

**Akademia Górniczo-Hutnicza
im. Stanisława Staszica w Krakowie**

Wydział Elektrotechniki, Automatyki, Informatyki i Elektroniki

KATEDRA AUTOMATYKI



PRACA INŻYNIERSKA

KONRAD MALAWSKI

**PROTODOC
IMPLEMENTACJA ODPOWIEDNIKA NARZĘDZIA JAVADOC
DLA JĘZKA DEFINICJI INTERFEJSÓW
GOOGLE PROTOCOL BUFFERS**

PROMOTOR:

dr inż. Jacek Piwowarczyk

Kraków 2011

OŚWIADCZENIE AUTORA PRACY

OŚWIADCZAM, ŚWIADOMY ODPOWIEDZIALNOŚCI KARNEJ ZA POŚWIADCZENIE NIEPRAWDY, ŻE NINIEJSZĄ PRACĘ DYPLOMOWĄ WYKONAŁEM OSOBIŚCIE I SAMODZIELNIE, I NIE KORZYSTAŁEM ZE ŹRÓDEŁ INNYCH NIŻ WYMIENIONE W PRACY.

.....

PODPIS

AGH
University of Science and Technology in Krakow

Faculty of Electrical Engineering, Automatics, Computer Science and Electronics

DEPARTMENT OF AUTOMATICS



BACHELOR OF SCIENCE THESIS

KONRAD MALAWSKI

PROTODOC
DEVELOPMENT OF A JAVADOC TOOL EQUIVALENT FOR
THE GOOGLE PROTOCOL BUFFERS
INTERFACE DESCRIPTION LANGUAGE

SUPERVISOR:

Jacek Piwowarczyk Ph.D

Krakow 2011

Spis treści

1. Wprowadzenie	7
2. Cel pracy	8
3. Analiza obecnie dostępnych rozwiązań	9
3.1. Google Protoc.....	9
3.2. Idea plugin protobuf	10
3.3. Decyzja: własnoręczna implementacji Parsera przy pomocy <i>Scala Parser Combinators</i>	10
4. Projekt systemu	13
4.1. Architektura aplikacji	15
4.1.1. Zastosowany model typów wiadomości Protocol Buffers w Scala	17
4.1.2. Zastosowany model typów pól Protocol Buffers w Scala.....	18
5. Szczegóły implementacyjne	22
5.1. ProtoBufParser.....	23
5.1.1. Wprowadzenie do kombinatorów parserów.....	23
5.1.2. Fragmenty implementacji	23
5.2. Verifier	23
5.2.1. Obsługiwane weryfikacje	23
5.3. CodeGenerator.....	23
5.3.1. Język szablonów - Mustache.....	23
5.4. Przygotowanie środowiska do rozwoju ProtoDoc.....	24
5.4.1.	24
5.5. Instalacja narzędzia SBT	24
6. Zrzuty ekranu wygenerowanej dokumentacji	25
7. Rola Testów oraz TDD w procesie tworzenia aplikacji	28
7.1. Zaimplementowane specyfikacje.....	29
8. Zastosowanie ProtoDoc do automatyzacji dokumentacji projektów	33
9. Spis obsługiwanych funkcjonalności	34
A. Google Protocol Buffers	35
A.1. Krótka historia języka.....	35

A.2. Zestawienie wydajności mechanizmów serializacji na JVM	35
A.3. Przykładowe definicje wiadomości	35
A.4. Dostępne narzędzia.....	35
B. Podstawy języka Scala oraz Scala Parser Combinators	36
B.1. Krótka historia języka.....	36
B.2. Podstawy.....	36
B.3. Traits - wmieszanie zachowania do klasy.....	37
B.4. Scala Parser Combinators	39

1. Wprowadzenie

2. Cel pracy

Celem projektu jest implementacja narzędzia generującego dokumentację na podstawie plików *.proto zawierających zapisane przy pomocy „języka definicji interfejsów” (tzw. *Interface Description Language*, w skrócie *IDL*) - Google Protocol Buffers.

Potrzebę implementacji takiego narzędzia motywuję doświadczeniem w pracy z Protocol Buffers, gdy mamy do czynienia z dużą ilością plików *.proto (setki). Brak automatycznie generowanej dokumentacji tak dużego zbioru wiadomości znacznie utrudniał zapoznanie się z systemem oraz przystąpienie do sprawnej pracy z nim. Gdyby taka, zawsze aktualna, dokumentacja była dostępna w firmowym intranecie na przykład, komunikacja między zespołami o wiadomościach byłaby znacznie prostsza - możliwe byłoby wówczas przesłanie sobie między programistami linku do właściwej wiadomości „to tej wiadomości szukasz”, włącznie z upewnieniem się, że na pewno wskazana wiadomość nie jest przestarzała - zawierałaby wówczas odnośnik do wiadomości którą obecnie powinno się stosować.

Proces generowania dokumentacji jest analogiczny do znanego z świata Javy narzędzia JavaDoc ¹ - stąd zainspirowana JavaDociem nazwa tego projektu. Sam proces generowania dokumentacji polega na dostarczeniu narzędziu plików *.proto, które następnie są parsowane oraz na podstawie tego procesu, generowana jest strona www zawierające wszystkie zebrane informacje, włącznie z komentarzami oraz dodatkowymi informacjami typu „*deprecated*” (ang. przestarzałe). Jako dodatkowy krok wygenerowana strona mogłaby automatycznie zostać opublikowana w firmowym intranecie.

Cały proces możliwe jest w pełni zintegrować z narzędziami stosowanymi do budowania projektów np. Javowych. W przypadku projektów Javowych, obecnym *de facto* standardem w wielu firmach stał się Apache Maven ². ProtoDoc może zostać użyty razem z Maven aby automatycznie, podczas budowania projektu generować dokumentację. Możliwe jest uruchomienie tego zadania samodzielnie, lub jako jeden z etapów budowy projektu - dzięki czemu nie konieczne jest pamiętanie oraz ręczne aktualizowanie dokumentacji - byłaby automatycznie generowana podczas buildu, na przykład na serwerze ciągłej integracji.

¹JavaDoc - Strona domowa projektu: <http://bit.ly/javadochome>

²Apache Maven - Strona domowa projektu: <http://maven.apache.org>

3. Analiza obecnie dostępnych rozwiązań

Celem ułatwienia zrozumienia poniższego, oraz kolejnych rozdziałów w przypadku gdy czytelnik nie miał jeszcze styczności z Google Protocol Buffers zalecane jest wpierv zapoznanie się z *Dodatkiem A*, gdzie wyjaśniane jest dokładnie jak oraz dlaczego działa ProtoBuf¹.

Niestety na chwilę obecną nie są dostępne narzędzia pozwalające na generowanie dokumentacji z plików Protocol Buffers. *Analiza obecnych rozwiązań zatem organiczy się do rozważenia opłacalności wykorzystania jakiegoś projektu open source jako bazy dla ProtoDoc.*

Jak się okaże, najopłacalniejsza z perspektywy programisty jak i użytkownika końcowego gotowej aplikacji ProtoDoc, będzie implementacja parsera, przy wykorzystaniu języka Scala, a nie wykorzystanie istniejących rozwiązań - które na przykład posiadają bardzo duże zewnętrzne zależności, lub ich dopasowanie do potrzeb tego projektu byłby zbyt dużym przedsięwzięciem.

3.1. Google Protoc

Protoc jest „oryginalnym” kompilatorem plików *.proto. Zawiera ręcznie zaimplementowany przez inżynierów google skaner oraz parser, potrafiący obsłużyć 100% specyfikacji ProtoBuf. Jego źródła są dostępne na stronie Google Code: <http://code.google.com/p/protobuf/source/browse/> Projekt objęty jest licencją *New BSD License*¹.

Warto również uwypuklić pewien problem z udostępnianym przez Google kompilatorem Protocol Buffers IDL - *protoc*. Otóż nawet jeżeli źródłowy plik *.proto posiada komentarze, kompilator *protoc* nie przeniesie je do wynikowych plików, np. *.java. Parser ten niestety ignoruje całkowicie komentarze.

Po wstępnej analizie kodu parsera dostarczanego przez Google doszedłem do wniosku, że niestety wykorzystanie go jako bazy ProtoDoc nie byłoby opłacalne, ze względu na bardzo dużą ilość zmian które trzeba by wprowadzić w *core* parsera - zaimplementowanego „ręcznie”, bez zastosowania znanych generatorów parserów, w C++.

¹New BSD License, znana również jako 2-clause BSD license - <http://www.opensource.org/licenses/bsd-license.php>

3.2. Idea plugin protobuf

Innym projektem open source zawierającym zaimplementowany parser ProtoBuf jest plugin do „IntelliJ IDEA”, popularnego w świecie programistów JVM IDE programistycznego. Źródła znajdują się na Google Code pod adresem: <http://code.google.com/p/idea-plugin-protobuf/source/browse> Projekt udostępniany jest na warunkach *Apache 2.0 License*².

Z perspektywy ProtoDoc, interesującymi fragmentami tego projektu jest skaner oraz parser. Skaner jest generowany przy pomocy *JFlex*³, , odpowiednika narzędzia GNU Flex, dla języka Java. Skaner teoretycznie nadawałby się do ponownego wykorzystania - obsługiwane są tutaj również komentarze.

Niestety druga z interesujących nas części aplikacji, parser, jest *ściśle związany ze środowiskiem IntelliJ IDEA*, dla którego to powstał ten projekt. IntelliJ dostarcza własny mechanizm parsowania do którego pluginy jedynie mogą się podpinąć, oraz pomagać w przeprowadzeniu parsingu pliku, nie można w tym przypadku powiedzieć że projekt zawiera całą implementację parsera. Część źródeł IntelliJ IDEA co prawda jest otwarta, jednak skorzystanie z podejścia dołączenia całego IDE, aby być w stanie parsować pliki, wydaje się bardzo nie optymalna - rozmiar dystrybucji ProtoDoc stałby się bardzo duży (rzędu setek MB, z racji dołączonych zależności w postaci IntelliJ).

Jak widać, również i ten projekt nie dostarcza w pełni funkcjonalnej oraz łatwej to rozbudowania o potrzebne w projekcie ProtoDoc funkcjonalności implementacji parsera Protocol Buffers. W związku z powyższym, postanowiłem wybrać sposób własnoręcznej implementacji parsera, aby proces był jednak możliwie przyjemny, oraz możliwy do utrzymania w przyszłości - na przykład przez społeczność Open Source. Ostatecznie wybrana przezemnie technika implementacji parsera zostanie przedstawiona w kolejnej sekcji.

3.3. Decyzja: własnoręczna implementacji Parsera przy pomocy *Scala Parser Combinators*

Podsumowując, istnieją implementacje parserów Protocol Buffers na wolnych (jak wolność) licencjach, jednak rozbudowa ich o pożądane funkcjonalności, albo byłaby zbyt czasochłonna by nazwać ją opłacalnym (zmiany manualnie implementowanego parsera *protoc*) lub wymagałyby przepisania parsera w całości, w powodu korzystania przez nie z zewnętrznych zależności których nie da się w prosty sposób dostarczyć. Tabela 3.3 przedstawia małe podsumowanie zastosowanych technik implementacji parserów w omawianych projektach.

Po przeanalizowaniu powyższych projektów i porzuceniu pomysłu rozwinięcia istniejącej już implementacji o potrzebne elementy, rozpocząłem wybór generatora parserów / skanerów który chciałbym zastosować podczas tego projektu.

²Apache 2.0 License - <http://www.apache.org/licenses/LICENSE-2.0>

³JFlex (Fast Lexical Analyzer for Java)- Strona domowa projektu: <http://jflex.de/>

Projekt	Metoda impl. skanera	Metoda impl. parsera
Google Protoc	"manualnie", C++	"manualnie", C++
Idea-Plugin-Proto	JFlex, Java	dostarczany z IntelliJ, Java

Tablica 3.1: Zestawienie sposobów implementacji parserów w rozważanych projektach open source

Pierwotnym kandydatem do zastosowania jako generator parsera był powszechnie znany *GNU Bison*⁴, który w połączeniu z Flexem pozwolił na wygenerowanie parsera w „znajomy” sposób. Oba te narzędzia są dobrze znane oraz sprawdzone od wielu lat oraz posiadają dobrą dokumentację. W ramach poszukiwań innych rozwiązań natknąłem się jednak na tak zwane „kombinatory parserów”, a następnie na fakt iż istnieje ich implementacja wewnątrz biblioteki standardowej języka Scala.

Scala jest statycznie typowanym językiem programowania na platformę Java który wspiera zarówno *obiektowy* jak i *funkcyjny* paradygmat programowania. Tak zwane „Parser Combinators” o których tutaj mowa nie są ideą nową. Pojawiły się wraz z językami funkcyjnymi, a pierwsze publikacje naukowe na ich temat można było już napotkać w 1996 roku [GH96] w publikacji Hutton oraz Meijer. Pojęcie „parsowania przy pomocy kombinatorów parserów” najłatwiej jest wytłumaczyć jako:

„Budowanie parserów rekursywnie zstępujących poprzez modelowanie parserów jako funkcji i definiowanie funkcji wyższego rzędu (zwanym kombinatorami) które implementują konstrukcje takie jak sekwencjonowanie, wybór oraz powtórzenie. [...]” [GH96]

Będziemy mieli zatem w efekcie do czynienia a parserem „rekursywnie zstępującym” ($LL(k)$). Przedstawicielem generatorów tworzących tego typu parsery jest na przykład ANTLR⁵, opublikowany po raz pierwszy w roku 1992 jako następcą *Purdue Compiler Construction Tool Set* który powstał jeszcze w roku 1989 (sic). Ten typ parserów dodaje do znanej klasy $LL(k)$ funkcjonalność „wycofania się”, z dowolnej głębokości look-ahead (parser może pracować z dowolnie dużym k , i zawsze będzie w stanie wykonać nawrot oraz wypróbować inną ścieżkę). Korzystanie z nawrotów przez parser oczywiście nieśie z sobą zmniejszenie jego wydajności, jednakw wielu przypadkach (jak choćby Protocol Buffers, które mają stosunkowo prostą gramatykę), obawa przed spadkiem wydajności nie odzwierciedla się zbyt w rzeczywistości. Dobrą wiadomością jest natomiast, że w *Scala Parser Combinators* możemy korzystać z wersji metod z dodanym wykrzyknikiem oznaczającym, że pragniemy aby dany fragment był faktycznie klasy $LL(1)$ - jeżeli gramatyka nie jest na tyle jednoznaczna aby dało się uzyskać $LL(1)$ w danym parserze zostaniemy powiadomieni o tym pod postacią błędu.

Jednym z wyróżniających *Scala Parser Combinators* czynników jest fakt iż zamiast pisać pliki w których deklarujemy naszą gramatykę a sam kod źródłowy parsera jest dopiero generowany na jego podstawie w przypadku Scali i wspomnianej biblioteki zdefiniować gramatykę parsera, w połączeniu z blokami kodu które miałyby dokonać odpowiednich transformacji sparsowanych tokenów dokładnie w

⁴GNU Bison - Strona domowa projektu: <http://www.gnu.org/software/bison>

⁵ANTLR - Strona domowa projektu: <http://www.antlr.org/>

tym samym pliku który jest „plikiem źródłowym parsera”. Dzięki temu oszczędzamy na „kroku” generowania kodu źródłowego parsera, który dopiero później zostałby skompilowany oraz wykonany. Daje to ogromną przewagę podczas poszukiwania błędów w parserze - ponieważ ewentualne problemy bezpośrednio odwołują się do tego co my napisaliśmy, a nie do odrębnego pliku który powstał na podstawie naszego pliku.

Pomimo tak wielu zalet sama struktura definicji parsera pozostaje podobna do znanej z Bisona oraz nadal jest złudnie podobna do notacji *BNF*⁶ - która jest bardzo przejrzysta oraz zazwyczaj znana, lub łatwa to utworzenia podczas pisania parsera znanego języka.

Kolejną zaletą jest umieszczenie definicji tokenów w tym samym pliku co definicja skanera - ponownie unikamy kroku generowania skanera (na przykład przy użyciu *JFlex*). Zmniejszenie ilości miejsc gdzie konieczne jest wprowadzenie modyfikacji, jest zatem kolejną z zalet tego podejścia.

Metoda impl. skanera	Metoda impl. parsera
Parser Combinators, Scala	

Tablica 3.2: Przedstawienie jednolitości rozwiązania z zastosowaniem *Scala Parser Combinators*

Gdyby umieścić Parser Combinators (tak jak przedstawiono w Tabeli 3.3) na przedstawionej powyżej tabeli, z zestawieniem jak implementowana jest która część parsera, okazałoby się że jesteśmy wyjątkowo spójni - wszystkie części implementowane są w jednym miejscu / języku / narzędziem.

Reasumując poniższe zalety są przyczyną wyboru tego podejścia do generowania parsera ponad klasyczne narzędzia typu Flex/Bison:

- Brak konieczności dodatkowego kroku generowania kodu źródłowego
 - dla skanera (brak osobnego pliku ze spisem tokenów)
 - dla parsera (brak osobnego języka, dedykowanego definiowaniu)
- Wykonywany kod jest bezpośrednio związany z pisaną przez nas definicją parsera, co pozwala na łatwe poszukiwanie błędów
- Minimalizacja miejsc w których konieczne jest wprowadzanie zmian, cała implementacja znajduje się w 1 miejscu

Opis działania *Parser Combinators* znajduje się w kolejnej sekcji, oraz w *Dodatku B*, gdzie wytłumaczone zostały wszystkie zasady konstruowania parserów przy pomocy tej biblioteki.

Jeżeli czytelnik jeszcze nie miał styczności z Scalą oraz dostarczaną przez wraz z nią biblioteką *Parser Combinators* zalecane jest zapoznanie się z *Dodatkiem B*, gdzie szczegółowo omówiono zasady działania samego języka jak i Parser Combinators.

⁶Notacja BNF - „Backus Naur Form”, metoda zapisu reguł gramatyki kontekstowej

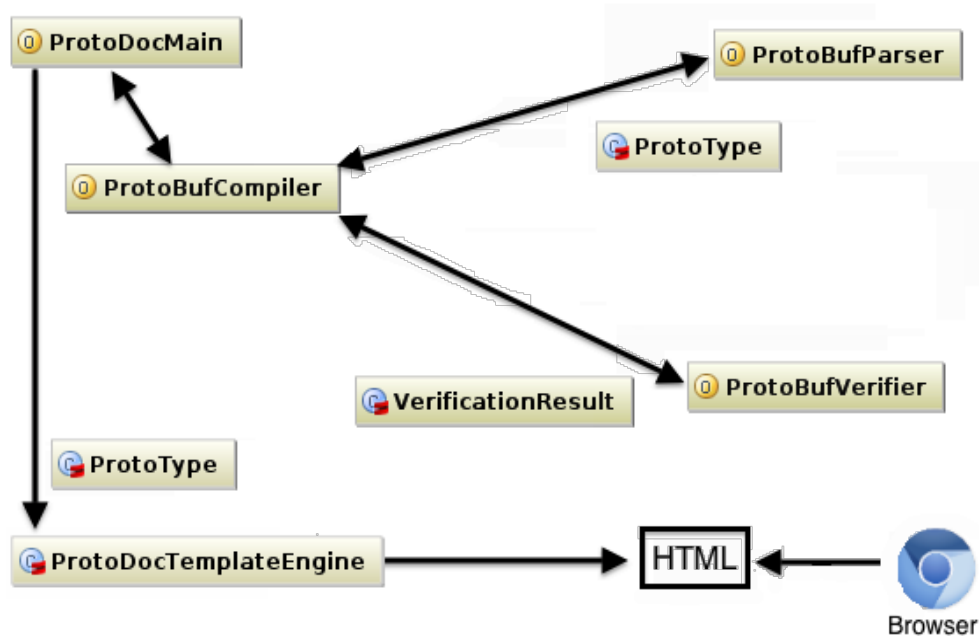
4. Projekt systemu

Rozdział ten ma za cel przedstawienie w sposób holistyczny architektury aplikacji *ProtoDoc*. Dopiero w kolejnym rozdziale (Rozdział 5) zostaną opisane szczegóły implementacyjne, oraz podjęte decyzje na poziomie kodu, na razie zostanie opisana jestnie generalna architektura oraz podjęte w tej warstwie abstrakcji decyzje.

Główne odpowiedzialności aplikacji zostały podzielone pomiędzy poniższe klasy (konkretniej, klasy typu **object** - omówionych dokładniej w Dodatku B, dotyczącym języka Scala):

- *ProtoDocCompiler* - jest fasadą nad *ProtoBufParser* oraz *ProtoBufVerifier*. Został wprowadzony celem ułatwienia pracy na „zweryfikowanych” już obiektach typu *ProtoType*, będącymi wynikami parsowania, jednak dopiero po wykonaniu weryfikacji, możemy być pewni że utworzone obiekty z pewnością są poprawne. Compiler enkapsuluje parsowanie oraz weryfikację w jedno publiczne API, znacznie upraszczając testowanie oraz korzystanie z *ProtoDoc* na poziomie kodu (a nie zewnętrznej aplikacji).
- *ProtoBufParser* - jest implementacją parsera przy pomocy *Scala Parser Combinators*. Odpowiedzialny jest również za analizę syntaktyczną - błędy odnalezione podczas parsowania (na przykład „za duży” **tag** dla pola wiadomości) zostałyby wykryty już podczas parsowania. Miłym akcentem jest tutaj iż parser jest w stanie podać kontekst w którym wystąpił błąd, zaznaczając go znakiem ^, co znacznie uprzyjemnia korzystanie z niego użytkownikom końcowym. Omówiony zostanie szczegółowo w sekcji 5.1.
- *ProtoBufVerifier* - „weryfikator” zajmujący się sprawdzaniem poprawności semantycznej sparsowanych plików *.proto. W przypadku napotkania błędów krytycznych, działanie protodoc może zostać w tym miejscu przerwane, oraz zwrócony zostałby komunikat informujący o przyczynie błędu.
- *ProtoDocTemplateEngine* - „generator kodu”, w naszym przypadku generowanym „kodem” jest HTML. *TemplateEngine* wykorzystuje wewnątrz silnik renderowania szablonów *Mustache*, który zostanie omówiony w sekcji 5.3.1. Mechanizm działania generatora jest stosunkowo prosty oraz sprowadza się do podstawiania odpowiednich wartości w odpowiednie „zmienne” celem udostępnienia ich do wypisania przez system szablonów *Mustache*.

W kolejnych sekcjach tego rozdziału zostaną przedstawione w formalny sposób interakcje oraz relacje między komponentami systemu. Proszone zostaną również typy którymi zamodelowano w języku Scala odpowiednie typy pól mogące pojawić się w wiadomości ProtoBuf. Wpierw zostanie jednak pokazany w sposób bardziej wizualny niż formalny, jak komponenty się ze sobą komunikują - służy temu Rysunek 4.



Rysunek 4.1: Nie formalna wizualizacja interakcji pomiędzy komponentami

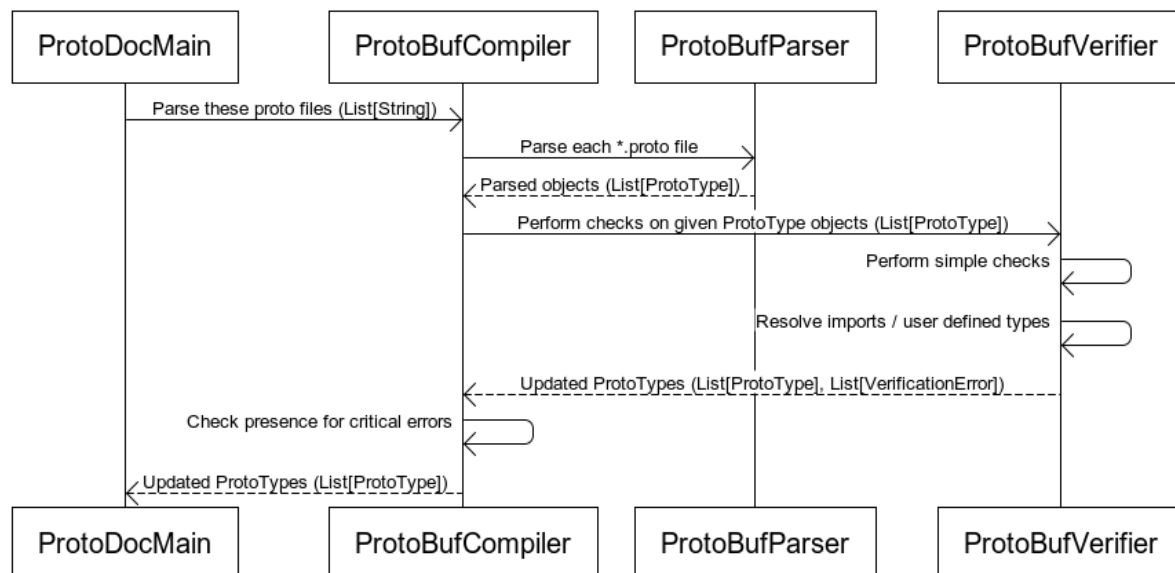
Dzięki nie formalnej wizualizacji interakcji między komponentami na Rysunku 4 można łatwo zobrazować sobie jak komponenty między sobą wymieniają informacje. Pliki *.proto są parsowane przez parser, po czym reszta komunikacji odbywa się na poziomie klas typu `ProtoType`, oraz jej specjalizacji.

Jak widać mamy wydzielone klasyczne dla kompilatorów odpowiedzialności: parsowanie/skanowanie, weryfikację semantyczną oraz generowanie kodu. W kolejnych sekcjach zostaną przedstawione diagramy oraz wizualizacje interakcji między tymi komponentami.

4.1. Architektura aplikacji

W tej sekcji przedstawione zostaną diagramy sekwencji opisujące interakcje między wcześniej już wstępnie opisanymi komponentami.

Na diagramie sekwencji ?? zostało przedstawione szczegółowo do jakich interakcji między wcześniej opisywanymi (nie formalnie) komponentami dochodzi podczas procesu parsowania.



Rysunek 4.2: Diagram sekwencji parsowania oraz weryfikowania wiadomości

Punktem wejściowym aplikacji jest klasa `ProtoDocMain`, po sparsowaniu argumentów wejściowych specjalnym Domain Specific Language (innym niż Parser Combinators, zostanie on omówiony i przedstawiony w sekcji poświęconej tej klasie), przekazuje przygotowane do parsowania zawartości plików do obiektu `ProtoBufCompiler`. Compiler jedynie deleguje parsowanie każdego z plików do `ProtoBufParser`, który wykonuje parsowanie przy pomocy *Scala Parser Combinators*.

Ponieważ jeden plik `*.proto` może zawierać więcej niż jedną wiadomość (lub enumerację), dla każdego sparsowanego pliku proto zwracana jest lista typów konkretnych (*omówionych szczegółowo w sekcji 4.1.1*), dziedziczących po abstrakcyjnej klasie `ProtoType`.

Kolejnym krokiem jest przekazanie otrzymanych właśnie `ProtoType` do Instancji `ProtoBufVerifier`, który zajmuje się weryfikacją poprawności typów. Może on, ponieważ otrzymuje *wszystkie* sparsowane typy, również rozstrzygnąć czy *import* z innego pakietu jest poprawny czy też nie. Innymi słowy, jedną z jego odpowiedzialności jest rozwiązanie problemów widoczności typów. Nie mógł, oraz nie powinien, przeprowadzać tego Parser, ponieważ nie posiadał jeszcze pełnego zestawu danych (typów) koniecznych do rezolucji typów. Sposób w jaki informacja o „nie rozstrzygniętym typie” jest przekazywana do weryfikatora, jest dość prosty: Parser podczas napotkania typu którego jeszcze nie zna, tworzy taki sam obiekt referencji do pola, jaki stworzyłby gdyby znał ten

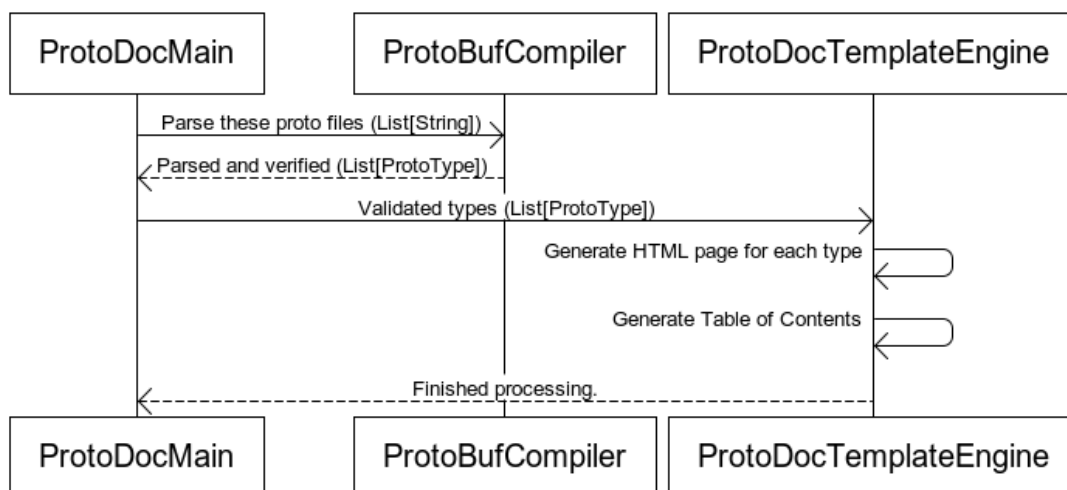
typ, jednak dodatkowo zaznacza iż typ nie był widoczny podczas parsowania. Gdy weryfikator dostaje wszystkie typy oraz ich pola, przeszukuje pola, zważywszy na wspomnianą flagę - oraz próbuje dokonać rezolucji typu, mając już pełne informacje o dostępnych typach w danym kontekście. Przyjęta przez niego strategia zostanie przybliżona dokładniej w sekcji ??.

`ProtoBufVerifier` odpowiada listą komunikatów mogących być albo błędami, albo ostrzeżeniami, a `ProtoBufCompiler` może na nie zareagować przerwaniem wykonania aplikacji oraz wypisaniem wszystkich problemów, lub może zwrócić wszystkie zweryfikowane typy (obiekty typu `ProtoType`) do głównej klasy programu, która pierwotnie rozpoczęła proces parsowania plików proto — do `ProtoDocMain`.

Kolejnym krokiem w kierunku wytworzenia wyniku działania aplikacji — gotowej strony www dokumentacji `ProtoDoc` — jest przekazanie zweryfikowanych obiektów `ProtoType` z `ProtoDocMain` do ostatniego z istotnych komponentów aplikacji — do generatora kodu, tj. do instancji klasy `ProtoTemplateEngine`. Warty zauważenia jest powrót to nazewnictwa *ProtoDoc*... tej klasy, w przeciwieństwie do *ProtoBuf*... występującego jako prefix poprzednich komponentów. Przyczyną takiego rozróżnienia w nazewnictwie jest, iż zarówno parser jak i weryfikator, nie są ściśle związane z `ProtoDoc` - mogą parsować oraz weryfikować dowolne pliki proto, oraz zwracają ich obiektową reprezentację. Równie dobrze komponenty te można by wykorzystać w innych celach - nie tylko celem generowania dokumentacji.

Proces generowania kodu jest bardzo prosty. Jako, że `ProtoBufCompiler` dostarczył już „gotowe” obiekty, które na pewno są poprawne, jedyne co pozostaje generatorowi do zrobienia to wygenerowanie strony HTML na podstawie każdego z obiektów proto-typów, które zostały mu przekazane przez `ProtoDocMain`.

Diagramie sekwencji 4.1 obrazuje proces generowania kodu. Pierwsze dwie wymiany wiadomości zostały szczegółowo przedstawione na poprzednim diagramie (4.1).



Rysunek 4.3: Diagram sekwencji generowania stron HTML dokumentacji

4.1.1. Zastosowany model typów wiadomości Protocol Buffers w Scala

Celem wykorzystania w pełni z potencjału statycznego typowania języka Scala — zamiast pracowania wprost na listach i mapach wartości które w domyślnym przypadku zostałyby zwrócone przez parser, konieczne było stworzenie modelu typów, odzwierciedlającego w Scali typy mogące pojawić się w ProtoBuf. Konieczne było zamodelowanie zarówno istniejącej wewnętrznie w Protocol Buffers jak i łatwej do rozszerzania o typy definiowanie przez użytkownika struktury typów.

Zdefiniowanie nowego typu przez użytkownika jest główną czynnością podczas opisywania interfejsu przy pomocy Google Protocol Buffers, także typy te powinny również mieć istotne odzwierciedlenie w ProtoDoc. W ramach przypomnienia, Listing 4.1 przedstawia składnię zdefiniowania prostej wiadomości (*message*) lub enumeracji (*enum*) - jedynych interesujących nas w zakresie ProtoDoc typów.

Listing 4.1: Przykład zdefiniowania nowych typów w Protocol Buffers IDL

```
package pl.project13;

/** some comments */
message MyNewType { required string pole = 1; }

/** some comments */
enum MyEnumeration { VALUE = 1; }
```

Okazuje się, że enumeracja i wiadomość dzielą wiele wspólnych cech, oba typy:

- mogą znajdować się w *package*
- mają nazwę
- mają „w pełni kwalifikowaną nazwę”, składającą się z połączenia nazwy typu
- zawierają pola. Mimo, że w przypadku enumeracji pole wygląda nieco inaczej, można przyjąć pewne uogólnienie i traktować je tak samo jak pozostałe pola.

W związku zauważeniem powyższych wspólnych cech został wprowadzony wprowadzony typ *ProtoType*, będący super-typem dla obu tych *klass*.

Dodatkowo warto zauważyć, że komentarze mogą być umieszczone „na” każdym z wspomnianych typów, oraz to samo można powiedzieć o polach umieszczonych wewnątrz tych typów - jak obrazuje to Listing 4.2.

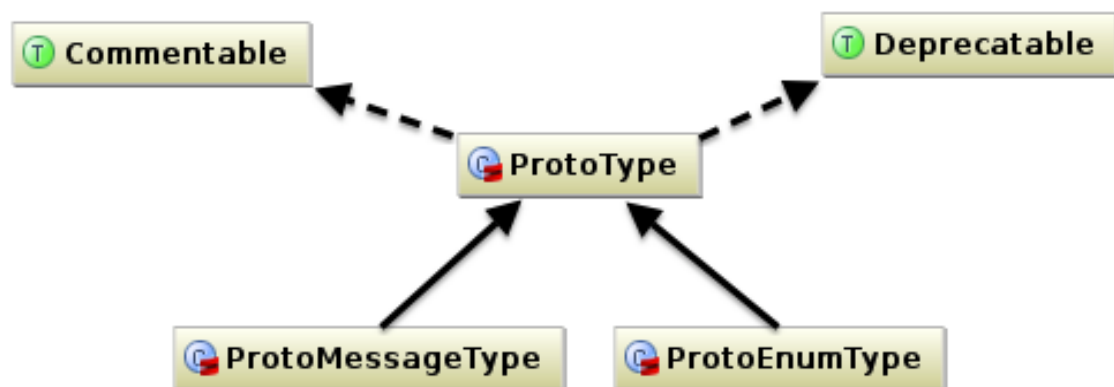
Listing 4.2: Przykład umieszczenia komentarza ProtoDoc na polach wiadomości oraz enumeracji

```
message MyNewType { /** docs */ required string field = 1; }

enum MyNewType { /** docs */ SOME_VALUE = 1; }
```

Umieszczenie pola *comment* wewnątrz klasy `ProtoType` nie byłoby zatem dobrym pomysłem, ze względu na nie-możliwość ponownego użycia interfejsu typu „X ma komentarz” na polach. Rozwiązaniem jest wprowadzenie interfejsu „*Commentable*”, które eksponuje metody związane z posiadaniem komentarza, na przykład „czy komentarz jest obecny?” etc. Pozwoliło to na implementację metod pracujących na tym interfejsie, nie ważne czy ów komentowalny element jest polem czy nowym typem, zatem zmniejszyło ścisłość wiązań w API tworzonego systemu.

W przypadku języka Scala, mowa tutaj nie o interfejsach a *Trait*ach (rozpoznawalnych na załączonych diagramach po zielonej ikonie T), jednak jako że celem tego rozdziału pracy nie przedstawienie implementacji a generalnego design projektu można przyjąć że Trait może być rozumiany jako „interfejs”. Szczegółowy opis czym są *Traits* oraz jak wpływają na design projektu, można przeczytać w Sekcji B.3, w Dodatku B.



Rysunek 4.4: Zastosowany model typów definiowanych przez użytkownika typów

Rysunek 4.1.1 przedstawia zastosowaną w projekcie strukturę typów. Warto zauważyć dodatkowy Trait o nazwie *Deprecatable*, oznaczający „coś co może zostać adnotowane jako przestarzałe”. Adnotowanie pól jako `@deprecated` jest dobrą praktyką programistyczną oraz bezcennym źródłem dodatkowej informacji o typie dla takiego narzędzia jak *ProtoDoc*, które może takie pola wyszarzyć lub zasugerować jakiego typu powinno się użyć zamiast oznaczonego taką adnotacją. Przyczyną wprowadzenia tego typu jako Trait ponownie jest fakt, że wiele elementów protocol buffers może zostać oznaczona jako przestarzała, nie jedynie same deklaracje typów.

4.1.2. Zastosowany model typów pól Protocol Buffers w Scala

Analogiczny jak przedstawiony powyżej problem dotyczy pól, które mogą być reprezentować jeden z predefiniowanych typów protocol buffers, lub mogą być enumeracją lub wiadomością definiowaną przez użytkownika.

Celem skorzystania również i tutaj z statycznego typowania również podczas pracy na polach, również tutaj konieczne było wprowadzenie modelu typów. Tutaj wymaganiem ponownie było aby możliwie prosto dało się wspierać już wbudowane w ProtoBuf jak i nowo definiowane przez użytkownika typy.

Listing 4.3 przedstawia różne możliwości jakie mamy do dyspozycji podczas definiowania pola w wiadomości lub enumeracji. W przypadku gdy czytelnik chciałby się zagłębić w szczegóły składni oraz znaczenia poszczególnych deklaracji, zachęcam do przeczytania Dodatku A, w którym wszystkie te elementy są szczegółowo omawiane. Przykład tutaj przytaczany jest celem unaocznienia ponownie cech wspólnych dla wspomnianych elementów.

Listing 4.3: Deklaracja pola w wiadomości oraz enumeracji

```
// message fields:
optional int32 age = 1;
required string age = 2 [default = "Konrad"];
repeated sint64 numbers = 3;

optional MyType it = 4;
optional MyEnum other = 5 [default = VALUE];

message MyType { }
enum MyEnum {
  // enum values:
  VALUE = 3;
}
```

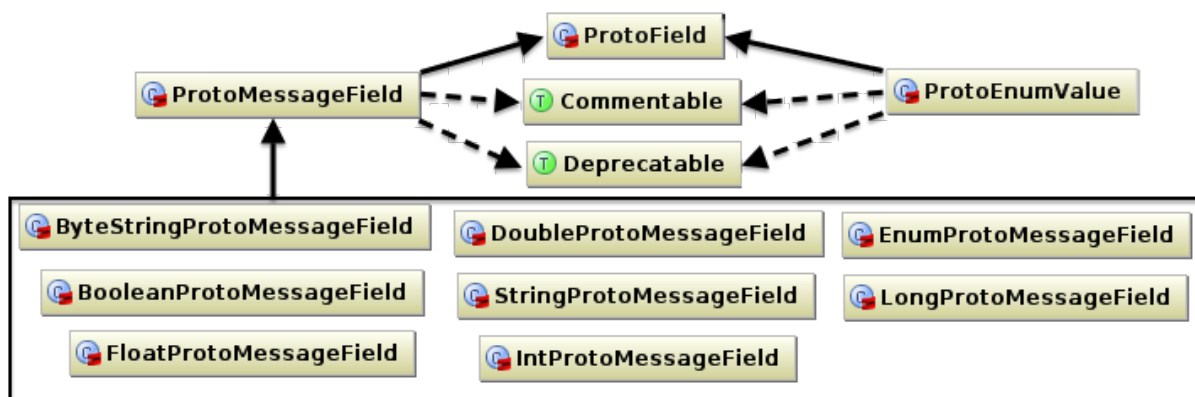
Deklarowanie pól w wiadomościach jest oczywiście bardziej interesujące niż pola w enumeracjach, co powyższy przykład powinien dość ładnie obrazować. Mimo wszystko, ponownie jesteśmy w stanie znaleźć pewne wspólne elementy pomiędzy wszelkimi „polami”, włączając w to również wartości enumeracji. Przy okazji, pamiętajmy iż każde z tych pól może być *Deprecated* oraz może zostać okomentowane komentarzem ProtoDoc.

- pole posiada nazwę
- pole posiada przypisany mu *tag* (liczba umieszczona po znaku równości, po szczegółowe wyjaśnienie czym tag jest, odsyłam do Dodatku A)
- pole może posiadać komentarz
- pole może być oznaczone jako przestarzałe (*Deprecated*)

Powyższa lista elementów wspólnych w sposób oczywisty doprowadziła do powstania klasy `ProtoField` będącej podstawą ku wszystkim pozostałym typom. Dzięki wydzieleniu Traitów `Commentable` oraz `Deprecatable`, również teraz można je zastosować aby osiągnąć te same funkcjonalności w nowo przedstawionych klasach.

Pole będące wartością enumeracji będziemy traktować analogicznie jak zwyczajny `ProtoField`, nie jest ono wyjątkowe z perspektywy struktury samej z siebie. Co je odróżnia

od `ProtoMessageField` jest natomiast fakt iż po `ProtoMessageField` dziedziczą wszystkie predefiniowane typy. Dokładne oddanie relacji pomiędzy utworzonymi tutaj typami a typami umieszczanymi w plikach proto nie jest w naszym przypadku konieczne, a być może nawet nie wskazane. W przypadku korzystania z protocol buffers na platformie JVM, nie istnieją odpowiedniki typów `unsigned` / `signed` - także korzystamy w tym przypadku po prostu z typu `scala.Int`, do przechowania np. wartości domyślnej takiego pola. Nazwa typu którego dana klasa jest mapowaniem zostaje jednak utrzymana w ciele klasy, abyśmy mogli podczas generowania dokumentacji przywołać jaki dane pole miało typ w pliku *.proto. Pełen spis przyjętych mapowań został przedstawiony w formie Tabeli 4.1.2.



Rysunek 4.5: Zastosowany model typów pól mogących wystąpić w ProtoBuf IDL

W ramach ciekawostki, chciałbym w tym miejscu przedstawić implementację dwóch przykładowych z omawianych klas.

Listing 4.4: Pełna implementacja klasy `ProtoEnumValue`

```

case class ProtoEnumValue(valueName: String, tag: ProtoTag)
  extends ProtoField
  with Commentable
  with Deprecatable

```

Listing 4.5: Pełna implementacja klasy `StringProtoMessageField`

```

case class StringProtoMessageField(override val fieldName: String,
  override val tag: ProtoTag,
  override val modifier: ProtoModifier,
  override val defaultValue: String = None)
  extends ProtoMessageField(fieldName,
    "string", // type name in *.proto
    "java.lang.String", // JVM representation
    tag,
    modifier,
    defaultValue)

```

Typ pola w Protocol Buffers	Odpowiednik w ProtoDoc
double	DoubleProtoMessageField
float	FloatProtoMessageField
int32	IntProtoMessageField
int64	LongProtoMessageField
uint32	IntProtoMessageField
uint64	LongProtoMessageField
sint32	IntProtoMessageField
sint64	LongProtoMessageField
fixed32	IntProtoMessageField
fixed64	LongProtoMessageField
sfixed32	IntProtoMessageField
sfixed64	LongProtoMessageField
bool	BooleanProtoMessageField
string	StringProtoMessageField
bytes	ByteStringProtoMessageField
enumeration type	EnumProtoMessageField(type)
user defined message type	ProtoMessageField(type)
enum value	ProtoEnumValue

Tablica 4.1: Typy pól oraz odpowiadające im typy Scala (bazując na reprezentacji typów przez protoc na JVM)

Warto zwrócić uwagę, że mimo iż klasy przedstawiona na Listingach 4.4 oraz 4.5 nie posiadają stricte zdefiniowanego ciała, dzięki pewnym konwencjom oraz mechanizmom udostępnianym przez Scala, takie definicje są wszystkim co jest potrzebne do zdefiniowania w pełni funkcjonalnych klas, włącznie z polami, akcesorami dla atrybutów etc.

Czytelnika zainteresowanego mechanizmami kryjącymi się za zastosowanymi tutaj „case classami” zachęcam do przeczytania sekcji ??, dotyczącej tego działu języka Scala, z Dodatku B.

5. Szczegóły implementacyjne

Przed rozpoczęciem lektury poniższego rozdziału zalecane jest zapoznanie się z podstawami języków Scala jak i narzędzia/języka Protocol Buffers. W sekcji tematów zostały przedstawione w ramach dodatków

- *Dodatek A* — omawiający Protocol Buffers. Gdzie i dlaczego powinno się je stosować, oraz przytoczenie benchmarku potwierdzającego te tezy. Zostanie też przedstawiona składnia ProtoBuf IDL oraz workflow stosowany podczas pracy z ProtoBuf.
- *Dodatek B* — omawiający podstawy języka *Scala*, po których przeczytaniu czytelnik będzie w stanie czytać ze zrozumieniem kod projektu będącego przedmiotem tego projektu. Przedstawione zostaną również szczegóły Parser Combinators, oraz jak je czytać / pisać w języku Scala.

W tym rozdziale omówione zostaną zarówno ogólne założenia przyjęte podczas projektowania systemu, jak i struktura klas przyjęta celem modelowania struktury typów Protocol Buffers. Następnie przedstawione zostaną poszczególne komponenty aplikacji, z naciskiem na uzasadnienie wybranych rozwiązań oraz rozważenia ich zalet, wad oraz potencjalnych możliwości usunięcia zauważonych wad.

Przedstawione zostaną również elementy kodów źródłowych, skrócone do postaci wystarczającej na cel omówienia danego tematu - w przypadku chęci zapoznania się ze całością implementacji np. komponentu parsera, zachęcam do zapoznania się z załączonymi do pracy plikami źródłowymi projektu.

W ostatniej sekcji (5.4) zostanie przedstawione jak należy przygotować środowisko pracy celem kompilowania oraz budowania projektu ze źródeł.

5.1. ProtoBufParser

5.1.1. Wprowadzenie do kombinatorów parserów

TODO klasyfikacja, opisać że są lewo stronnie rekurencyjne etc.

http://en.wikipedia.org/wiki/Recursive_descent_parser

http://en.wikipedia.org/wiki/Left_recursion <http://en.wikipedia.org/wiki/Parser>

<http://stackoverflow.com/questions/17840/how-can-i-learn-about-parser-combinators>

5.1.2. Fragmenty implementacji

Przedstawić fragmenty parsera. Najlepiej go dać jako dodatek jednak w całości.

```
def messageTypeDef: Parser[ProtoMessageType] = opt(comment)
    ~ messageNameDef
    ~ messageBody ^^ {
case maybeDoc ~ id ~ allFields =>
    // utworzenie instancji ProtoMessageType
}
```

5.2. Verifier

Generalny opis dlaczego musiał powstać

Przedstawić jakie sprawdzania obsługuje. Pokazać że obsługuje importy oraz udowodnić dlaczego konieczny jest dodatkowy krok na to. No bo bez takiego kroku nie wiedziałbym czy przypadkiem gdzieś indziej nie został zdefiniowany jakiś message etc.

5.2.1. Obsługiwane weryfikacje

Może sub sectiony o tych checkach oraz konkretne przykłady błędów i jak są komunikowane?

5.3. CodeGenerator

Generator kodu w tym przypadku jest bardzo prostą serią transformacji. Opisać, wspomnieć że korzystam z mustache etc.

5.3.1. Język szablonów - Mustache

Podczas generowania plików HTML klasa ProtoDocTemplateEngine korzysta z biblioteki *Scalate*¹

¹Scalate - strona domowa projektu: <http://scalate.fusesource.org/>

5.4. Przygotowanie środowiska do rozwoju ProtoDoc

W rozdziale ten pokrótce przedstawię jak należy przygotować środowisko programistyczne celem rozwijania narzędzia *ProtoDoc*, może się to okazać przydatne w przypadku chęci sprawdzenia testów jednostkowych bądź wprowadzenia nowych funkcjonalności do aplikacji.

5.4.1.

5.5. Instalacja narzędzia SBT

ProtoDoc budowany oraz testowany jest przy wykorzystaniu najpopularniejszego obecnie narzędzia do zarządzania buildem w świecie programistów Scala: Simple Build Tool, w skrócie zwanym SBT ².

²SBT - Strona domowa projektu: <https://github.com/harrah/xsbt>

6. Zrzuty ekranu wygenerowanej dokumentacji

Rozdział ten zawiera przykładowe zrzuty ekranu wygenerowanych przy pomocy *ProtoDoc* dokumentacji dla różnych typów wiadomości.

Na Rysunku 6.1 przedstawiona została strona wygenerowana na podstawie

ProtoDoc

by Konrad Malawski
konrad.malawski@java.pl

Table of Contents

Messages:

- [pl.project13.AmazingMessage](#)
- [pl.project13.AmazingMessage.E](#)
- [pl.project13.AmazingMessage.Ir](#)
- [pl.project13.AmazingMessage.S](#)
- [pl.project13.MessageWithInner](#)
- [pl.project13.MessageWithInner.I](#)
- [pl.project13.TopLevel](#)
- [pl.project13.TopLevel.MiddleLeve](#)
- [pl.project13.TopLevel.MiddleLeve](#)
- [pl.project13.WithEnum](#)
- [pl.project13.WithEnum.Message](#)

ProtoDoc (by Konrad Malawski) is Free Software,
licensed under the Apache2 License.
Want the sources? [Fork protodoc.github](#).

M

MessageWithInner

This is a simple Message which has some Inner Message defined Also note that it has a default value on the name property

defined in package: pl.project13

Fields:

name

This can be a name of your liking the default value is lorem ipsum etc

Default value:
loremipsum

Modifier:
required

Tag:
2

Defined as:
string

Mapped to:
java.lang.String

number

A number is just a simple property

Modifier:
required

Tag:
1

Defined as:
int32

Mapped to:
scala.Int

Inner Enums:

This message defines no enums.

Inner messages:

[InnerMessage](#)

Rysunek 6.1: Widok wygenerowanej strony dla typu **message**

E

EnumType

ProtoDoc on GitHub

Nam iaculis sapien id est elementum pulvinar pharetra libero faucibus Duis risus leo fermentum in tincidunt a elementum id tellus Sed sodales eleifend nisi et cursus elit aliquet sit amet Vestibulum rhoncus nulla eu nunc faucibus commodo Integer malesuada facilisis dui sed posuere Maecenas commodo nulla et metus facilisis mattis a ut libero

defined in message: [pl.project13.AmazingMessage](#)

Values:

SMS

Tag
1

PHONE

Tag
2

Rysunek 6.2: Widok wygenerowanej strony dla typu **enum**

7. Rola Testów oraz TDD w procesie tworzenia aplikacji

Projekt prowadzony był zgodnie z zasadami Test Driven Development (zwanego dalej *TDD*), co znacznie ułatwiło ustabilizowanie API oraz głównych konceptów jeszcze we wczesnych etapach tworzenia aplikacji. Ponadto, metodyka ta umożliwiła pracę z dotychczas nieznanym mi API bez obaw o zniszczenie zaimplementowanych wcześniej funkcjonalności.

Metodykę *TDD* możnaby opisać jako cykl składający się z trzech faz:

- napisanie najpierw (sic!) testu, sprawdzającego automatycznie czy stawiane przed nami oczekiwania zostało spełnione

 pewną sub-fazą jest upewnienie się że test faktycznie na stan obecny aplikacji nie przechodzi. Najlepiej aby wiadomość niepowodzenia jasno wskazywała na to co jest przyczyną problemu. Jest to istotne nie tyle teraz, podczas implementacji, jednak podczas dalszego rozwoju aplikacji, kiedy to być może sprawimy, że ten test przestanie przechodzić - wówczas, „*kilka tygodni później*”, pomocny komunikat o przyczynie problemu znacznie przyspieszy zlokalizowanie oraz naprawienie problemu.

- implementacji funkcjonalności, tak aby warunki w teście zostały spełnione.

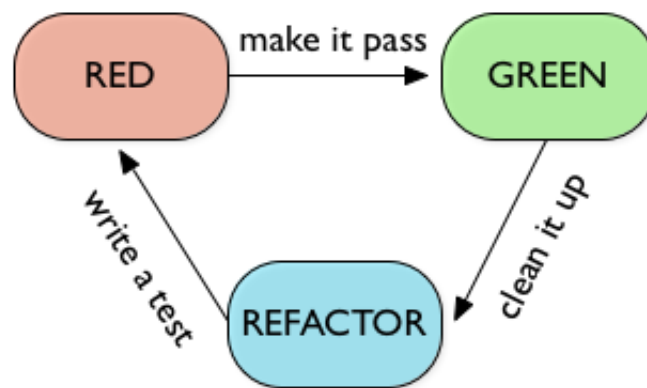
 należy pamiętać aby była to implementacja minimalna - nie wolno wychodzić „do przodu” z implementacją, nawet jeżeli uważa się, że pewna funkcjonalność *prawdopodobnie* będzie niebawem implementowana.

- oraz refaktoringu właśnie zaimplementowanych komponentów aplikacji, lub zauważonych podczas implementacji ewentualnych powtórzeń kodu itp.

Fazy te w literaturze znane są jako „Red - Green - Refactor”, i obrazuje się ją przy pomocy przedstawionego na Rysunku 7 grafu.

Przedstawiony powyżej cykl zazwyczaj trwa pomiędzy kilkoma a trzydziestoma minutami. Technika ta jest ściśle związana z samo-dyscypliną programisty i stosunkowo trudna do zastosowania w przypadku nie stosowania jej na codzień - jednak rezultaty, pod postacią wzrostu jakości kodu oraz zmniejszeniu czasu traconego na poszukiwania błędów są znaczne.

Oprócz pisania testu zanim powstanie jakkolwiek implementacja, bardzo ważnym elementem fazy implementacji jest aby jej celem było napisanie *minimalnej ilości kodu doprowadzając test to „przejścia”* (spełniania wymagań w nim stawianych). Przykładowo, nie dozwolone jest implementowanie dodatkowych funkcjonalności („na zapas”), nawet jeżeli uważa się iż będą niebawem konieczne podczas fazy implementacji związanej z właśnie napisanym testem. Faza implementacji nie może zostać zakończona w przypadku uszkodzenia (sprawienia że inny niż obecnie rozwijany test „nie przejdzie”)



Rysunek 7.1: Schemat obrazujący fazy pracy w metodyce *TDD* (źródło: własne)

7.1. Zaimplementowane specyfikacje

MustacheFilenameTest:

- Should create mustache template filenames

TagVerifierTest:

validateTags

- should detect duplicated tags
 - + Given an message with duplicated field tags
 - + When tags are validated
 - + Then it should detect duplicates
 - + And errors are about the 'second' and 'fail' fields

validateTags

- should should 'OK' a valid tags list
 - + Given a valid message
 - + When tags are validated
 - + Then it have not detected any problems

InnerMessagesTest:

Inner message

- should be parsed properly

PackageTest:

Package name

- should be read from proto file with it

InnerInnerMsg package

- should contain it's super Messages in package name

MultipleProtoFilesTest:

Parser given multiple files

- should parse multiple separate (independent) files

CommentsTest:

Comment on top level message

- should be parsed properly

Comment on field

- should be parsed properly
- should be parsed properly, even if inline
- should be parsed properly, using JavaDoc style markers
- should be parsed properly, even if spanning multiple lines

Multi Line Comment on top level message

- should be parsed properly

Multi Line Comment on inner enum

- should be parsed properly

Comment on enum value

- should be parsed properly

ProtoBufVerifierTest:

The Verifier should validate field types

- should detect an unresolvable field
 - + Given a message with an invalid fieldtype
 - + When the message is parsed and verified
 - + Then the Verifier report it as invalid
 - + And it should point out that the UnknownType is unresolvable
- should have no problems with resolvable field Type
 - + Given a message with valid, resolvable fieldtype, defined before the message
 - + When the message is parsed and verified
 - + Then the result should contain one HasResolvableField message
 - + And the field should be resolved to the proper type

RealSimpleParsingTest:

Parsing of an real message, with outer enum

- should be parsed properly
 - + Given A real proto file
 - + When it is parsed
 - + And it is verified
 - + Then parsed size should be 2

- + And the inner message should be detected
- + And the inner message should be named properly
- + And the enum field should have the proper type resolved
- + And it's tag should be equal 3
- + And it's resolved type should be the outer enumeration
- + And the outer enum should be parsed and named properly

ProtoBufParserTest:

Parser

- should parse single simple message
- should parse single message with enum
- should have no problems with field modifiers

addOuterMessageInfo

- should fix package info of inner enums/messages
 - + Will fix packages of: List(ProtoMessageType [InnerMessage] in package: [])
 - + Fix resulted in: List(ProtoMessageType [InnerMessage] in package: [pl.proj])

DeprecationTest:

Message with deprecations

- the deprecated fields should be detected
- should not detect deprecations where there are none
- should detect deprecation on message type
- should detect deprecation on enum type
- should detect inner types

EnumsTest:

Enum

- should be parseable inside of an Message
- should be usable as field type
- should be usable even before it's type declaration
- should detect an unresolvable enum or message type reference

Undefined enum

- should be type checked, so an not existing enum type used as field type will

MultipleMessagesInOneFileTest:

Parser

- should deal with multiple messages defined in the root level of one file
 - + Given a proto file with two root level messages
 - + When the messages are parsed and verified
 - + Then the result should contain two messages

- should deal with multiple enums and messages defined in root scope, in one f
 - + Given a proto file with two root level messages
 - + When the messages and enum are parsed and verified
 - + Then the result should contain two messages and one enum

FieldsTest:

Message with 2 fields

- should in fact have 2 fields

Parser

- should parse single int32 field
- should parse single fixed32 field
- should parse single sfixed64 field
- should parse single int64 field
- should parse single fixed64 field
- should parse single optional string field
- should parse single required string field with default value

MessageTemplateTest:

ProtoDocTemplateEngine

- should render simple message page

TableOfContentsTest:

ProtoDocTemplateEngine

- should render table of contents from sample data

FullIntegrationTest:

ProtoDoc

- should not fail for simple proto files
 - + Given the simple/ director, with proto files
 - + And a valid destination directory
 - + When the files are parsed
 - + Then no exception should be thrown
 - + And the output should be a valid doc

Passed: : Total 45, Failed 0, Errors 0, Passed 45, Skipped 0

8. Zastosowanie ProtoDoc do automatyzacji dokumentacji projektów

9. Spis obsługiwanych funkcjonalności

Podajemy takie wejście:

```
message Person {  
  required int32 id = 1;  
  required string name = 2;  
  optional string email = 3;  
}
```

Następnie wykonanie:

A ostatecznie otrzymujemy taką stronę: <http://protodoc.project13.pl/sample>.

A. Google Protocol Buffers

W tym dodatku zostanie omówiona idea oraz szczegóły implementacyjne stojące za Google Protocol Buffers.

A.1. Krótka historia języka

A.2. Zestawienie wydajności mechanizmów serializacji na JVM

A.3. Przykładowe definicje wiadomości

A.4. Dostępne narzędzia

```
message Person {  
  required int32 id = 1;  
  required string name = 2;  
  optional string email = 3;  
}
```

B. Podstawy języka Scala oraz Scala Parser Combinators

Celem tego dodatku jest przybliżenie czytelnikowi języka „Scala” aby w wystarczająco płynny sposób mógł czytać przykłady kodu używane w tym dokumencie.

B.1. Krótka historia języka

Język Scala („Scalable Language”) najłatwiej jest przedstawić jako hybrydę dwóch znanych nurtów programowania: programowania obiektowego oraz funkcyjnego, wraz z powiązanymi z nimi językami programowania. Twórca języka Scala, Martin Oderski ¹ był ściśle związany z językiem Java - był głównym projektantem generyków w Javie (*Java Generics*) oraz głównym autorem utrzymywanej po dziś dzień serii kompilatorów **javac** ².

Jako konkretnych „rodziców” można by wskazać:

- **Java** - jako reprezentant nurtu obiektowego
- oraz języki: **Haskell**, **SML** oraz pewne elementy języka **Erlang** (głównie *Actor model*).

B.2. Podstawy

Ta sekcja służy przybliżeniu czytelnikowi języka *Scala* na poziomie wystarczającym aby swobodnie czytać przykłady kodu umieszczone w tej pracy. W niektórych przykładach pomijane są przypadki skrajne lub nietypowe, celem szybkiego oraz jasnego przedstawienia minimum wiedzy na temat języka aby móc swobodnie go „czytać”.

Scala jest językiem statycznie typowanym posiadającym lokalne „Type Inference”. Pozwala to kompilatorowi *scalac* na „odnajdywanie” typów wszystkich zmiennych oraz typów zwracanych przez metody podczas kompilacji, bez potrzeby definiowania ich wprost. System ten

Użycie nawiasów `()`, średnika `;` oraz kropki `.` jest analogiczne jak w przypadku Javy, jednak w wielu przypadkach opcjonalne gdyż kompilator jest w stanie wydedukować gdzie powinny się znaleźć.

```
val value = Option(42);  
val other = value.getOrElse(0);
```

¹Martin Odersky - Strona domowa: <http://lamp.epfl.ch/~odersky/>

²Wywiad z Martinem Odersky na temat korzeni języka Scala - www.artima.com/.../origins_of_scala

```
// moze zostac zastpione
val value = Option(42)
val other = value orElse 0
```

Jednym z ciekawych przykładów stosowania notacji bez nawiasów i kropek jest *ScalaTest*³ (przy którego pomocy pisano testy w tym projekcie). Przykładowa *asercja* napisana w *DSL*u definiowanym przez tę bibliotekę wygląda następująco:

```
messages should (contain key ("Has") and not contain value ("NoSuchMsg"))
```

Dostępne jest wiele sposobów definiowania metod / pól w klasie, w efekcie (na poziomie bytecode), wszystkie przekładane są na wywołania metod. Dostępne są słowa kluczowe:

- **def**, definiujący zwyczajną metodę instancyjną. Warto nadmienić że Javowa koncepcja pojęcia *static* nie jest dostępna z poziomu Scala.
- **val**, deklarujący „stałą” - to jest metodę która raz zawołana, zwróci wartość oraz pole to będzie konsekwentnie zwracać tą samą wartość. Dodatkowym efektem jest traktowanie zmiennych tego typu analogicznie do Jawowych zmiennych z modyfikatorem **final**.
- **var**, deklaruje zwyczajną „zmienną”, do jakiej przyzwyczajeni jesteśmy z Java.
- modyfikator **lazy**, wpływający na moment inicjalizacji zmiennej - metody zadeklarowane z modyfikatorem **lazy** zostaną dopiero zainicjalizowane podczas pierwszego odwołania się do tego pola z innego miejsca w kodzie. W przypadku pary **lazy val**, metoda ta zostanie zawołana jedynie jednokrotnie, a zwrócona po raz pierwszy wartość zostanie zapisana w cache oraz będzie konsekwentnie zwracana podczas ponownych wywołań tej metody.

Modyfikator **lazy** pozwala na eleganckie budowy konstrukcji stosowanych w Parser Combinators, omówionych poniżej.

B.3. Traits - wmieszanie zachowania do klasy

Słowo kluczowe **trait** rozpoczyna definicję typu zwanego traitem. Implementacja nie różni się (na potrzeby tego szybkiego omówienia) od implementowania klasy, jednak różnica jest podczas „dziedziczenia” przy wykorzystaniu traitów. Nie mówimy bowiem o „dziedziczeniu” w przypadku *trait*ów, a o „wmieszaniu” (ang. *mixin* - wmieszanie) zachowania do klasy konkretnej.

Poniżej został przedstawiony najprostrzy trait zawierający jakieś zachowanie, oraz jeden ze sposobów jego wmieszania do klasy konkretnej. Warto zauważyć że w przypadku wmieszania *traita* A do klasy *Test*, wprowadzamy między nimi relację „Test **IS-A** A”, analogicznie jak w przypadku dziedziczenia.

```
trait A {
  def test = "A" // definicja metody zwracającej "A"
```

³ScalaTest - framework do testowania - <http://www.scalatest.org>

```
}  
  
class Test extends A { } // wmieszanie A  
  
new Test().test // skompiluje i wykona sie poprawnie
```

Co ciekawe, nie zauważamy różnicy w przypadku składni odnoszącej się do dziedziczenia dwóch klas konkretnych, oraz wmieszania traita. Składnia ulega zmianie w przypadku korzystania z więcej niż jeden trait lub domieszania traita do klasy która już dziedziczy po innej klasie, wówczas zamiast słowa kluczowego **extends** należy stosować **with** (nie dozwolone jest wielokrotne zapisanie **extends**, jednak wielokrotne **with** są często spotykane). Przykład wmieszania większej ilości traitów zostanie przedstawiony poniżej.

Jest to namiastka dziedziczenia wielobazowego jednak Scala dzięki swojemu bardzo rygorystycznemu kompilatorowi jest w stanie uniknąć sytuacji gdzie dziedziczenie wielobazowe byłoby niebezpieczne (klasyczne przykłady problematycznych sytuacji w przypadku dziedziczenia wielobazowego można przeczytać w „Symfonii C++”, autorstwa pana Grębosza [Gre08]).

Kompilator *scalac* przy napotkaniu konfliktów nazw mogących doprowadzić do niejasności „którą metodę należy zawołać”, nie skompiluje takiego kodu oraz poprosi o rozwiązanie konfliktu w sposób *explicite*. Jako przykład rozważmy dwa *traity* udostępniające metodę `def test: String`:

```
trait A { def test = "A" }  
trait B { def test = "B" }  
  
class Example extends A with B {  
  // blad kompilacji!  
}
```

Przy napotkaniu problemu tego typu kompilator zgłosi:

```
error: overriding method test in trait A of type => java.lang.String;  
      method test in trait B of type => java.lang.String needs `override`  
class Example extends A with B {
```

Dzieje się tak ponieważ **scalac** próbuje odnaleźć która metoda powinna mieć większą wagę, a tym samym powinna zostać wywołana. Ponieważ nie jesteśmy w stanie dodać modyfikatora **override** do żadnego z *traitów* (ponieważ nie nadpisują one tej metody, a jedynie deklarują), jedynym możliwym miejscem na rozwiązanie tego konfliktu jest uzupełnienie *Example* o następujący fragment kodu, rzucający poprawnie nasze wywołanie metody:

```
class Example extends A with B {  
  // selektywne odwołanie sie do metody konkretnego supertypu  
  override def test = super[B].test  
}
```

```
new Example().test // poprawne
```

B.4. Scala Parser Combinators

Bibliografia

[GH96] Erik Meijer Graham Hutton. *Monadic Parser Combinators*. 1996.

[Gre08] Jerzy Grebosz. *Symfonia C ++ Standard*. 2008.

[Ode07] Martin Odersky. *Programming in Scala*. 2007.