ProtoDoc v1.0
Google Protocol Buffers Documentation Tool

Konrad Malawski
konrad.malawski@java.pl

Termin zajęć: Pon 9:45
EAIiE - Informatyka Stosowana, III Rok

Data oddania projektu: 28 Czerwca 2011

27 czerwca 2011

Spis treści

1 Cel programu

Celem projektu była wstępna implementacja narzędzia typu javadoc dla języka *Google Protocol Buffers*. Protocol Buffers, dalej zwane *ProtoBuf*, jest to zbiór: języku definicji interfejsów, binarnego protokołu oraz rozszerzalnego kompilatora plików *.proto do postaci kompilowalnych plików w dowolnym języku programowania, potrafiących przeczytać binarny strumień ProtoBuf tworząc z niego wygodne do użycia klasy dla programisty. Z racji wysokiej wydajności parsowania tego typu *Wiadomości* Protocol Buffers znajdują zastosowania w dużych systemach Enterprise, gdzie liczby wiadomości osiągają rzędy setek lub tysięcy per system. Niestety nie jest możliwe w sposób czytelny dokumentować znaczenie poszczególnych pól lub wiadomości.

ProtoBuf jest odpowiedzią na tą potrzebę. Narzędzie to (napisane w całości w języku *Scala*) umożliwia analogicznie jak JavaDoc, wygenerowanie dokumentacji projektu w postaci strony www na podstawie plików źródłowych oraz umieszczonych nad polami/wiadomościami komentarzami.

2 Gramatyka Protocol Buffers IDL

2.1 Przykład pliku *.proto

Celem unaocznienia gramatyki przytaczam tutaj przykład rzeczywistego pliku proto:

```
package pl.project13;

/** sample comment */
message FormMessage {

    /** sample comment */
    enum ContactVia {
        SMS = 1;
        PHONE = 2;
    }

    /** sample comment */
    required ContactVia contact_me_via = 1;

    optional string name = 2 [default = "loremipsum"];

    /** sample comment */
    message InnerMessage {
        /** sample comment */
        required string name = 3;
        required string surname = 4;
        optional string age = 5;
    }
}
```

2.2 Gramatyka języka Protocol Buffers IDL

2.2.1 Tabela tokenów

L.p.	Nazwa Tokena	Opis	Komentarz
1	newLine	\r | \n | \r\n	Znak nowej linii, na różnych systemach
2	endOfLine	\n	Rzeczywista nowa linia
3	whiteSpace	" " | \t | \f | {NEW_LINE}	Biały znak
7	commentLine	"/""/"[^\r\n]*/	C-Style Komentarz
8	intLit	decInt | hexInt | octInt	Liczba całkowita
9	decLit	/[1-9]\d*/	Liczba w zapisie dziesiętnym
11	hexLit	/0[xX]([A-Fa-f0-9])+/	Liczba w zapisie heksadecymalnym
11	octLit	/0[0-7]+/	Liczba w zapisie ósemkowym
10	strLit	quote (hexEscape | octEscape | charEscape | /[^\0\n]/)* quote	Ciąg znaków
11	floatLit	/\d+(\.\d+)?([Ee][\+-]?\d+)?/	Liczba zmienno przecinkowa
11	boolLit	true|false	Wartość booleanowa
11	quote	/["']/	Rozpoczęcie lub zakończenie ciągu znaków
15	ident	/[A-Za-z_][\w_]*/	Identyfikator
16	coma	,	Przecinek
17	equal	=	Przypisanie
18	semiColon	;	Średnik, koniec linii kodu
19	openBlock	{	Otwarcie bloku kodu
20	closeBlock	}	Zamknięcie bloku kodu
21	openBrace	[	Otwarcie inicjalizatora
22	closeBrace	]	Zamknięcie inicjalizatora
23	openParant	(	Otwarcie nawiasu
24	closeParant	)	Otwarcie nawiasu
32	hexEscape	/\\[Xx][A-Fa-f0-9]{1,2}/	Escape'owana liczba hexadecymalna
33	octEscape	/\\0?[0-7]{1,3}/	Escape'owana liczba ósemkowa
34	charEscape	/\\[abfnrtv\\\?'"]/	Escape'owany znak
	enum	enum	Enumeracja
	package	package	Paczka
	message	message	Wiadomość

L.p.	Nazwa Tokena	Opis	Komentarz
	repeated	repeated	Pole powtarzane
	optional	optional	Pole opcjonalne
	required	required	Pole wymagane
	true	true	
	false	false	
	camelLit	/[A-Z][A-Za-z_]*/	
	int32	int32	
	uint32	int32	
	int64	int64	
	uint64	uint64	
	bytes	bytes	
	float	float	
	double	double	
	sint32	sint32	Typ całkowity, ze znakiem, 32 bitowy
	sint64	sint64	Typ całkowity, ze znakiem, 64 bitowy
	fixed32	fixed32	Typ
	fixed64	fixed64	Typ
	sfixed32	sfixed32	Typ
	sfixed64	sfixed64	Typ
	bool	bool	Typ
	string	string	Typ
	bytes	bytes	Typ
	userType	/\.? ident (\. ident)* /	Typ

2.2.2 Gramatyka

```
proto : ( message | enum | package | ";" )*

package : "package" ident ( "." ident )* ";"

message : "message" ident messageBody

enum : "enum" ident "{" ( option | enumField | ";" )* "}"

enumField : ident "=" intLit ";"

messageBody : "{" ( field | enum | message | ":" )* "}"

# tag number must be 2^29-1 or lower, not 0, and not 19000-19999 (reserved)
field : label type ident "=" intLit ( "[" fieldOption ( "," fieldOption )* "]" )? ";"

fieldOption : "default" "=" constant

label : "required" | "optional" | "repeated"
```

```
type : "double" | "float" | "int32" | "int64" | "uint32" | "uint64"
       | "sint32" | "sint64" | "fixed32" | "fixed64" | "sfixed32" | "sfixed64"
       | "bool" | "string" | "bytes" | userType

# leading dot for identifiers means they're fully qualified
userType : "."? ident ( "." ident )*

constant : ident | intLit | floatLit | strLit | boolLit
```

3 Opis i schemat struktury logicznej programu

Program korzysta z *Scala ParserCombinators*, części biblioteki **będącej częścią tego języka** przeznaczonej właśnie genrowaniu parserów. Wykorzystanie *ParserCombinators* nad inne znane rozwiązania typu JBison/JFlex i im podobne (JACC (odpowiednik Javowy narzędzia YACC)) motywuję niesamowitym potencjałem języka Scala oraz możliwości wprost programowania oraz wpinania się w nasz odeclowy model obiektowy domeny podczas parsowania. Takie podejście nie byłoby możliwe przy zastosowaniu klasycznych 'compiler compiler'ów.

Pierwszym krokiem jest przeparsowanie pliku proto przez zaimplementowany dla tego projektu kombinator parserów. Krok ten wykonywany jest dla każdego z plików jaki zostanie odnaleziony na ścieżce wejściowej podanej aplikacji. Pierwotnie nie była przewidywana obsługa wielu plików na tym etapie ProtoDoc, jednak implementacja okazała się dzięki Scali bardzo wygodna i sprawna. Po przeparsowaniu wszystkich plików, otrzymujemy listę wiadomości zawierających *kompletną* strukturę pól oraz wiadomości lub enumeracji zagnieżdżonych w tych typach. Zachowana zostaje również poprawność ścieżek package do wiadomości wewnętrznych - wówczas *package* sub-wiadomości powinien zawierać (kończyć się na) *message* w którym się zawiera.

Kolejnym krokiem jest generowanie na podstawie listy wiadomości stron HTML zawierających dokumentację. Dzięki pelnej reprezentacji wszystkich elementów przeparsowanych plików - w tym komentarzy oraz zagnieżdżeń typów w typach proces ten jest stosunkowo prosty i odbywa się przy pomocy zastosowaniu kilku funkcyjnych metod/własności kolekcji języka Scala, konkretniej: map, reduce, fold, oraz filter. Przy pomocy złożeń tych transformacji kolekcji bardzo łatwo było uzyskać na przykład listę wszystkich typów, które następnie transformowano na wspólny im nad typ, aby przekazać do systemu szablonowego. Jest to o tyle ciekawe oraz istotne iż w każdym kroku tego procesu zachowywane są całe struktury obiektów jak i ich typy. W żadnym miejscu programu nie mamy do czynienia z operowaniem na jedynie napisach - gdyż tak byłoby w wielu językach o wiele łatwiej filtrować tylko interesujące nas wiadomości; dzięki *Scala* praca na rzeczywistych typach nie tylko była wygodna, ale również naturalna.

Wybrany do implementacji warstwy widoku język szablonów nazywa się "*Mustache*". Jego nazwa pochodzi od składni jaką się posługuje wypisując zmienne w szablonach *mustache*. Poniżej mały przykład prostego szablonu *mustache*:

```
<ul>
  {{#fields}}
    <li>name: {{name}}</li>
  {{/fields}}
</ul>
```

Proces generowanie ProtoDoc kończy się w momencie przekazania ostatniej wiadomości (lub enumeracji) do systemu szablonów. Wynikowe pliki HTML są zapisywane do podanego podczas uruchamiania aplikacji przez użytkownika folderu.

Przeglądanie stron jest możliwe zarówno przy wykorzystaniu przeglądarek obsługujących JavaScript – wówczas dostępna jest wyszukiwarka jak i nawigacja „na jednej stronie" z zachowaniem historii, oraz przy wykorzystaniu przeglądarek tekstowych (takich jak links). W przypadku przeglądania strony w trybie tekstowym, wyświetlana jest bardzo czytelna dla takich przeglądarek stron wersja, która oczywiście nie wymaga do działania obsługi *JavaScript*u.

4 Wykorzystane pakiety zewnętrznye i narzędzia

Poniżej wymieniono zależności projektu wykorzystane celem implementacji jak i testowania tego projektu.

- **Scala** — sam język w którym zaimplementowano to rozwiązanie jest wart wymienienia jako najważniejsze narzędzie w tym projekcie. Jest on niebywałym połączeniem języków funkcyjnych (takich jak LISP czy Haskell) oraz klasycznych (w sensie 'znanych nam') zorientowanych stricte obiektowo, vide Java. Scala jest językiem statycznie typowanym, jednak posiada o wiele bardziej rozbudowany system inferowania typu niż inne obecnie popularne języki na rynku informatycznym, właściwość ta niejednokrotnie przydała się podczas implementowania parsera, gdzie łatwo było napisać 'implicite' konwersje między pewnymi typami. Same ParserCombinators oczywiście również zasługują na wspomnienie, są bowiem częścią biblioteki standardowej Scala i dzięki nim projekt ten stał się *czystą przyjemnością*.

- **Scalate** oraz **Mustache** — Scalate został wykorzystany jako silnik szablonów celem renderowania wynikowych stron HTML z dokumentacją ProtoBuf. Mustache jest jednym z kilku implementowanych przez Scalate języków szablonów. Mustache wyróżnia się tym spośród innych iż zupełnie nie posiada logiki - dzięki temu logika biznesowa (bądź wszelakiego rodzaju filtrowania kolekcji etc) nie przecieknę nam do wartwy widoku. Dzięki niemu testowanie widoku oraz rozwój aplikacji włącznie ze zmianami reprezentacji pewnych typów przed przekazaniem ich do szablonu były bardzo wygodne.

- **sbt** (Simple Build Tool) – jest to odpowiednik Maven lub Ant znanych ze świata Javowego jednak dla języka Scala. Możliwe jest również kompilowanie przy jego pomocy projektów Java jednak nie jest to raczej spotykane. Dzięki łatwości konfiguracji zadań sbt w przeciwieństwie do dodawania własnych pluginów do systemu Maven okazał się wysoce pomocny podczas budowania wykonywalnego pliku jar z protodoc. Najważniejszą cechą **sbt** która okazała się wysoce pomocna podczas tego projektu jest możliwość odpalania testów w trybie ciągłym, dzięki czemu zawsze (co każdą zmianę w pliku źródłowym) mamy informację zwrotną jaki wpływ ta zmiana miała na testy - metoda ta w połaczeniu z metodyką TDD okazała się wysoce efektywna i przyjemna podczas rozwoju oraz poprawiania usterek w aplikacji.

- **ScalaTest** – Prosty framework służący pisaniu testów jednostkowych w języku Scala. Można go potraktować jako odpowiednik JUnit (znany z świata Javowego) jednak odrobinę potężniejszy - dzięki przewadze semantycznej języka Scala.

5 Informacje o zastosowaniu specyficznych metod rozwiązania problemu

Specyficznym rozwiązaniem w samym parserze są *Scala ParserCombinators* - część języka dedykowana budowaniu Parserów, przy pomocy przeznaczonego temu DSLa (Domain Specific Language), choć nadal pozostając z „czystej Scala". Jedną z zalet zastosowania tego narzędzia nad innymi jest brak kolejnego procesu kompilacji który zazwyczajbyśmy musieli zastosować - na przykład wykorzystując Flex/Bisona lub inne rozwiązania którymi „klasycznie" generuje się parsery.

ParserCombinators wywodzą się z bardzo ciekawego pomysłu, jak sama nazwa wskazuje, kombinowania parserów. Dzięki temu poniższy, poprawny w języku Scala, zapis:

```
def boolValue = ("true" | "false")
```

Definiuje w sumie 3 parsery:

- parser wyrażenia „true", zwracający wartość jako *java.lang.String* (gwarantują to właściwości typów języka *Scala*)
- parser wyrażenia „false", zwracający wartość jako *java.lang.String* (gwarantują to właściwości typów języka *Scala*)
- parser **alternatywa**, dwóch wcześniej zdefiniowanych parserów.

Dzięki takiej kombinacji parserów, osiągamy bardzo czytelną składnię do definiowania naszego języka oraz bardzo potężne narzędzie w postaci możliwości „wpięcia się" w dowolny moment parsowania zwyczajnym kodem *Scala* i przetworzenia na przykład wybranego typu, na wartość innego typu. Idąc tym tokiem rozumowania możemy rozbudować nasz boolValue parser aby zwracał wartość **Boolean** zamiast **String**.

Dokonamy tego dzięki zastosowaniu kolejnego kombinatora, o nazwie *„function applying combinator"* oraz oznaczeniu: `^^`. Kombinator ten dokonuje transformacji kombinatora parserów po jego lewej stronie, przy pomocy funkcji przekazanej jego prawej stronie. W języku Scala przekazywanie funkcji wyższego rzędu jest dość trywialne, zatem nasz problem transformacji **String**a na **Boolean** rozwiązalibyśmy następująco:

```
def boolValue = ("true" | "false") ^^ {
  s =>
    s.toBoolean
}
```

Pamiętamy nadal iż typ kombinatora parserów na którym pracujemy tą funkcją (przekazany jako parametr funkcji anonimowej - **s**) to *java.lang.String*. Przypominamy sobie iż klasyczny String nie posiada przecież metody **toBoolean()** (w *Scala* dozwolone jest pomijanie nawiasów okrągłych, stąd **toBoolean** oznacza wywołanie metody **toBoolean()**). Poznaliśmy właśnie kolejną możliwość języka Scala.

Istnieją tak zwane konwersje ***implicit***, dzięki którym „jeżeli wygląda na to iż potrzebujemy skonwertować jeden typ na inny" zostanie zastosowana konwersja

implicit, i uzyskamy pożądany typ. W powyższym przykładzie zadziałało to na zasadzie, istniejącej w bibliotece standardowej konwersji *implicit*:

```
implicit def string2richString(str: String) = new RichString(str)
```

Dzięki tej konwersji otrzymaliśmy obiekt RichString, na którym zostanie zawołana metoda toBoolean (`http://www.scala-lang.org/api/2.7.4/scala/runtime/RichString.html#toBoolean`). Zwracany typ tej metody to **scala.Boolean**. Ostatnim krokiem utworzenia tego kombinatora jest kolejna konwersja *implicit* która zostanie wywołana w momencie zwrócenia naszej funkcji anonimowej. Podczas gdy zwróciliśmy typ scala.Boolean (słowo kluczowe "**return**"jest w tym przypadku opcjonalne), def boolValue powinno być **Parser**em, dzięki kolejnej implicit konwersji ostatecznie boolValue otrzyma typ Parser[Boolean] i gdy będziemy chcieli pobrać wartość tym sposobem przeparsowanego Tokena, otrzymamy wartość typu **scala.Boolean**.

Jak łatwo się domyślić, przedstawione mechanizmy do dopiero czubek góry lodowej jaką jest język **Scala** i **ParserCombinators**. Uważam, że praca z tak zaawansowanym językiem jak i DSLem do definiowania parserów gramatyk jest niebywale interesującym oraz pouczającym przeżyciem.

Ponad to, otrzymany w efekcie kod Parsera do złudzenia przypomina gramatykę EBNF, a więc jest bardzo czytelna oraz łatwa w implementacji oraz dalszym rozwoju. Przed wyborem tego narzędzia testowałem klasyczne podejścia do problemu przy pomocy JFlex/JBison jednak nie były to tak przyjemne w użyciu oraz rozwoju narzędzia jak czysty język Scala i jego funkcyjna natura.

6 Instrukcja obsługi

6.1 Pobranie aktualnych źródeł

Aby pobrać najświeższą wersję źródeł należy skorzystać z systemu wersjonowania **git**:

```
git clone git://github.com/ktoso/protodoc-scala.git
cd protodoc-scala
```

6.2 Uruchomienie aplikacji przez sbt

Następnie do uruchomienia aplikacji (bądź jej skompilowania) o skompilowania można wykorzystać poniższy ciąg poleceń. Zakładamy iż użytkownik posiada zainstalowany w systemie **sbt** - Simple Build Tool, najpopularniejsze narzędzie budowania projektów w języku Scala (obsługuje również projekty w Java, z domyślnym layotem **Maven**owym).

Aby uruchomić aplikację:

```
sbt run [parametry aplikacji]
```

W przypadku nie podania wymaganych parametrów do uruchomienia ProtoDoc, zostaniemy o tym poinformowani następującą wiadomością:

```
usage: ProtoDoc [options] proto_dir out_dir

options:
  -v, --verbose   active verbose output, [default = false]
  ------------------------------------------------
  proto_dir       directory containing proto files to parse
  out_dir         output directory for the protodoc html webpage
```

Zatem poprawne wywołanie aplikacji wyglądałoby następująco:

```
sbt run ~/coding/protodoc-scala/src/main/proto/simple /tmp

# kompilacja aplikacji

[info] == run ==
[info] Running pl.project13.protodoc.runner.ProtoDocMain
                        ~/coding/protodoc-scala/src/main/proto/simple /tmp
verbose: false
proto_dir: ~/coding/protodoc-scala/src/main/proto/simple
out_dir: /tmp
Parsing file: ~/coding/protodoc-scala/src/main/proto/simple/simple.proto
Parsing file: ~/coding/protodoc-scala/src/main/proto/simple/multiple_inner_msgs.proto
Parsing file: ~/coding/protodoc-scala/src/main/proto/simple/amazing_message.proto
Parsing file: ~/coding/protodoc-scala/src/main/proto/simple/with_enum.proto
Message: pl.project13.WithEnum
Message: pl.project13.AmazingMessage
Inner:   pl.project13.AmazingMessage.InnerMessage
Message: pl.project13.TopLevel
Inner:   pl.project13.TopLevel.MiddleLevel
```

```
Inner:   pl.project13.TopLevel.MiddleLevel.InnerInnerLevel
Message: pl.project13.MessageWithInner
Inner:   pl.project13.MessageWithInner.InnerMessage
[info] == run ==
[success] Successful.
```

Możliwe jest również przekazanie opcji *-v*, która skutkuje znacznym zwiększeniem stopnia logowanych na konsolę wiadomości. Raczej powinna być ona zbędna użytkownikowi, pozostawiono ją dostępną jednak na wypadek potrzeby debugowania aplikacji 'wizualnie'.

W razie potrzeby możliwe jest również wygenerowanie pliku wykonywalnego JAR, aby tego dokonać należy skorzysać z dwóch poleceń sbt:

```
sbt collect-jars
sbt package
```

Plik znajdzie się w folderze **target/scala-2.8.1** (ponieważ pod tą wersję Scala projekt został właśnie skompilowany).

6.3 Z pliku JAR

Udostępniona została również skompilowana paczka tej aplikacji, można ją poprać z:

```
wget http://up.project13.pl/protodoc/protodoc-1.0.tar.gz
```

A następnie wykonać w następujący sposób:

```
tar xzvf protodoc-1.0.tar.gz
java -jar protodoc-1.0.jar [parametry aplikacji]
```

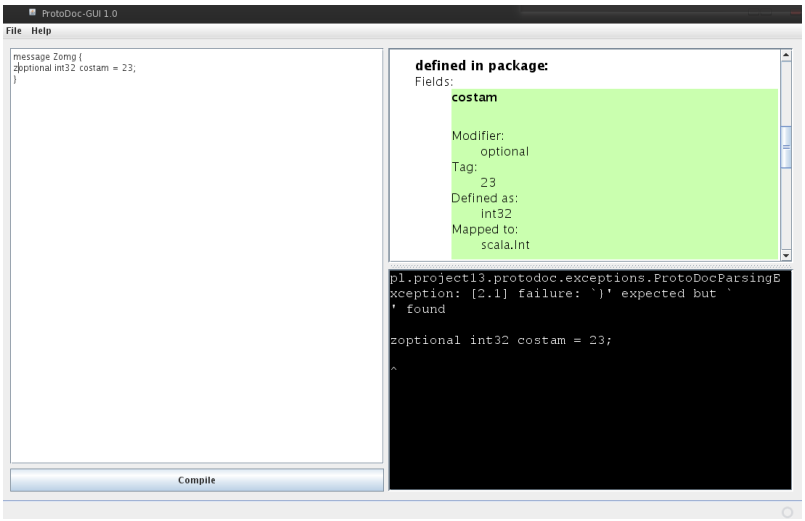
6.4 Przy wykorzystaniu aplikacji protodoc-gui

Dla tego projektu powstała również malutka aplikacja pozwalająca możliwie najłatwiej testować parser ręcznie. Aplikację można uruchomić poprzez przejście do jej folderu oraz uruchomienie polecenia:

```
git clone git://github.com/ktoso/protodoc-gui.git
ant run
```

Zakładamy iż ścieżki do bibliotek zostały ustalone poprawnie przez użytkownika oraz iż posiada zainstalowane na swoim systemie narzędzie **ant**.

Aplikacja ta pozwala na edycję definicji Protocol Buffers po lewej stronie aplikacji oraz skompilowanie jej w miejscu. Kompilacja definicji tworzonej po lewej stronie jest automatycznie właczana co zmianę w źródle. Dzięki temu można na żywo śledzić jak zachowuje się parser podczas powstawania kolejnych fragmentów definicji interfejsu.

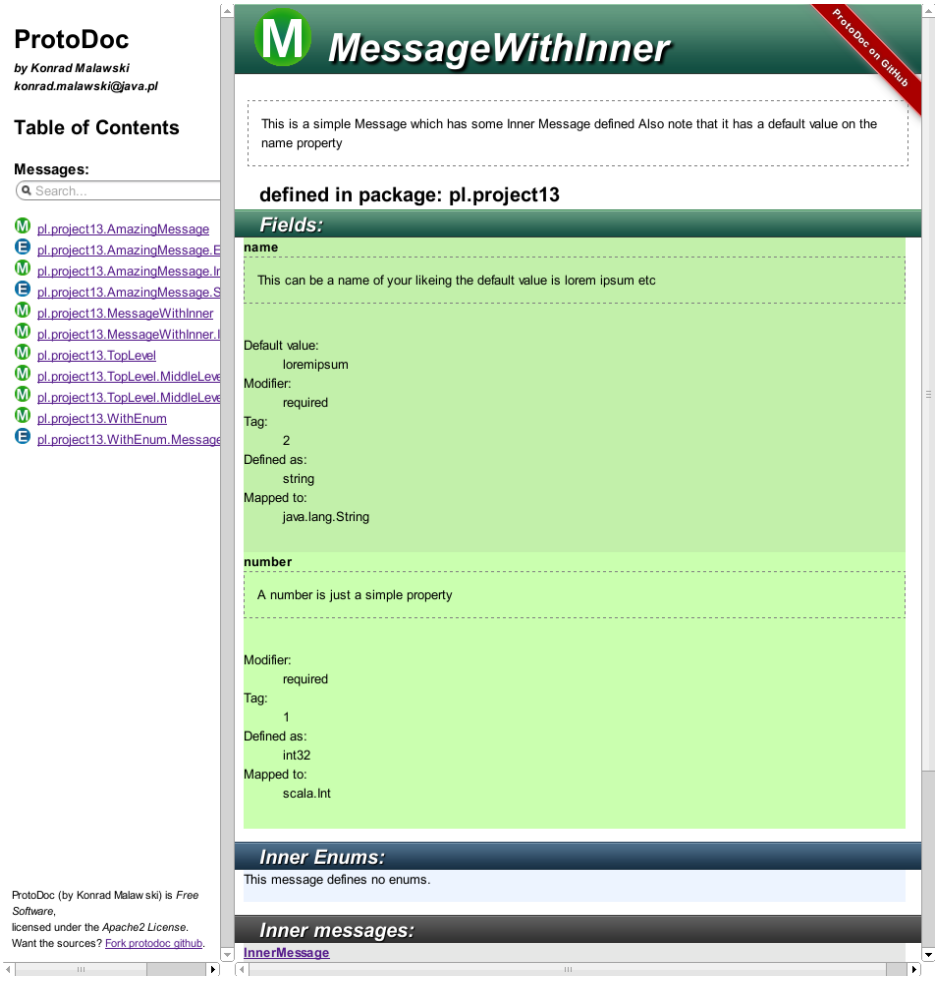


Rysunek 1: Screenshot GUI ułatwiającego manualne testowanie aplikacji

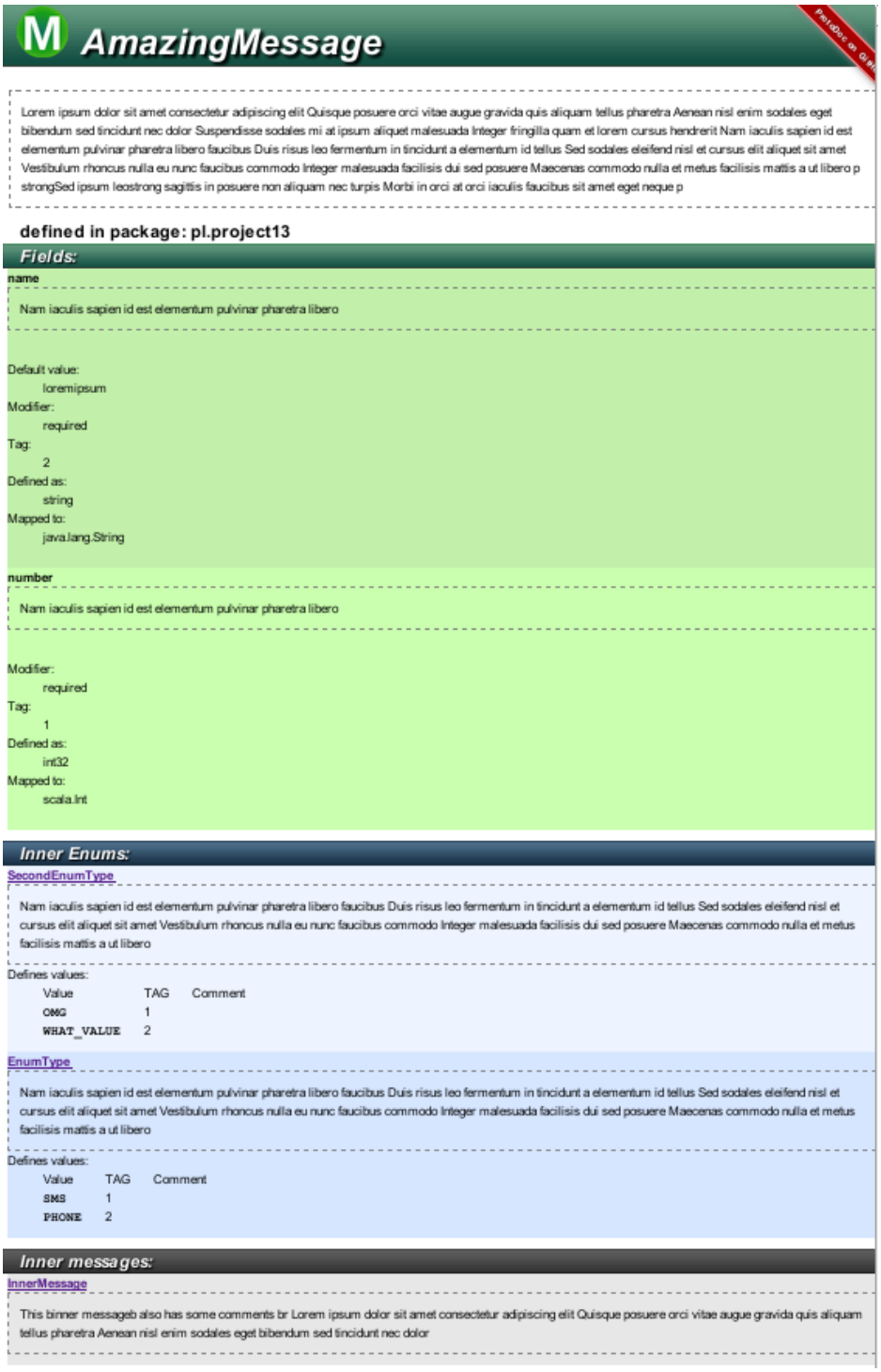
7 Przykładowe wyniki działania programu

7.1 Wynikowa strona www ProtoDoc

Z przykładowych plików *.proto, umieszczonych w folderze projektu 'src/main/proto/simple' zostanie wygenerowana strona ProtoDoc o wyglądzie przedstawionym na załączonych zrzutach ekranu. W obecnej wersji aplikacji generowane są widoki Message (wiadomości - vide Rysunek 3), Enum (enumeracji - vide Rysunek 4) oraz ogólny widok index.html (vide Rysunek 2) zawierający również spis wszystkich Typów dostępnych w analizowanym folderze. Dostępna jest również wyszukiwarka, filtrująca tę listę na podstawie wpisanego fragmentu nazwy.



Rysunek 2: Główny widok ProtoDoc



Rysunek 3: Widok dokumentacji wiadomości

Rysunek 4: Widok dokumentacji enumeracji (enum)

7.2 Przykłady wykrywanych błędów

ProtoDoc w obecnej wersji potrafi już wykrywać niektóre rodzaje błędów jakie może napotkać w plikach źródłowych. Poniżej zostaną przedstawione niektóre z nich. Poza przedstawionymi tutaj mechanizmami typowe „nie pasowanie do gramatyki" na przykład poprzez definicję enumeracji jako „top level" jednostki również zostanie zgłoszone w analogiczny sposób. Każdy z tych składniowych niesie ze sobą informację o miejscu gdzie błąd wystąpił (w postaci numeru linii i kolumny), wraz dotyczącym błędu fragmentem kodu oraz wskazaniem znakiem ^ miejsca wystąpienia problemu.

7.2.1 Wykrycie niepoprawnego modyfikatora

Poniższa wiadomość zawierająca błąd w słowie „optional", będącym modifykatorem pola ProtoBuf.

```
message Msg {
zoptional int32 number = 12;
}
```

wywoła komunikat błędu:

```
pl.project13.protodoc.exceptions.ProtoDocParsingException:
[2.1] failure: '}' expected but '' found

zoptional int32 costam = 23;
^
```

Warto zauważyć znak ^ oznaczający dokładnie miejsce wystąpienia problemu, ponad to zwracana jest pozycja problemu, w postaci objętych kwadratowymi nawiasami numeru wiersza oraz numeru kolumny gdzie problem wystąpił.

7.2.2 Wykrycie zastosowanie nie zdefiniowanego typu

Poniższa definicja zawiera odwołanie do niezdefiniowanego typu - UnknownType. ProtoDoc jest w stanie wykryć ten problem oraz zareaguje rzuceniem wyjatku typu „*pl.project13.protodoc.exceptions.UnknownTypeException*" podając również przyczynę problemu.

```
message Msg {
required UnknownType field = 23;
}
```

spowoduje rzucenie następującego komunikatu:

```
pl.project13.protodoc.exceptions.UnknownTypeException:
Unable to link 'UnknownType' to any known enum Type.
```

7.2.3 Wykrycie nie zdefiniowanej opcji

Program jest gotowy do dalszego rozwoju w kierunku deklarowania własnych Opcji. Jest to funkcja dokumentowana przez Google jako najprawdopodobniej nie potrzebna 98 procent użytkowników Protocol Buffers jednak jako, że dokładnie przez opcje jest zdefiniowana wartość domyślna pola, po niekąd część funkcjonalności (rozpoznawanie) ich została już zaimplementowana w tej wersji ProtoDoc. Poniżej przykład nie odnalezienia pasującej opcji na polu *field*.

```
message Msg {
  required string field = 1 [default = VALUE];
  required uint64 field = 2 [def = VALUE];
}
```

Spowoduje to wypisanie następującego komunikatu błędu:

```
pl.project13.protodoc.exceptions.ProtoDocParsingException:
[3.30] failure: 'default' expected but 'd' found

  required uint64 field = 2 [def = VALUE];

                            ^
```

Jak widać, opcja **def** nie została odnaleziona spośród rozpoznawanych opcji (zbiór jedynie opcji **default**, i został rzucony wyjątek ProtoDocParsingException.

7.3 Przykład możliwie zaawansowanej wiadomości rozpoznawanej na tym etapie przez ProtoDoc

Poniżej przykład wiadomości wykorzystujący wszystkie zaimplementowane możliwości parsera oraz generatora będących częścią ProtoDoc v1.0.

```
package pl.project13.protodoc;

message Msg {

  enum KnownType {
    VALUE = 1;
    OTHER = 2;
  }

  /** This is an inner message */
  message InnerMsg {
    enum InnerEnum {  }

    /** Infinite embeddability */
    message InnerInnerMsg {  }

    required double dNumber = 1;
    optional float fNumber = 2;
  }

  required string field = 1 [default = VALUE];
  optional sint64 number = 2;
  repeated KnownType repEnum = 3;
}
```

8 Ograniczenia programu

- Nie są obsługiwane Serwisy
- Nie jest sprawdzana poprawność wartości domyślnych pól (czy nie przypisujemy napisu do pola uint32 itp).
- Message jeszcze nie mogą być, tak jak Enum'y, stosowane jako typy pól. Zostanie to jednak szybko dodane - analogiczny mechanizm działa już z Typami enum.
- Message jeszcze nie są w stanie po sobie dziedziczyć
- Nie są obsługiwane pola rosrzerzenia Option ani Extension

9 Możliwe rozszerzenia programu

W związku z planowanym kontunuowaniem prac nad tym projektem, możliwości rozwoju były ciagle brane pod uwagę podczas implementowania tej aplikacji. Obecnie jako realne oceniam *usunięcie wszystkich jeszcze nie zrealizowanych funkcji (wymienianych powyżej).*

Ponadto, ze względu na architekturę tego programu oraz znajomość systemów zarządzania projektami **Maven** oraz **sbt**, uważam że dopisanie pluginów do obu tych systemów aby ProtoDoc mógł normalnie być integrowany z cyklem życia aplikacji byłoby dość trywialne (zważywszy na moje wcześniejsze doświadczenie z implementowaniem tego typu pluginów) stąd też takie integracje powstaną. Istotnym jest podkreślić dlaczego taka integracja z zewnętrznymi systemami zarządzania cyklem życia aplikacji jest istotna - firmy korzystające z rozwiązań klasy java enterprise zawsze korzystają z tego typu systemów, włącznie z fazą generowania dokumentacji programistycznej. Wpięcie się do tych procesów poprzez napisanie łatwo używalnego pluginu automatyzującego ten proces ułatwiłoby znacznie proces adaptacji ProtoDoc w dużych firmach, i mogłoby faktycznie pomóc przyjęciu się tej aplikacji w świecie Enterprise.

10 Proces Test Driven Development a rozwój tej aplikacji

Aplikacja ta była rozwijana w całości w metodologii 'Test Driven Development'. Polega ona na pisaniu testów jednostkowych zanim powstanie właściwa implementacja oraz oczywiście w razie znalezienia jakiś błędów w aplikacji również poprzedzania prób naprawy ich napisaniem odpowiedniego testu replikującego dany problem. Dzięki rygorystycznemu przestrzeganiu tej metodologii udało mi się uniknąć wielu potencjalnych regresji w zachowaniu parsera, ponieważ nowe zmiany wprowadzające łamanie wcześniej już ustalonych kontraktów można było szybko wychwycić nie przechodzeniem poprzednich testów.

Wielokrotnie dzięki zastosowaniu tego podejścia udało mi się dostrzec potencjalnie luki w parserze oraz sprawnie je naprawić. Wynikiem tego procesu jest ponad 30 testów sprawdzających wybiórczo poszczególne funkcjonalności aplikacji, począwszy od samego parsera a kończąc już na walidowaniu wynikowych string HTML.

Poniżej kilka przykładowych testów tej aplikacji, aby unaocznić jak ekspresywnym i potężnym językiem jest scala:

```
"ProtoDocTemplateEngine" should "render simple message page" in {
  val message = ProtoBufParser.parse(sampleMessageProtoString)
  val page = templateEngine.renderMessagePage(message)

  page should include ("pl.project13.protobuf")
  page should include ("AmazingMessage")
  page should include ("name")
  page should include ("age")
}
```

Wyniki tak przeprowadzanych testów reprezentowane są następnie w następującej postaci:

```
[info] MessageTemplateTest:
[info] ProtoDocTemplateEngine
[info] - should render simple message page
[info] - should render top level message comment
```

W przypadku niepododzenia testów, owczywiście pojawiłby się komunikat o nie spełnieniu asercji, na przykład:

```
[info] Multi Line Comment on top level message
[info] - should be parsed properly *** FAILED ***
[info]   "comment" did not include substring "NOT" (CommentsTest.scala:83)
```

Dzięki ciągłemu testowaniu tworzonej aplikacji łatwiej jest utrzymać jakość kodu oraz zabezpieczamy się przed regresjami funkcjonalności - co uważam za bezcenne.

11 Bibliografia

1. dokumentacja języka Protocol Buffers IDL – `http://code.google.com/apis/protocolbuffers/docs/proto.html`
2. `http://code.google.com/p/protobuf/` – oficjalna strona domowa projektu (**kod źródłowy**)
3. oficjalna dokumentacja projektu – `http://code.google.com/apis/protocolbuffers/docs/overview.html`
4. ogólny opis języka / narzędzia – `http://en.wikipedia.org/wiki/Protobuf`
5. Dokumentacja Scala, dotycząca ParserCombinators – `http://www.scala-lang.org/api/current/scala/util/parsing/combinator/Parsers.html`
6. Poradnik tworzenia parserów w języku Scala – `http://henkelmann.eu/2011/1/13/an_introduction_to_scala_parser_combinators`
7. Dokumentacja języka szablonów Mustache – `http://mustache.github.com/mustache.5.html`
8. Dokumentacja biblioteki Scalate – `http://scalate.fusesource.org/`
9. Dokumentacja biblioteki ScalaTest 1.5 (kompatybilnej z Scala 2.8.1) – `www.scalatest.org/scaladoc-1.5/`