki.

```
Model
|-- EPUBDocument.swift
|-- EPUBManifest.swift
|-- EPUBMetadata.swift
|-- EPUBSpine.swift
`-- EPUBTableOfContents.swift
```

Listing 4.1: Struktura modelu EPUBKit.

4.2.1. EPUBDocument

EPUBDocumnet jest klasą publiczną reprezentującą całą poblikację EPUB i agregującą pozostałe struktury modelu biblioteki jako jej stałe własności (ze względu na nomenklaturę stosowaną w języku Swift, 'properties' będą nazywane własnościami). W języku Swift odróżniamy dwa rodzaje własności, są to stałe (słowo kluczowe let) do których wartość może zostać przypisana wyłącznie jednokrotnie oraz zmienne (słowo kluczowe var) których wartość może być modyfikowana w dowolnym momencie.

```
let (constant name): (type) = (expression)
var (variable name): (type) = (expression)
```

Listing 4.2: Deklaracje własności w Swifcie.[7]

Własności zostały oznaczone jako stałe ze względu na statyczną naturę struktury publikacji EPUB. Ciężko sobie wyobracić powód dla którego któreś z metadanych publikacji miały by zostać zmienione albo któreś z dokumentów XML usunięte z manifestu publikacji. Dlatego biorąc pod uwagę kontekst, w którym klasa się znajduje, zdecydowałem się oznaczenie jej własności jako stałe aby zapewnić klasie niemodalność oraz zgodność z wytycznymi odnośnie projektowania klas i struktór w języku Swift według których powinno oznaczać się własności jako stałe w każdej sytuacji która nie wymaga od nas ich mutowania, a słowa "mutowania" użyłem tutaj nie bez powodu, co stanie się jasne w kolejnym akapicie.

```
public class EPUBDocument {
    public let directory: URL
    public let contentDirectory: URL
    public let metadata: EPUBMetadata
    public let manifest: EPUBManifest
    public let spine: EPUBSpine
    public let tableOfContents: EPUBTableOfContents
}
```

Listing 4.3: Klasa EPUBDocument i jej stałe publiczne.

W przeciwieństwie do C, struktury w Swiftcie mogą posiadać metody. W przypadku gdy metoda w jakiś sposób zmiania własności musi ona zostać oznaczona słowem kluczowym 'mutating' co tyczy się również metod enumeracji. Wspominam o tym, ponieważ chciałbym wytłumaczyc dlaczego typy własności klasy EPUBDocument są strukturami a nie klasami. Ze względu na to, że instancje klasy są przekazywane przez referencję, a instancje struktur są przekazywane przez kopiowanie wartości, oznacz to że są one przeznaczone do innych zadań. Zgodnie z wytycznymi Apple, strukturami powinno ozna-

czać się typy, których zadaniem jest enkapsulacja relatywnie prostych wartości[8], co jest prawdą w przypadku wcześniej wspomnianych typów (zostaną one opisane w kolejnych paragrafach).

Klasa EPUBDocument posiada dwa inicjalizatory które pozwalają tworzyć instancje tej klasy. Pierwszym z nich jest inicjalizator prywatny w nomenklaturze swiftowej "memberwise initializer" ze względu na kolejność argumentów które przyjmuje, która zgodna jest z kolejnością deklajacji własności. Inicjalizator ten został oznaczony jako prywatny ponieważ jego przeznaczeniem jest inicjalizować instancję jedynie przy pomocy drugiego inicjalizatora. Drugi z inicjalizatorów jest dostępny publicznie, i jest jedynym publicznym inicjalizatorem dla tej klasy.

```
1  private init (directory: URL, contentDirectory: URL, metadata: EPUBMetadata,
       manifest: EPUBManifest, spine: EPUBSpine, toc: EPUBTableOfContents) {
2      self.directory = directory
3      self.contentDirectory = contentDirectory
4      self.metadata = metadata
5      self.manifest = manifest
6      self.spine = spine
7      self.tableOfContents = toc
8  }
9
10 public convenience init?(named: String) {
11     let parser = try? EPUBParser(named: named)
12     guard let directory = parser?.directory,
13         let contentDirectory = parser?.contentDirectory,
14         let metadata = parser?.metadata,
15         let manifest = parser?.manifest,
16         let spine = parser?.spine,
17         let tableOfContents = parser?.tableOfContents else { return nil }
18     self.init(directory: directory, contentDirectory: contentDirectory, metadata:
           metadata, manifest: manifest, spine: spine, toc: tableOfContents)
19 }
```

Listing 4.4: Inicjalizatory klasy EPUBDocument.

Zważając na naturę publikacji EPUB jako spójnej całości, zdecydowałem ograniczyć się inicjalizowanie klasy EPUBDocument do inicjalizatora pomocniczego, który wykorzystuje do tego parser. Ten inicjalizator wykorzystuje w pełni możliwości Swifta. Oznaczając go słowem kluczowym "convenience", zmuszam go do wykorzystania wyznaczonego (z ang. designated) inicjalizatora ponieważ pomocniczy inicjalizator nie może samemu tworzyć instancji, musi do tego wykorzystać wyznaczony inicjalizator (w tym przypadku jest to pierwszy inicjalizator, który jest prywatny). Dodatkowo pomocniczy inicjalizator, jest oznaczony znakiem zapytania "init?" co oznacza, że inicjalizacje może się niepowieść, a w takiej sytuacji inicjalizator zwróci... nic, czyli "nil" w Swifcie. Konsekwencją tego jest to, że typ która zwraca ten inicjalizator to "EPUBDocument?" a nie "EPUBDocument", co oznacza że może on nie mieć żadnej wartości co trzeba w odpowiedni sposób obsłużyć. Przeanalizujmy więc krok po kroku operacje, które wykonuje inicjalizator pomocniczy.

```
    let parser = try? EPUBParser(named: named)
```

Na wstępie tworzy instancję parsera, i jako argument inicjalizatora podaje własny parametr który wskazuje na nazwę publikacji EPUB. Słowo kluczowe "try" onacza, że inicjalizator może zwrócić błąd, a dzięki znakowi zapytania błąd ten gdy zostanie rzucony, będzie interpretowany jako zwrócenie nila przez inicjalizator. W ten sposób unikamy umieszczenia bloku "do-catch" co znacznie upraszcza kod. Końcowo znajdujemy się w posiadaniu stałej "parser", która jest typu "EPUBParser?" czyli opcjonalny EPUBParser.

```
guard let directory = ... else { return nil }
```

Wyrażenie "guard let" jest jednym ze sposobów obsłużenia typu opcjonalnego. Jest to odmiana wyrażenia "if let" które pozwala nam na przypisanie wartości zmiennej a, do nowej stałej b, jeżeli zmienna a takową posiada. Wadą takiego rozwiązania jest to, że nowo powstała zmienna b, znajduje się jedynie w zasięgu bloku "if", co w pewien sposób ogranicza dostęp do niej. Z pomocą przychodzą wyrażenia "guard", dzięki który zadeklarujemy nową stałą, która będzie przyjmowała wartość zmiennej, którą chcemy "rozpakować" (z ang. unwrap, co odnosi się do czynności wywłaszczania wartości z typu opcjonalnego) i będzie ona dostępna w obrębie tego samego bloku co wyrażenie guard. Dodatkowo mamy możliwość wykonania jakiejść czynności w sytuacji gdy zmienna którą rozpakowywujemy nie ma wartości, co w tym konkretnym przypadku będzie oznaczało niepowodzenie wywłaszczenia którejś z wartości parsera a więc inicjalizator EPUBDocument zwróci nil. Ponieważ wyrażenie "guard" działa w podobny sposób co "if", otrzymuje on również te same funkcjonalności co "if" w Swifcie, czyli możliwość kolejkowania wyrażeń zwracających wartość boolowską (wypisujemy je kolejno po przecinku), i w przypadku zwrócenia fałszu przez jedno z nich, instrukcja natychmiast zostaje przerwana a pozostałe wyrażenia nie zostają ewaluowane, a w tym przypadku program przechodzi do bloku "else".

```
self.init(...)
```

Jeżeli udało się wywłaszczyć wszystkie potrzebne wartości z parsera, to można przejść do tworzenia instancji EPUBDocument. Inicjalizator pomocniczy wywołuje inicjalizator wyznaczony, i dokument zostaje pomyślnie stworzony a wszystkie informacie otrzymane dzięki parserowi zostają przypisane na stałe do jednej instancji EPUBDocument. W ten sposób zostaje zachowana niemutowalność instancji, oraz gwarancja, że wszystkie wartości w których posiadaniu znajduje się instancja, pochodzą z jednego źródła, z którego czerpie parser. Działanie samego parsera zostawiam na kolejny podrozdział.

4.2.2. EPUBManifest

Jak juz wspomniano w rozdziale opisującym format EPUB, jego struktura jest oparta o standard OPF a to oznacza, że znajduję się w nim wykaz (manifest) wszystkich dokumentów oraz zasobów na które składa się dokument. Każdy element wymieniony w manifescie posiada swoje ID, wskazaną ścieżke w strukturze dokumentu oraz typ (Media Type). Parser starannie analizuje manifest i tworzy strukturę w której posiadaniu następnie znajduje się instancja EPUBDocument o czym więcej przy okazji omawiania parseru.

```
1 public struct EPUBManifest {
2     public struct Item {
3         public var id: String
```

```
4           public var path: String
5           public var mediaType: EPUBMediaType
6           public var property: String?
7       }
8
9       public var id: String?
10      public var items: [String:Item]
11
12      public func path(forItemWithId id: String) throws -> String {
13          if let item = items[id] {
14              return item.path
15          } else {
16              throw EPUBParserError.noPathForItem(id)
17          }
18      }
19  }
```

Listing 4.5: Struktura EPUBManifest.

EPUBManifest deklaruje własną strukturę Item, na którą skłda się kilka własności opisujących daną pozycję w manifescie. EPUBManifest posiada dwie własniości. Pierwszą z nich jest opcjonane ID manifestu które może się pojawić w publikacji EPUB, ale w specyfikacji nie jest określone jako wymagane pole. Drugą własnością jest słownik, którego kluczem jest ID elementu a wartością jest instancja struktury Item. Ponieważ manifest nie musi być listą posortowaną, zdecydowałem się na użycie słownika jako struktury danych która ma przechowywać wszystkie jego elementy, dzięki czemu dostęp do nich jest natychmiastowy. Należy zwrócić uwagę na to, że nie został zadeklarowany inicjalizator dla EPUBManifest, tak jak miało to miejsce przy EPUBDocument. Powód jest następujący, Swift w przypadku gdy żaden inicjalizator nie został zadeklarowany dostarcza domyślny "memberwise initializer" dzięki czemu w EPUBManifest dostajemy go "za darmo", natomist w przypadku struktury EPUBDocument został zadeklarowany pomocniczy inizjalizator przez co domyślny inicjalizator nie został dostarczony przez swift. EPUBManifest posiada publiczną metodę która przyjmuje jako argument id elementu zwraca do niego ścieżkę w przypadku gdzy element znajduje się w słowniku "items". Słownik w swifcie zwraca wartość o typie opcjonalnym, ponieważ nie ma żadnej gwarancji, że do podanego przez nas kluczu przypisana jest jakaś wartość. Zastosowano tutaj rozpakowanie typu opcjonalnego przy pomocy wyrażenia "if let" aby otrzymać ścieżkę elementu. W przypadku gdy znaduje się on w słowniku zostanie on przypisany do nowej stałej której typ już nie jest opcjonalny więc posiadanie przez niej jakiejś wartości jest zagwarantowane. Dodatkowo w przypadku gdyby taki element manifestu o wskazanym ID nie istniał, zostanie rzucny błąd co zostało oznaczone słowem kluczowym "throws" przy deklaracji zwracanego typu.

```
func (function name)((parameters)) throws -> (return type) {
    (statements)
}
```

Listing 4.6: Funkcje i metody które mogą rzucać błędy, przy deklaracji muszą zostać oznaczone słowem kluczowym "throws"[7].

W kontekście EPUBManifest pozostało jeszcze wspomnieć o typie enumeracji, który został stworzony aby określać charakter elementu znajdującego się w publikacji EPUB, i wymienionego w Manifescie. Mowa tutaj o EPUBMediaType, enumeracji która posiada powiązany typ (Associated Type) którym jest typ String.

```
1  public enum EPUBMediaType: String {
2      case gif = "image/gif"
3      case jpeg = "image/jpeg"
4      case png = "image/png"
5      case svg = "image/svg+xml"
6      case xHTML = "application/xhtml+xml"
7      case rfc4329 = "application/javascript"
8      case opf2 = "application/x-dtbncx+xml"
9      case openType = "application/font-sfnt"
10     case woff = "application/font-woff"
11     case mediaOverlays = "application/smil+xml"
12     case pls = "application/pls+xml"
13     case mp3 = "audio/mpeg"
14     case mp4 = "audio/mp4"
15     case css = "text/css"
16     case woff2 = "font/woff2"
17     case unknown
18 }
```

Listing 4.7: Enumeracja EPUBMediaType.

Enumeracja ta deklaruje wszystkie przypadki typu mediów wspieranych przez standard EPUB i wymienionych w specyfikacji. Dzięki tej enumeracji element manifestu posiada własność które jest ograniczona do kilku przypadków, a w sytuacji potrzeby obsłużenia takiego elementu w prosty sposób można zdeterminować jego rodzaj. Powiązana wartość dla każdego przypadku jest ciągiem znaków reprezentującym typ określony w specyfikacji EPUB, i gwarantuje nam ona prostą inicjalizację przez podanie wartości jako argument inicjalizatora enumeracji.

4.2.3. EPUBMetadata

Kolejny elementem wymaganym przez OPF jest metadata, który enkapsuluje meta informacje na temat konkretnej interpretacji zawartej w publikacji. W celach reprezentacji tych meta danych w bibliotece EPUBKit została swtorzona struktura EPUBMetadata.

```
1  public struct EPUBMetadata {
2      public struct Creator {
3          public var name: String?
4          public var role: String?
5          public var fileAs: String?
6      }
7      public var contributor: Creator?
8      public var coverage: String?
9      public var creator: Creator?
10     public var date: String?
```

```
11      public var description: String?
12      public var format: String?
13      public var identifier: String?
14      public var language: String?
15      public var publisher: String?
16      public var relation: String?
17      public var rights: String?
18      public var source: String?
19      public var subject: String?
20      public var title: String?
21      public var type: String?
22      public var coverId: String?
23  }
```

Listing 4.8: Struktura EPUBMetadata.

EPUBMetadata definiuje własny publiczny typ pomocniczy Creator aby w lepszy sposób reprezentować element twórcy i współtwórcy publikacji którzy mogą zostać wymienieni w meta danych publikacji EPUB. Wszystkie własności EPUBMetadata posiadają typ opcjonalny, ponieważ ich obecność w dokumencie EPUB nie jest zagwarantowana. Inicjalizator nie jest obecny, ponieważ po raz kolejny jest dostarczony domyslny inicjalizator przez Swift. Własności są oznaczone jako zmienne publiczne, ponieważ udostępnienie ich globalnie ma sens ze względu na ich informatywny cel. Własność metadata w klasie EPUBDocument jest stałą, a więc pomimo tego iż w deklaracji struktury EPUBMetadata jej własności są zmiennymi, to w momencie przypisania jej instancji do instancji klasy EPUBDocument wartości zmiennych nie mogą zostać zmienione, co wynika z natury struktury w swiftcie która jest kopiowana przez wartość. Mutowanie wartości zmiennych w instancji struktury EPUBMetadata, która nie znajduje się w kontekście całego dokumentu, ma sens a przynajmniej nie jest niedopuszczalne. W przypadku gdyby parser znajdujący się w posiadaniu takiej instancji, w pewnym momencie wywłaszczył pewne dodatkowe dane z dokumentu, warto pozwolić mu na uaktualnienie ich w strukturze. Niemutowalność zostaje zagwarantowana dopiero w momencie przypisania jej jako stałej w klasie EPUBDocument.

4.2.4. EPUBSpine

Element "spine" w publikacji EPUB definiuje kolejność w jakiej elementy manifestu są uporządkowane, czyli w jakiej kolejności należy je wyświetlać. Element ten znajduje swoją reprezentację w bibliotece EPUBKit, jako struktura EPUBSpine.

```
1  public struct EPUBSpine {
2      public struct Item {
3          public var id: String?
4          public var idref: String
5          public var linear: Bool
6      }
7
8      public var id: String?
9      public var toc: String?
10     public var pageProgressionDirection: EPUBPageProgressionDirection?
```

```
11      public var items: [Item]
12  }
```

Listing 4.9: Struktura EPUBSpine.

EPUBSpine podobnie do EPUBManifest deklaruje własny typ pomocniczy Item, które odzwierdziedla "itemref" znajdujący się w "spine" publikacji EPUB. Item posiada własności które odnoszą się do identyfikatora elementu manifestu (idref), dodatkowego indentyfikatora, oraz istotnej informaci, czy element powinien zostać wyświetlony w kolejności liniowej, czy nie. Struktura EPUBSpine składa się z pola "id", "toc" które opcjonalnie może posiadać identyfikator spisu treści jeżeli takowy się znajduje w dokumencie, własności która definiuje kierunek w którym powinno się wyświetlać elementy dokumentu, oraz tablicę elementów własnego typu Item.

```
1  public enum EPUBPageProgressionDirection: String {
2      case leftToRight = "ltr"
3      case rightToLeft = "rtl"
4  }
```

Zmienna "pageProgressionDirection" posiada typ który został zadeklarowany przeze mnie jako enumeracja EPUBPageProgressionDirection składająca się z dwóch przypadków, pierwszego który reprezentuje kolejność czytania od lewej do prawej oraz drugiego, który mówi o kierunku przeciwnym. W przypadku gdy kolejność nie zostanie zdefiniowana w publikacji, decyzja jak dokument powinien zostać wyswietlony zostaje pozostawiona systemowi czytającemu. W celu szybkiej inicjalizacji i zdefiniowaniu odpowiedniego kierunku dzieki powiązanemu typowi, parser podaje jako argument inicjalizatora enumeracji wartość która wywłaszczył z publikacji EPUB, która przyjmuje wartość "rtl" lub "ltr".

4.2.5. EPUBTableOfContents

Ostatnia struktura modelu reprezentuje spis treści zawarty w publikacji EPUB. Pomimo tego iż w wersji trzeciej standardu nie jest wspierany w takiej formie jak w poprzednich wersjach, to ze względu na zapewnienie wsparcia tym starszym wersjom publikacji zdecydowałem się zaimplementować tą relatywnie prostą strukturę. Spis treści przedstawiony w formie pliku typu NCX służył w poprzednich wersjach EPUB użytkownikowi do nawigacji w obrębie dokumentu, jednak specyfikacja wymaga od systemu czytającego ignorowania tego pliku w publikacjach wersji trzeciej.

```
1  public struct EPUBTableOfContents {
2      public var label: String
3      public var id: String
4      public var item: String?
5      public var subTable: [EPUBTableOfContents]?
6  }
```

Listing 4.10: Struktura EPUBTableOfContents.

EPUBTableOfContents posiada własności które znamy już z poprzednich struktur, jednakże to co ją od nich wyróżnia to możliwość zagnieżdzenia. Ponieważ rozdziały mogą posiadać podrozdziały, a te z kolei mogą być podzielone na jeszcze mniejsze części, co może zostać uwzględnione w takim spisie

treści, konieczne jest pozwolenie przechowywania przez każdą pozycje swojej własnej tablicy. A skoro sam spis posiada takie same atrybuty co jej elementy, postanowiłem nie tworzyć dodatkowej struktury Item jak we wcześniej omówionych strukturach. Swift zapewnia nam domyślny inizjalizator więc po raz kolejny unikamy konieczności implementowania go ręcznie.

4.3. Parser

Pierwotnie planowałem opisać Parser w rozdziale poświęconym modelowi, ponieważ w architekturze MVC bibliteki EPUBKit jego rolna mocno wpasowywuje się w definicję modelu, ponieważ to własnie on go tworzy. Jednakże zdecydowałem, że rozsądniej będzie gdy przeznacze na to osobny podrozdział ze względu na to jak obszerny jest to temat ze względu na jego architekturę, oraz ilość wykonywanych operacji.

```
1  Utils
2  |-- EPUBParsable.swift
3  |-- EPUBParser.swift
4  `-- EPUBParserError.swift
```

Listing 4.11: Struktura folderu narzędzi służacych modelowi EPUBKit.

4.3.1. EPUBParsable

Aby praca parsera była dobrze zorganizowana, zdecydowałem się zaimplementować wzorzec projektory Budowniczy (ang. Buildier), ponieważ jest on zgodny z naturą parsera, który wywłaszcza informacje z publikacji EPUB a następnie na ich podstawie buduje model EPUBKit którym jest EPUBDocument. Pierwszym krokiem w celu zaimplementowania wzorcu Budowniczego było stworzenie protokołu, który zostanie zastosowany przez parser.

```
1 protocol EPUBParsable {
2     func unzip(archive name: String) throws -> URL
3     func getContentPath(from bookDirectory: URL) throws -> URL
4     func getMetadata(from xmlElement: AEXMLElement) -> EPUBMetadata
5     func getManifest(from xmlElement: AEXMLElement) -> EPUBManifest
6     func getSpine(from xmlElement: AEXMLElement) -> EPUBSpine
7     func getTableOfContents(from xmlElement: AEXMLElement) -> EPUBTableOfContents
8 }
```

Listing 4.12: Protokół EPUBParsable.

Protokół ten deklaruje metody, które każda stosująca go klasa lub struktura będzie musiała zaimplementować. Konwencja nazewnictwa protokołów w swifcie jest taka, aby nazywać je tak aby dobrze określały naturę klasy którą go stosuje, aby pasowały do cech które nadają klasom. Jeżeli protokół opisuje czym klasa jest, to powinien być czytany jako rzeczownik jak np. protokół standardowej biblioteki swifta MutableCollection. Protokoły, które opisują funkcjonalność, pewne dodatkowe możliwości powinny być kończyć się -ible lub -able. Dzięki tej konwencji, odrazu staje się jasne jakich cech nabywa klasa juz w momencie oznaczenia że stosuje ona dany protokół. Zaimplementowanie przez stosującą

protokół klasę jego metod jest wymagane, ponieważ na ten moment nie istnieje żadna implementacja domyślna. Swift pozwala na dostarczenie tak zwanych rozszeżeń deklarowanych typów przy pomocy słowa kluczowego "extension". Dzięki tym rozszerzeniom deklarację typu można podzielić na schludnie wyglądające i dobrze zorganizowne bloki kodu, co zostanie przedstawione przy omawianiu klasy EPUBParser.

```
extension (type name): (adopted protocols) {
    (declarations)
}
```

Listing 4.13: Deklaracja wyrażenia "extension"[7].

Takie rozszeżenie można wykorzystać również w protokołach (które są typem tak samo jak klasa czy struktura), i dostarczyć im domyślną implementacje metody lub zadeklarować nowe. Jednakże w tym przypadku nie będę tego robił ze względu na to iż EPUBParser powinien zaimplementować każdą z nich w kontekście konkretnego dokumentu.

4.3.2. EPUBParser

Zadaniem parsera jest zebranie wszystkich informacji na temat publikacji i enkapsulacji ich w zmienncyh o typach które posiada EPUBDocument, i pełni rolę budowniczego. Jego deklaracja to zbiór właściwości o typie opcjonalnym oraz inicjalizator, w którym odbywa się cały proces budowania modelu, poprzez parsowanie publikacji EPUB.

```
1  class EPUBParser {
2      var directory: URL?
3      var contentDirectory: URL?
4      var metadata: EPUBMetadata?
5      var manifest: EPUBManifest?
6      var spine: EPUBSpine?
7      var tableOfContents: EPUBTableOfContents?
8
9      init(named: String) throws {
10         do {
11             directory = try unzip(archive: named)
12             let contentPath = try getContentPath(from: directory!)
13             contentDirectory = contentPath.deletingLastPathComponent()
14             let data = try Data(contentsOf: contentPath)
15             let content = try AEXMLDocument(xml: data)
16             metadata = getMetadata(from: content.root["metadata"])
17             manifest = getManifest(from: content.root["manifest"])
18             spine =  getSpine(from: content.root["spine"])
19             let tocPath = contentDirectory!.appendingPathComponent(try manifest!.
                   path(forItemWithId: spine?.toc ?? ""))
20             let tocData = try Data(contentsOf: tocPath)
21             let tocContent = try AEXMLDocument(xml: tocData)
22             tableOfContents = getTableOfContents(from: tocContent.root)
23         } catch {
24             print(error.localizedDescription)
```

```
25              throw error
26          }
27      }
28  }
```

Listing 4.14: Klasa EPUBParser.

W przeciwieństwie do wcześniej zadeklarowanych typów, EPUBParser tak jak i protokół EPUBParsable, nie są oznaczone jako publiczne, więc nie mogą zostać użyte poza biblioteką. Domyślnym parametrem dostępu w Swifcie jest "internal", który ogranicza typ, czy właściwość do użytku w obrębie jednego projektu, w tym przypadku biblioteki. Powodem tego jest sposób w jaki zorganizowana została inizjalizacja EPUBDocument. Osobie z zewnątrz najprawdopodobniej nie zależy aby się dowiedzieć w jaki sposob powstaje instancja dokumentu, dla niej ważne jest to, że może otrzymać ją podając jako argument inicjalizatora nazwę pliku, a cały proces będzie odbywał się poza jego zasięgiem. EPUBParser odgrywa tutaj rolę budowniczego, i w momencie rozpoczęcia inicjalizacji dokumentu jego własna instancja zostaje stworzona i "oddelegowana" do wykonania zadania do którego jest przeznaczona. Klasa EPUBParser nie ma innego zastosowania, i nie ma absolutnie żadnego powodu aby używać jej niezależnie i do innych celów niż ten jeden, z góry narzucony. Dlatego też ani klasa, ani jej własności nie są oznaczone publicznymi. Warto zwrócić uwagę na to, iż w swojej deklaracji EPUBParser nie stosuje protokołu EPUBParsable, co wynika ze stylu i wytycznych apple odnośnie projetkowania klas. Klasa powinna deklarować to co dla niej najbardziej elementarne, a dopiero przy użyciu "extension" funkcjonalność powinna zostawać do niej dodawana. I tak jest w tym przypadku, gdzie zadeklarowane są zmienne oraz inicjalizator który już wykonuje metody których jeszcze są dla niego nie znane, a zostaną dopiero zaimplementowane po zastosowaniu protokołu.

```
1  //MARK: - EPUBParsable
2  extension EPUBParser: EPUBParsable {
3      func unzip(archive named: String) throws -> URL { ... }
4      func getContentPath(from bookDirectory: URL) throws -> URL { ... }
5      func getMetadata(from content: AEXMLElement) -> EPUBMetadata { ... }
6      func getManifest(from content: AEXMLElement) -> EPUBManifest { ... }
7      func getSpine(from content: AEXMLElement) -> EPUBSpine { ... }
8      func getTableOfContents(from toc: AEXMLElement) -> EPUBTableOfContents { ... }
9  }
```

Listing 4.15: Klasa EPUBParser stosuje protokół EPUBParsable.

Dobrą praktyką jest dołączenie komentarzu przy każdym rozszeżeniu aby oznaczyć dodatkowo co oferuje dane rozszeżenie. Należy również pamiętać aby dla każdego stosowanego protokołu przeznaczyć osobne rozszeżenie w celu zwięszenia czytelności kodu. Omówię teraz, krok po kroku w jaki sposób postępuje inicjalizacja parsera, i w jaki sposób wywłaszcza on dane z publikacji w formacie EPUB prezentując kod implementacji każdej z metod protokołu.

```
  init(named: String) throws {
      do {
          ...
      } catch {
```

```
            print(error.localizedDescription)
            throw error
        }
    }
```

Inicjalizator przyjmuje jako argument nazwę pliku publikacji, i składa się bloku do-catch w którym odbywa się parsowanie. W przypadku gdy któraś z metod rzuci błąd, zostanie on rzucony dalej i zostanie wyświelnona na konsoli informacja o błędzie, który został napotkany. Błędy, które mogą zostać rzucone są typu EPUBParserError, który zostanie omówiony pod koniec podrozdziau dotyczącego parsera. Przejdźmy zatem do instrukcji znajdujących się w bloku do-catch.

```
11      directory = try unzip(archive: named)
```

Pierwszą czynnością parsera jest rozpakowanie archiwum .epub, które w rzeczywistości jest archiwum w stylu ZIP, co omówiono dokładnie w rozdziale trzecim. Dzięki wykorzystaniu open-sourcowej biblioteki ZIP, proces jest ten wyjątkowo prosty.

```
func unzip(archive named: String) throws -> URL {
    Zip.addCustomFileExtension("epub")
    do {
        let filePath = Bundle.main.url(forResource: named, withExtension: "epub"
            )!
        let unzipDirectory = try Zip.quickUnzipFile(filePath)
        return unzipDirectory
    } catch ZipError.unzipFail {
        throw EPUBParserError.unZipError
    }
}
```

Listing 4.16: Implementacja metody unzip(archive named:).

W pierwszej kolejności metoda dodaje do klasy Zip rozszerzenie ".epub" aby bilbioteka rozpoznała plik jako archiwum. Następnie w bloku do-catch dochodzi do ustalenia lokalizacji pliku w paczce aplikacji oraz rozpakowania archiwum metodą, która zwraca ścieżkę do docelowego folderu, w którym znajduje się rozpakowana publikacja EPUB. Jeżeli proces rozpakowania przebiegnie bezbłędnie, metoda zwróci ścieżkę.

```
12      let contentPath = try getContentPath(from: directory!)
13      contentDirectory = contentPath.deletingLastPathComponent()
```

Inicjalicator wywołuje następnie metodę, która lokalizuje w publikacji obowiązkowy plik "content.opf" w folderze "META-INF" a następnie szuka w nim elementu "rootfile", który jest plikiem OPF, opisującym zawartość całej publikacji.

```
func getContentPath(from bookDirectory: URL) throws -> URL {
    do {
        let path = bookDirectory.appendingPathComponent("META-INF/container.xml"
            )
        let data = try Data(contentsOf: path)
        let container = try AEXMLDocument(xml: data)
```

```
            let content = container.root["rootfiles"]["rootfile"].attributes["full-
                path"]
            return bookDirectory.appendingPathComponent(content!)
        } catch {
            throw EPUBParserError.containerParseError
        }
    }
```

Listing 4.17: Implementacja metody getContentPath(from bookDirectory:).

Po raz kolejny instrukcje metody znajdują się w bloku do-catch aby zapewnić obsługę ewentualnych błędów. W przypadku zlokalizowania pliku OPF, ścieżka do niego zostaje zwrócona a w przeciwnym wypadku, zostanie rzucony błąd. Jeżeli funkcja zwróci ścieżkę pliku OPF, inicjalizator przypisze do zmiennej "contentDirectory" ścieżkę do folderu w którym się on znajduje przez usunięcie ostatniego komponentu ściezki.

```
14          let data = try Data(contentsOf: contentPath)
15          let content = try AEXMLDocument(xml: data)
```

Znając już lokalizację pliku OPF, należy go sparsować przy użyciu zewnętrzenej biblioteki AEXML. Dzięki temu dostęp do elemnetów dokumentu będzie znacznie uproszczony, dokument zostanie przedstawiony w formie klasy AEXMLDocument. Jeżeli próba inicjalizacji klas Data oraz AEXMLDocument przebiegnie poprawnie, otrzymamy zmienna "content", z której parser będzie wywłaszczał kluczowe infromacje dotyczące dokumentu.

```
16          metadata = getMetadata(from: content.root["metadata"])
17          manifest = getManifest(from: content.root["manifest"])
18          spine =  getSpine(from: content.root["spine"])
```

Inicjalizator gdy znajdzie się w posiadaniu dokumentu OPF w formie dokumentu AEXMLDocument, będzie następnie podawał jako argumenty odpowiednim funkcjom jego elementy.

```
    func getMetadata(from xmlElement: AEXMLElement) -> EPUBMetadata {
        var metadata = EPUBMetadata()
        metadata.contributor = EPUBMetadata.Creator(name: content["dc:contributor"].
            value,
                                                    role: content["dc:contributor"].attributes["
                                                        opf:role"],
                                                    fileAs: content["dc:contributor"].attributes[
                                                        "opf:file-as"])
        metadata.coverage = content["dc:coverage"].value
        metadata.creator = EPUBMetadata.Creator(name: content["dc:creator"].value,
                                                role: content["dc:creator"].attributes["opf:role"
                                                    ],
                                                fileAs: content["dc:creator"].attributes["opf:
                                                    file-as"])
        metadata.date = content["dc:date"].value
        metadata.description = content["dc:description"].value
        metadata.format = content["dc:format"].value
        metadata.identifier = content["dc:identifier"].value
```

```
        metadata.language = content["dc:language"].value
        metadata.publisher = content["dc:publisher"].value
        metadata.relation = content["dc:relation"].value
        metadata.rights = content["dc:rights"].value
        metadata.source = content["dc:source"].value
        metadata.subject = content["dc:subject"].value
        metadata.title = content["dc:title"].value
        metadata.type = content["dc:type"].value
        for metaItem in content["meta"].all! {
            if metaItem.attributes["name"] == "cover" {
                metadata.coverId = metaItem.attributes["content"]
            }
        }
        return metadata
    }
```

Listing 4.18: Implementacja metody getMetadata(from xmlElement:).

Działanie metody agregującej meta dane jest relatywnie proste, musi ona przeanalizować elementy zawarte w przekazanej instancji AEXMLElement. Elementy te są oznaczone w sposób określony w specyfikacji, więc nazwane onę będa zawsze w ten sam sposób co znacznie upraszcza proces budowy instancji EPUBMetadata.

```
    func getManifest(from xmlElement: AEXMLElement) -> EPUBManifest {
        var items: [String:EPUBManifest.Item] = [:]
        for item in xmlElement["item"].all! {
            let id = item.attributes["id"]!
            let path = item.attributes["href"]!
            let mediaType = item.attributes["media-type"]
            let properties = item.attributes["properties"]
            items[id] = EPUBManifest.Item(id: id, path: path, mediaType:
                EPUBMediaType(rawValue: mediaType!) ?? .unknown, property:
                properties)
        }
        return EPUBManifest(id: xmlElement["id"].value, items: items)
    }
```

Listing 4.19: Implementacja metody getManifest(from xmlElement:).

Metoda "getManifest(from content:)" w pierwszej kolejności tworzy instancję słownika, który znamy ze struktury EPUBManifest. reprezentuje on wszystkie elementy wymienione w manifeście. Następnie iteruje po tablicy elementów zawartych w parametrze "xmlElement", i inicjalizuje przy każdym z nich nową instancję struktury EPUBManifest.Item z informacji, które udało jej się wywłaszczyć.

```
    func getSpine(from xmlElement: AEXMLElement) -> EPUBSpine {
        var items: [EPUBSpine.Item] = []
        for item in xmlElement["itemref"].all! {
            let id = item.attributes["id"]
            let idref = item.attributes["idref"]!
            let linear = (item.attributes["linear"] ?? "yes") == "yes" ? true :
                false
```

```
            items.append(EPUBSpine.Item(id: id, idref: idref, linear: linear))
        }
        let pageProgressionDirection = xmlElement["page-progression-direction"].
            value ?? ""
        return EPUBSpine(id: xmlElement.attributes["id"], toc: xmlElement.attributes
            ["toc"], pageProgressionDirection: EPUBPageProgressionDirection(rawValue
            : pageProgressionDirection), items: items)
    }
```

Listing 4.20: Implementacja metody getSpine(from xmlElement:).

W identycznym stylu działa metoda "getSpine(from content:)", kórej zadaniem jest stworzenie instancji struktury EPUBSpine.

```
19          let tocPath = contentDirectory!.appendingPathComponent(try manifest!.path(
                forItemWithId: spine?.toc ?? ""))
20          let tocData = try Data(contentsOf: tocPath)
21          let tocContent = try AEXMLDocument(xml: tocData)
22          tableOfContents = getTableOfContents(from: tocContent.root)
```

Ostatnimi czynnościami parsera są odszukanie w publikacji dokumentu typu NCX, który reprezentuje spis treści (table of contents) i przedstawienie go w formie struktury EPUBTableOfContents poprzez parsowanie pliku NCX.

```
    func getTableOfContents(from xmlElement: AEXMLElement) -> EPUBTableOfContents {
        let item = xmlElement["head"]["meta"].all(withAttributes: ["name":"dtb=uid"
            ])?.first?.attributes["content"]
        var tableOfContents = EPUBTableOfContents(label: xmlElement["docTitle"]["
            text"].value!, id: "0", item: item, subTable: [])

        func evaluateChildren(from map: AEXMLElement) -> [EPUBTableOfContents]{
            if let _ = map["navPoint"].all {
                var subs: [EPUBTableOfContents] = []
                for point in map["navPoint"].all! {
                    subs.append(EPUBTableOfContents(label: point["navLabel"]["text"
                        ].value!, id: point.attributes["id"]!, item: point["content"
                        ].attributes["src"]!, subTable: evaluateChildren(from: point
                        )))
                }
                return subs
            } else {
                return []
            }
        }
        tableOfContents.subTable = evaluateChildren(from: xmlElement["navMap"])
        return tableOfContents
    }
```

Listing 4.21: Implementacja metody getTableOfContents(from xmlElement:).

Ze wzgledu na możliwość zagnieżdżania elementów metoda deklaruje własną funkcję, która będzie odpowiedzialna za zbieranie informacji o elementach i penetrowanie w głąb kolejnych poziomów zagnieżdżenia.

W tym miejscu praca parsera kończy się. Jego inicjalizacja dobiegła końcowi, a metody z powodzeniem lub bez zebrały odpowiednie informacje o publikacji EPUB. To co dzieje się z danymi w parserze jest dokładnie opisane w podrozdziale opisującym EPUBDocument.

4.4. Widok

4.5. Dystrybucja

5. Aplikacja demonstracyjna

5.1. Tworzenie aplikacji na iOS

5.2. Wykorzystanie frameworku w aplikacji

5.3. Publikacja

6. Podsumowanie

Bibliografia

[1] Epub 3 core media types, OCT 2016.

[2] Epub 3.1, recommended specification, JAN 2017.

[3] Epub 3.1, recommended specification, JAN 2017.

[4] Epub content documents 3.1, recommended specification, JAN 2017.

[5] Epub content documents 3.1, recommended specification, JAN 2017.

[6] Epub content documents 3.1, recommended specification, JAN 2017.

[7] *The Swift Programming Language (Swift 4).* Apple Inc., 2017.

[8] *The Swift Programming Language (Swift 4).* Apple Inc., 2017.

[9] M. Garrish. *What is EPUB 3?* O'Reilly Media, 2011.