AKADEMIA GÓRNICZO-HUTNICZA
IM. STANISŁAWA STASZICA W KRAKOWIE

Wydział Informatyki, Elektroniki i Telekomunikacji
Katedra Informatyki

PROJEKT INŻYNIERSKI

ROZSZERZENIE PROCESU BUDOWY OPROGRAMOWANIA O MECHANIZM AUTOMATYZUJĄCY JEGO LOKALIZACJĘ

ŁUKASZ JACKOWSKI, GRZEGORZ KACZMARCZYK

OPIEKUN:
dr hab. inż. Marek Kisiel-Dorohinicki

Kraków 2014

Spis treści

1 Koncepcja Systemu

Proces lokalizacji oprogramowania jest nieodłącznym elementem tworzenia większości współczesnych aplikacji. Niestety w większości przypadków proces tłumaczenia nie jest ustandaryzowany i zintegrowany z procesem wydawania nowych wersji produktu. Można spotkać się z sytuacjami gdzie osoba odpowiedzialna za tłumaczenia ręcznie przegląda wszystkie pliki z tłumaczeniami w poszukiwaniu duplikatów, błędów i niewykorzystywanych kluczy, co czyni ten proces żmudnym oraz podatnym na błędy.

Sytuacja jest podobna do tej sprzed kilkunastu lat, gdy w świecie Javy wiodącym narzędziem do budowania projektów był Ant. Każdy programista stosował wtedy swoją własną konwencję i organizował strukturę plików na własną rękę. Brak ogólnie przyjętego standardu sprawiał, że wdrożenie się przez nowe osoby do projektu było ciężkie i często zdarzało się, że nawet po jakimś czasie nowa osoba nie zdawała sobie sprawy z istnienia dodatkowych skryptów ułatwiających pracę. Jednak parę lat później pojawił się Maven, który wprowadził ideę Convention over Configuration, dzięki czemu obecnie większość projektów trzyma się ogólnie przyjętego schematu, co znacznie ułatwia pracę programistom. Podobne rozwiązanie można by zastosować w procesie lokalizacji oprogramowania.

Nasz projekt będzie składał się z trzech elementów:

- protokołu służącemu wymianie informacji między aplikacją a podmiotem świadczącym tłumaczenia
- aplikacji klienckiej służącej do zarządzania lokalizacją projektu
- aplikacji serwerowej umożliwiającej zarządzanie projektami oraz eksport do formatu XLIFF

Poniżej znajduje się bardziej szczegółowy opis poszczególnych punktów.

1.1 Protokół komunikacji

Na protokół komunikacyjny pomiędzy aplikacją kliencką a serwerową składać się będą: struktura zasobów przesyłanych przez obie strony sekwencja wysyłanych komunikatów

Definicja protokołu jest kluczowa dla późniejszego zdefiniowania działania aplikacji klienckiej i serwerowej.

1.2 Aplikacja kliencka

Aplikacja kliencka służyć będzie do analizowania projektu w poszukiwaniu plików zawierających zasoby do przetłumaczenia. Dodatkowo użytkownik otrzyma możliwość konfigurowania sposobu tłumaczenia (np. języki wyjściowe). Następnie program pozwoli na wysłanie zasobów do podmiotu tłumaczącego oraz odebranie ich po zakończeniu procesu zgodnie ze standardami przyjętego protokołu komunikacji.

1.3 Aplikacja serwerowa

Aplikacja serwerowa będzie umożliwiała założenie, aktualizację projektu tłumaczeniowego jak również przeglądanie istniejących projektów. Kluczowym jej zadaniem będzie udostępnianie gotowych zasobów aplikacji klienckiej w ramach przyjętego protokołu komunikacji. Ponadto możliwy będzie eksport do formatu XLIFF (ogólnie przyjętego i powszechnego formatu używanego przez tłumaczy) oraz edycja plików XLIFF.

1.4 Ogólne założenia na temat usługi

- Proces lokalizacji pojedynczej aplikacji jest zorganizowany w projekt, w ramach którego dodawane są kolejne zlecenia. Zlecenie obrazuje stan systemu w danym momencie. Pomiędzy zleceniami zachodzi dodanie, usuwanie lub zmiana wartości kluczy będących przedmiotem usługi.

- Aplikacja nie zajmuje się wersjonowaniem tłumaczonych plików - za każdym razem wysyłamy wszystkie pliki (te do tłumaczenia jak również i te już przetłumaczone) - szczegółową analizą zajmie się wyspecjalizowane narzędzie po stronie tłumacza.

- Użytkownik ma zdefiniowane dwa przypadki użycia aplikacji klienckiej:

  - założenie projektu na serwerze (wymaga: podania danych klienta, danych biura tłumaczeń oraz wygenerowanie unikalnego identyfikatora projektu dla klienta)

  - aktualizacja projektu (wymaga: podania znanego identyfikatora projektu, wygenerowania unikalnego w obrębie projektu identyfikatora zlecenia oraz zdefiniowania danych określonych przez strukturę pliku konfiguracyjnego)

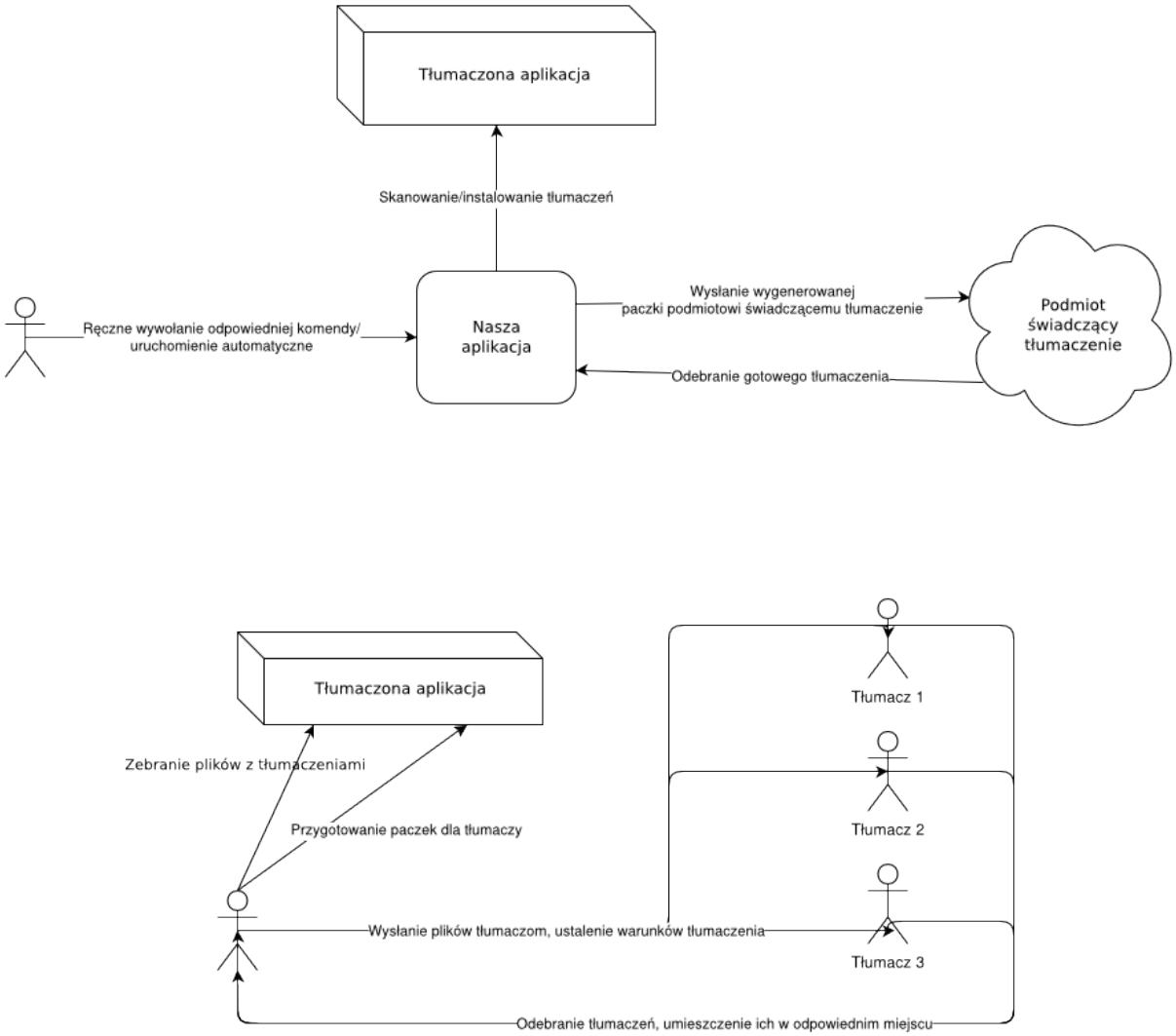
- Aplikacja serwerowa istnieje wyłącznie w celach demonstracyjnych umożliwiając bazową interakcję z częścią kliencką oraz wizualizując poszczególne stany procesu lokalizacji oprogramowania.

- Aplikacja kliencka przechowuje dane tymczasowe w katalogu roboczym. Wśród danych zgromadzonych w tym katalogu są informacje takie jak: lista zarządzanych plików, ID projektu, bieżąca wersja, itd. W razie utraty katalogu konieczne będzie podanie ponownie danych uzyskanych przy pierwszym uruchomieniu.

- Najistotniejsze z perspektywy działania całości systemu jest wygenerowanie paczki opisującej zlecenie. Komunikacja pomiędzy aplikacją kliencką a serwerową odbywać się może na różne sposoby - z wykorzystaniem API udostępnianego przez serwer (tu podmiot świadczący usługę tłumaczeniową) czy nawet drogą mailową. W związku z tym wymaganiem udostępnimy proste API umożliwiające wysłanie paczki jak i pobranie wyników tłumaczenia zgodnie z zaistniałą potrzebą.

1.5 Schemat ideowy usługi

Poniżej znajduje się bardzo poglądowy schemat w różnicy między wykorzystaniem zintegrowanego rozwiązania takiego jak to tworzone przez nas (górny rysunek), a sytuacją gdy tłumaczenia są indywidualnie zamawiane u tłumaczy.



Rysunek 1: Schemat ideowy usługi

Dodanie dodatkowej “warstwy abstrakcji” jaką jest nasza aplikacja oraz ustandaryzowany format sprawia, że znaczna część kroków zostaje zautomatyzowana, bądź odpowiedzialność za nie (jak np. znajdywanie tłumaczy) jest przerzucona na podmiot podejmujący się obsługi tłumaczenia aplikacji.

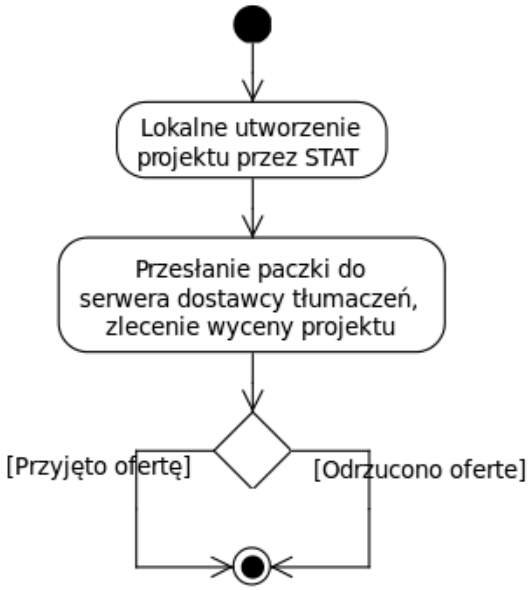
1.6 Ogólny opis użycia aplikacji

Zakładamy, że w danym projekcie istnieje osobowa odpowiedzialna za utrzymywanie tłumaczeń - nazwijmy ją Opiekunem Tłumaczeń. Opiekun do swej pracy wykorzystuje narzędzie STAT. Pracę zaczyna od założenia u siebie lokalnie projektu, wskazując pliki mające zostać objęte zarządzaniem oraz ustala warunki tłumaczenia (tj. języki). Używając dalej STAT-u przesyła do biura tłumaczeń prośbę o wycenę projektu. Po ustaleniu warunków współpracy biuro tłumaczeń może rozpocząć swoją pracę. Opiekun w razie potrzeby aktualizuje złożone już wcześniej zamówienie (bądź tworzy nowe) oraz pobiera sukcesywnie kolejne przetłumaczone paczki.

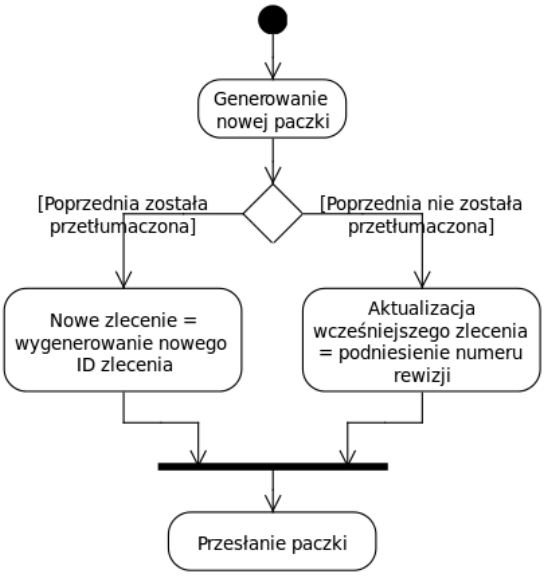
1.7 Schematy działania

Poniżej znajdują się dwa schematy obrazujące działanie finalnego projektu.

Pierwszy przedstawia działanie podczas tworzenia projektu programem STAT. Z uwagi na dążenie do maksymalnej prostoty tak systemu jak i protokołu wymiany danych pewne nieodzowne elementy jak choćby proces negocjowania warunków umowy pomiędzy klientem a biurem tłumaczeń przebiega poza systemem i nie jest przez niego wspierany.



Kolejny schemat przedstawia działanie narzędzia STAT w sytuacji generowania nowej paczki do przetłumaczenia. Przyjęliśmy założenie, że każda kolejna wysłana paczka jest kolejnym zleceniem (i może być np. kryterium wyznaczania kosztów tłumaczenia).



2 Zastosowany proces

Ze względu na to iż projekt tworzony był tylko w dwie osoby, uznaliśmy że użycie bardzo sformalizowanego podejścia byłoby zbędne. Dlatego też w pracach nad stworzeniem projektu stosowaliśmy podejście zgodne z szeroko rozumianymi standardami agile - dążyliśmy do działania aplikacji na każdym etapie jej rozwoju, zaś nowe funkcje dodawaliśmy stopniowo, niejednokrotnie zmieniając je po konsultacjach z klientem.

W celu podziału prac, wydzieliliśmy w projekcie możliwe niezależne obszary:

- Aplikacja kliencka (stat)
- Aplikacja serwerowa (pseudo CAT)
- Warstwa integracyjna - IntegrationAPI
- Wtyczka maven-a
- Konwerter formatów STAB - XLIFF

Ponadto kolejne etapy rozwoju zorganizowaliśmy w taki sposób, żeby jak najszybciej można było obserwować efekty prac (poniżej lista kamieni milowych).

Dla lepszej organizacji pracy wykorzystywaliśmy usługi:

- Bitbucket - hosting repozytorium Git, issue tracker, wiki
- drone.io - Continuous Integration

3 Kamienie milowe

1. Zdefiniowanie protokołu (struktury generowanych zasobów i zawarte w niej informacje) + makieta aplikacji klienckiej (generowanie paczek bez możliwości konfiguracji skanera) - 24.06.13
2. Wstępna implementacja aplikacji serwerowej. Funkcjonalność: (ręczne) ładowanie paczek. Konwerter STAB do XLIFF - 17.08.13
3. Dostarczenie API do tworzenia rozszerzeń umożliwiających wysyłanie paczek i odbieranie wyników tłumaczenia. Część kliencka i serwerowa. Dodanie do serwera możliwości zakładania kont użytkownika oraz organizacja wrzucanych paczek w projekty - 17.09.13
4. Dalsza rozbudowa aplikacji klienckiej - dodanie możliwości konfiguracji skanera, zapisywanie informacji o projekcie w katalogu projektu - 17.10.13
5. Edytor plików XLIFF w aplikacji serwerowej, konwerter XLIFF do STAB - 17.11.13
6. Dostarczenie dodatkowych interfejsów - graficznego i wtyczki do Mavena - 17.12.13

Wszystkie kamienie milowe zakładają równoczesne tworzenie dokumentacji.

4 Szczegółowe plany iteracji i podział prac

4.1 Plan pierwszej iteracji - 03.06 - 24.06

- Zdefiniowanie struktury paczki (wspólnie)
- Zdefiniowanie schematu XML dla pliku opisującego projekt (Łukasz Jackowski)
- Makieta aplikacji bez opcji konfiguracyjnych, generująca paczkę dla znalezionych plików *.properties (Grzegorz Kaczmarczyk)

4.2 Plan drugiej iteracji - 10.07.13 - 17.08.13

- Wybór technologii (wspólnie)
- Konwersja paczek do formatu XLIFF (Grzegorz Kaczmarczyk)
- Szkielet serwera (tj. skonfigurowanie i połączenie użytych komponentów + strona do ładowania paczek) (Łukasz Jackowski)

4.3 Plan trzeciej iteracji - 17.08.13 - 17.09.13

- Wybór technologii umożliwiającej tworzenie rozszerzeń (JSPF, JPF, OSGi, własne rozwiązanie) (Grzegorz Kaczmarczyk)
- Określenie możliwości oferowanych przez API (Grzegorz Kaczmarczyk)
- Przykładowe rozszerzenie umożliwiające komunikację (np. przez zwykłe nieszyfrowane połączenie) (Łukasz Jackowski)
- Możliwość zakładania kont użytkowników na serwerze (Łukasz Jackowski)
- Organizacja wrzucanych paczek w projekty oraz możliwość pobierania przetłumaczonych paczek(Łukasz Jackowski)

4.4 Plan czwartej iteracji - 17.09.13 - 17.10.13

- Stworzenie interaktywnego interfejsu tekstowego do obsługi aplikacji klienckiej (Grzegorz Kaczmarczyk)
- Zaimplementowanie wymagań dotyczących skanera i generatora paczek (Grzegorz Kaczmarczyk)
- Ulepszenia w aplikacji serwerowej - możliwość ręcznego załadowania paczki przetłumaczonej i zmiany statusu tłumaczenia zamówienia (Łukasz Jackowski)
- Praca nad stroną wizualną aplikacji serwerowej (Łukasz Jackowski)

4.5 Plan piątej iteracji - 17.10.13 - 17.11.13

- Stworzenie edytora plików XLIFF w języku Java Script (Łukasz Jackowski)
- Stworzenie konwertera plików XLIFF do STAB (Łukasz Jackowski)
- Interfejs graficzny dla STAT (Grzegorz Kaczmarczyk)
- Dodanie rewizji (Łukasz Jackowski, Grzegorz Kaczmarczyk)

4.6 Plan szóstej iteracji - 17.11.13 - 17.12.13

- Dalsze prace nad interfejsem graficznym (Grzegorz Kaczmarczyk)
- Repozytorium wtyczek (pluginów)(Grzegorz Kaczmarczyk)
- Przebudowa API rozszerzeń (Łukasz Jackowski, Grzegorz Kaczmarczyk)
- Wtyczka do Maven-a (Grzegorz Kaczmarczyk)

5 Raporty z iteracji

5.1 Iteracja 1

Głównym celem pierwszej iteracji było zdefiniowanie struktury formatu używanego do przekazywania plików będących przedmiotem tłumaczenia oraz odbierania wyniku. Struktura przykładowej paczki:

```
|-bundle.xml
|-refs
| |-fr_FR
| | |-msgs.properties
| |-pl_PL
|   |-msgs.properties
|-source
|   |-msgs.properties
|-target
```

Deskryptor projektu

Z każdą paczką stowarzyszony jest jej deskryptor - plik opisujący strukturę paczki oraz podstawowe dane o niej, takie jak identyfikatory projektu, zamówień oraz klienta. Co istotne - są one generowane po stronie klienta w celu uproszczenia protokołu komunikacyjnego.

Plik z opisem projektu (bundle.xml):

```
<?xml version="1.0" encoding="UTF-8"?>
<bundle
    xmlns="http://student.agh.edu.pl/~gaczm/stab/v0.1/"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://student.agh.edu.pl/~gaczm/stab/v0.1/␣␣␣␣␣␣http://student.ag
    <projectId>cd7fd87a-7b33-430b-b256-02477209f7e7</projectId>
    <orderId>4054233e-6351-41c3-99ae-beb4e6d223ea</orderId>
    <clientId>8980ce29-258d-4a90-9c68-8064b0ae291a</clientId>

    <assets>
        <!-- Plik z paczki -->
        <asset file="/resources/translations.properties" type="java-properties">
```

```
            <!-- Kod jezyka (tzw. language tag) -->
            <sourceLanguage>en_US</sourceLanguage>
            <targetLanguage>pl_PL</targetLanguage>
            <targetLanguage>fr_FR</targetLanguage>
        </asset>

        <!-- Plik zdalny -->
        <asset file="http://example.com/legal/licence.txt" type="plaintext">
            <sourceLanguage>en_US</sourceLanguage>
            <targetLanguage>fr_FR</targetLanguage>
        </asset>
    </assets>
</bundle>
```

Pojęcia:

clientId - Unikalne ID klienta zgłaszającego chęć przetłumaczenia aplikacji. W zakładanym przez nas scenariuszu klient zgłasza się do biura tłumaczeń, wypełnia formularz swoimi danymi i w odpowiedzi dostaje przydzielone mu ID. ID klientów muszą być unikalne w obrębie jednego biura, między różnymi biurami już niekoniecznie, choć w praktyce można założyć że będą.

projectId - Unikalne ID projektu. Przez projekt rozumiemy projekt tłumaczeniowy dotyczący pojedynczej aplikacji.

orderId - Unikalne ID zgłoszenia (paczki). Unikalne w obrębie pojedynczego projektu.

Rozróżnienie między *projectId* oraz *orderId* wprowadzane jest w celu ułatwienia i kategoryzowania przez biuro zasobów takich jak pamięci tłumaczeniowe czy glosariusze.

5.1.1 Aktualna postać aplikacji klienckiej

W aktualnej wersji, aplikacja kliencka umożliwia wygenerowanie paczki na podstawie plików w bieżącym katalogu. Odnalezione zostaną wszystkie pliki z rozszerzeniem *.properties, tworzony jest plik bundle.xml zgodnie z powyższymi wytycznymi, przygotowana struktura katalogów i całość umieszczona w jednym archiwum ZIP.

Użycie ogranicza się do wpisania komendy:

```
java -jar target/ClientApp-1.0-SNAPSHOT-jar-with-dependencies.jar -g
```

W jej wyniku w bieżącym katalogu zostanie utworzony plik `bundle.zip` zawierający rzeczoną paczkę.

5.1.2 Napotkane problemy

Problemy napotkane w trakcie tej iteracji koncentrowały się głównie na stworzeniu schematu XML zgodnego z normami. Wynika to z braku wcześniejszego doświadczenia przy tego typu pracach.

5.2 Iteracja 2

Celem drugiej iteracji było utworzenie wstępnej wersji serwera imitującego aplikację używaną przez biuro tłumaczeń do odbierania od klientów paczek.

5.2.1 Wizja

Wedle naszych założeń nasz serwer ma oferować możliwość zakładania kont klienckich, w ramach których zgromadzone byłyby projekty. Skaner po utworzeniu paczki będzie wysyłał ją do serwera i odbierał od niego przetłumaczone pliki. Ponadto chcemy udostępnić możliwość konwersji paczek do formatu XLIFF oraz prosty edytor tego formatu.

W tej iteracji skupiamy się na wyborze odpowiednich technologii, utworzeniu konwertera oraz umożliwieniu konwersji załadowanych paczek.

5.2.2 Wybór technologii dla aplikacji serwerowej

Stając przed zadaniem stworzenia serwera musieliśmy podjąć decyzję z jakich technologii skorzystać. Ilość dostępnych w świecie Javy frameworków do tworzenia aplikacji webowych jest dość pokaźna. Wymieniając tylko niektóre z nich: JSP, JSF, Apache Struts, Apache Wicket, Spring MVC, Spark, Play!. W celu dokonania w pełni świadomego wyboru musielibyśmy dobrze poznać każdy z nich, co jest w praktyce niezbyt możliwe. W związku z tym musieliśmy zdać się na dostępne w Internecie porównania oraz własne preferencje.

Pierwszym mocno kojarzącym się z Javą frameworkiem jest JSP. W swoich założeniach przypomina PHP: kod strony HTML przeplatany jest kodem Javy. Rozwiązanie jest bardzo proste w użyciu, jednak już dość mocno zdeprecjonowane i nie polecane. Inną również mocno kojarzoną z Javą technologią jest JSF. Stosuje on podejście, w którym strony internetowe buduje się z komponentów - faceletów. Strony opisywane są w formacie xhtml, przy czym pojedyncze tagi (np. <h:outputText />) są zamieniane przez JSF na właściwe tagi HTML. Problemem frameworku jest jego nadmierne skomplikowanie oraz utrudnianie niektórych wydawałoby się prostych rzeczy (np ciężko jest zastosować własne style CSS lub choćby narzędzia takie jak Twitter Bootstrap). Koncepcyjnie podobny jest Apache Wicket. Spark jest bardzo prostym frameworkiem wzorowanym na Sinatrze. Wprawdzie jego możliwości byłyby wystarczające do naszych zastosowań, niemniej uznaliśmy że użycie go byłoby dla nas mało rozwijające. Pozostały nam zatem do wyboru Spring MVC oraz Play!. Pomimo sympatii do rozwiązań spod szyldu Spring-a zdecydowaliśmy się na użycie mocno zyskującego na popularności frameworku Play!.

Play! jest relatywnie młodym, bezstanowym frameworkiem MVC. Można używać go zarówno z poziomu Javy jak Scali. Używany jest m.in. przez portal LinkedIn.

5.2.3 Stworzenie szkieletu serwera

Technologia w jakiej zdecydowaliśmy się realizować aplikację serwerową była nam zupełnie nieznana. Fakt ten oraz brak dużego doświadczenie w tworzeniu aplikacji webowych sprawił nam na początku nieco trudności zarówno ze skonfigurowaniem projektu jak i spowodował pewne błędne decyzje projektowe, które potem skutkowały koniecznością wprowadzania poprawek i przeróbek.

Na tym etapie serwer umożliwia jedynie ręczne wrzucanie archiwów zip (bez sprawdzania zgodności ze zdefiniowanym w poprzedniej iteracji formatem) oraz wyświetlanie ich nazw. Aplikacja została zintegrowana z frameworkiem Spring natomiast wszelkie dane przechowuje w pamięci.

5.2.4 Konwersja do XLIFF

Zanim przedstawimy sposób konwersji między paczkami (zwanymi STAB, od Software Translation Automation Bundle) a formatem XLIFF przedstawmy pokrótce podobieństwa i różnice między nimi.

5.2.5 XLIFF

Format XLIFF służy do przechowywania treści tłumaczonych dokumentów, oraz różnego rodzaju dodatkowych informacji. Treści przechowywane są w nim w formie już zsegmentowanej. Przykładowy dokument XLIFF wygląda następująco:

```
<xliff version="1.2">
   <file
      original="foo.txt"
      source-language="pl_PL" target-language="en_EN">
      <body>
         <trans-unit id="1" maxbytes="14">
            <source xml:lang="pl\_PL">Ala ma kota.</source>
            <target xml:lang="en\_EN">Ala has a cat.</target>
         </trans-unit>
         <trans-unit id="3" maxbytes="114">
            <source xml:lang="pl\_PL">Kot ma Ale.</source>
            <target xml:lang="en\_EN">Cat has Ala.</target>
         </trans-unit>
      </body>
   </file>
</xliff>
```

Element <file> zawiera w sobie kolejne segmenty (<trans-unit>) pochodzące z danego pliku. Ponadto możliwe jest umieszczanie wielu dodatkowych informacji: notek dla tłumacza, informacji pomocnych dla narzędzia CAT, informacji o narzędziu tworzącym dokument i inne. Narzędzia CAT takie jak MemoQ, SDL Trados czy XTM umieszczają w plikach wiele dodatkowych i niestandardowych informacji (robią to jednak zgodnie ze specyfikacją formatu, format umożliwia bowiem wiele miejsc rozszerzeń). Pełna specyfikacja formatu znajduje się tutaj.

5.2.6 STAB

Format STAB z kolei skupia się na organizowaniu i przesyłaniu całych plików przeznaczonych do tłumaczenia. Istotne jest w nim zachowanie oryginalnej struktury katalogów w których znajdowały się pliki. Sam format poza plikami zawiera dodatkowo deskryptor z informacjami na temat zlecanego zadania.

STAB jest formatem prostym - zawiera pliki określonych typów, zebrane w zdefiniowanej przez nas strukturze i opisane odpowiednim deskryptorem. XLIFF natomiast jest dokumentem w zasadzie gotowym do tłumaczenia. Aby tłumacz mógł przystąpić do pracy potrzebne jest narzędzie CAT które dokona odpowiednich analiz i będzie asystowało tłumaczowi w jego pracy. Niemniej musimy pamiętać, że formaty używane do lokalizacji aplikacji (pliki .properties, .po i inne tym podobne) pomimo różnej struktury sprowadzają się w gruncie rzeczy do idei plików typu klucz wartość. Nie potrzebujemy zatem dodatkowej segmentacji, co ułatwia konwersję. Przy konwersji stosujemy następującą konwencję: elementy <asset> mapowane są na elementy <file>. Elementy <trans-unit> tworzone są z oryginalnego klucza (element <note>), wartości oryginalnej (<source>) i ew wartości przetłumaczonej (pochodzącej z katalogu refs, element <target>).

5.2.7 Przykład konwersji

Struktura paczki:

```
|-bundle.xml
|-refs
| |-fr_FR
| | |-locals
| |   |-licence.txt
| |-pl_PL
|   |-locals
|     |-msgs.propertiess
|-source
| |-locals
|   |-msgs.properties
|-target
```

bundle.xml:

```
<bundle>
    <clientId>ddf3a240-c7d1-11e2-8b8b-0800200c9a66</clientId>
    <projectId>cd7fd87a-7b33-430b-b256-02477209f7e7</projectId>
    <orderId>4054233e-6351-41c3-99ae-beb4e6d223ea</orderId>

    <assets>
        <asset file="/locals/msgs.properties" type="java-properties">
            <sourceLanguage>en_US</sourceLanguage>
            <targetLanguage>pl_PL</targetLanguage>
            <targetLanguage>fr_FR</targetLanguage>
        </asset>
    </assets>
</bundle>
```

```
   /locals/msgs.properties:

some.error=Some error occurred
some.info=Something good happend
```

Wynikowy XLIFF:

```
<xliff>
   <sta:info>
      <sta:clientId>ddf3a240-c7d1-11e2-8b8b-0800200c9a66</sta:clientId>
      <sta:projectId>cd7fd87a-7b33-430b-b256-02477209f7e7</sta:projectId>
      <sta:orderId>4054233e-6351-41c3-99ae-beb4e6d223ea</sta:orderId>
   </sta:info>
   <file original="locals/msgs.properties"
         source-language="en_US"
         target-language="pl_PL"
         datatype="plaintext">
      <header>
         <tool tool-id="stab2xliff"
               tool-name="stab2xliff"
               tool-version="0.1"
               tool-company="Grzegorz␣Kaczmarczyk,␣Lukasz␣Jackowski"/>
      </header>
      <body>
         <trans-unit id="0">
            <note>Key name: "some.error"</note>
            <source xml:lang="en_US">Some error occurred</source>
            <target xml:lang="pl_PL"></target>
         </trans-unit>
         <trans-unit id="1">
            <note>Key name: "some.info"</note>
            <source xml:lang="en_US">Something good happend</source>
            <target xml:lang="pl_PL"></target>
         </trans-unit>
      </body>
   </file>
   <file original="locals/msgs.properties"
         source-language="en_US"
         target-language="fr_FR"
         datatype="plaintext">
      <header>
         <tool tool-id="stab2xliff"
               tool-name="stab2xliff"
               tool-version="0.1"
               tool-company="Grzegorz␣Kaczmarczyk,␣Lukasz␣Jackowski"/>
      </header>
      <body>
         <trans-unit id="0">
            <note>Key name: "some.error"</note>
            <source xml:lang="en_US">Some error occurred</source>
            <target xml:lang="fr_FR"></target>
         </trans-unit>
         <trans-unit id="1">
            <note>Key name: "some.info"</note>
            <source xml:lang="en_US">Something good happend</source>
            <target xml:lang="fr_FR"></target>
         </trans-unit>
      </body>
   </file>
</xliff>
```

5.3 Iteracja 3

Iteracja trzecia miała dwa oddzielne cele: API do przesyłania danych między aplikacją kliencką a serwerową, a także rozbudowa aplikacji serwerowej.

5.3.1 API do przesyłania danych

Zgodnie z naszą ideą, dostarczana przez nas ąplikacja kliencka", powinna być raczej traktowana jako podwaliny pod budowę przez odpowiedni podmiot własnej, dopasowanej do swoich możliwości i wymagań aplikacji. Ponieważ tworzone przez aplikację kliencką paczki chcemy z reguły przesłać gdzieś dalej, musimy zapewnić do tego odpowiednią warstwę transportową. Niemniej, jako że nie tworzymy aplikacji klienckiej z myślą o integracji z jakąś inną konkretną aplikacją, potrzebujemy API które pozwoli nam w prosty sposób taką integrację zapewnić.

5.3.2 Pseudokod interfejsu

```
interface TransportApi:
    void upload(InputStream bundle, BundleInfo info)
    Status status(BundleInfo info)
    OutputStream download(BundleInfo info)

class BundleInfo:
    UUID projectId, clientId, orderId

enum Status:
    NOT_STARTED, IN_PROGRESS, READY
    Map<String, String> extraInfo()
```

Jak widać jest on maksymalnie prosty. Twórca rozszerzenia zajmuje się wyłącznie implementacją warstwy transportowej i nie potrzebuje wnikać w resztę kodu aplikacji.

5.3.3 Wybór technologii

W tym miejscu powstaje pytanie: z jakiego rozwiązania (jeśli w ogóle) skorzystać, aby tworzenie odpowiednich rozszerzeń było jak najprostsze?

Po poszukiwaniach, wytypowaliśmy trzy możliwości:

- Java Plugin Framework
- Apache Felix (implementacja OSGi)
- Java Simple Plugin Framework

Pierwszy z nich można bez wątpienia nazwać technologią dojrzałą. Równocześnie jednak jest też technologią lekko przestarzałą (ostatnie zmiany w projekcie sięgają 2007 roku). Framework oferuje wiele możliwości, idzie to jednak w parze ze skomplikowaniem. Tworzenie pluginów wymaga odpowiedniej struktury aplikacji oraz tworzenia specjalnych deskryptorów w XML-u, przez co całość wygląda nieco zniechęcająco.

Kolejna możliwość, czyli Apache Felix jest minimalistyczną implementacją framework-a OSGi. OSGi samo w sobie nie jest wprawdzie technologią do tworzenia pluginów, nie mniej świetnie nadaje się do tego celu (m.in. rozszerzenia dla środowiska Eclipse bazują na OSGi, konkretnie na własnej implementacji Equinox). Dodatkowo projekt jest aktywnie rozwijany i udokumentowany. Niemniej podobnie jak w przypadku JPF konieczne jest tworzenie dodatkowych plików (manifestów) opisujących bundle.

Następnym rozwiązaniem, na które zwróciliśmy uwagę jest JSPF. Zgodnie z nazwą framework jest naprawdę prosty w użyciu. Ze strony osoby udostępniającej możliwość tworzenia wtyczek w swojej aplikacji, jego użycie praktycznie ogranicza się do zdefiniowania interfejsu który będzie implementowany. Następnie, w samym kodzie aplikacji należy jedynie zainstancjonować rozszerzenie przy pomocy tego typu kodu:

```
PluginManager pluginManager = PluginManagerFactory.createPluginManager();
pluginManager.addPluginsFrom(new URI("path/to/plugin.jar"));
MyPlugin plugin = pluginManager.getPlugin(MyPlugin.class);
```

Co istotne, ścieżka do pluginu może być dowolnym, poprawnym URI, w szczególności adresem internetowym.

Biorąc pod uwagę wady i zalety poszczególnych rozwiązań zdecydowaliśmy się na ostatnie z nich.

5.3.4 Tworzenie rozszerzenia

API które należy zaimplementować znajduje się w projekcie TransportApi. Do zaimplementowania jest w nim jeden interfejs (TransportApi) z trzema metodami:

```
public interface TransportApi extends Plugin {
    void setArguments(Map<String, String> arguments);
    void upload(InputStream bundle, BundleInfo info);
    Status status(BundleInfo info);
    OutputStream download(BundleInfo info);
}
```

Dodatkowo przygotowaliśmy archetyp Maven-a upraszczający tworzenie wtyczki.
Zatem proces tworzenia wygląda następująco:

1. Należy wygenerować projekt korzystając z przygotowanego archetypu:

```
PluginManager pluginManager = PluginManagerFactory.createPluginManager();
pluginManager.addPluginsFrom(new URI("path/to/plugin.jar"));
MyPlugin plugin = pluginManager.getPlugin(MyPlugin.class);
```

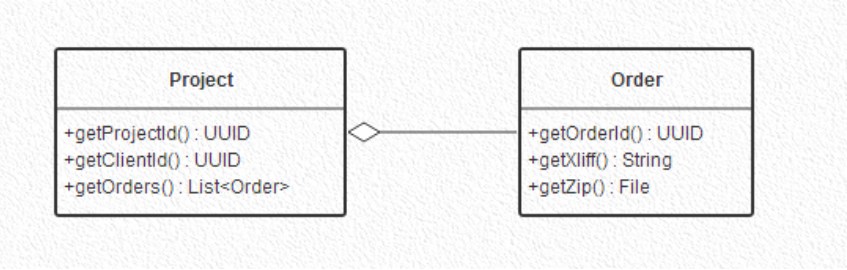
2. Zaimplementować interfejs z uwzględnieniem wymagań danego systemu
3. Zbudować archiwum korzystając z komendy:

```
mvn package
```

4. Umieścić powstałe archiwum (z suffiksem: `-jar-with-dependencies`) w dostępnym miejscu.
5. W konfiguracji skanera dodać konfigurację wtyczki (ścieżkę do niej oraz dodatkowe, wymagane argumenty)

5.3.5 Zarys modelu

Model używany w aplikacji serwerowej nie jest skomplikowany. Naszym celem jest posiadanie zamówień zorganizowanych w projekty wg schematu :

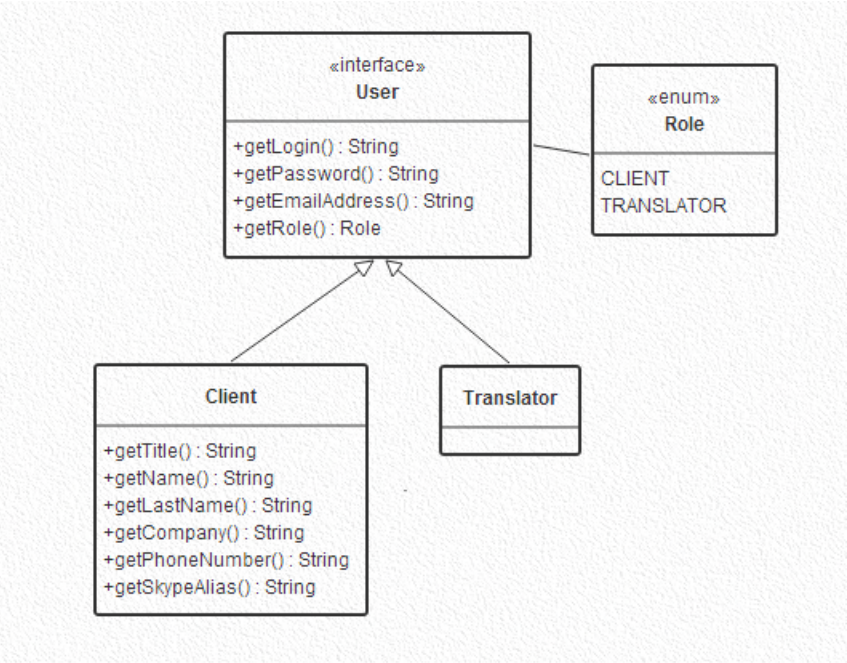


Każde zamówienie przechowuje wejściowe archiwum (na podstawie którego, możliwa będzie potem konwersja do formatu STAB) oraz string reprezentujący plik XLIFF. Pomimo, że plik XLIFF ma określoną strukturę to wygodne dla nas jest reprezentowanie w postaci stringa z uwagi na to, że wszystkie akcje na nim wykonywane będą z poziomu języka javascript, który dostarczy wszelkich niezbędnych operacji oraz prześle potem wyedytowany odpowiednio napis do serwera.

5.3.6 Rejestracja użytkowników

Na tym etapie prac zdaliśmy sobie sprawę, że czym innym jest użytkownik mający dostęp do naszej aplikacji z poziomu przeglądarki, a czym innym użytkownik korzystający z naszego API poprzez mechanizm wtyczek. Pierwsza grupa to klienci, którzy korzystają aplikacji klienckiej i jednocześnie dostarczający zleceń wizualizowanych i obsługiwanych przez aplikację serwerową. Takim użytkownikom nie dajemy możliwości przeglądania projektów oraz ich edycji - mają dostęp do serwera tylko i wyłącznie poprzez REST API. Drugą grupą są tłumacze, którzy na co dzień korzystają z aplikacji CAT'owskich (których to zachowanie ma symulować nasz serwer). Mogą oni

edytować tłumaczenia oraz zmieniać ich statusy. Z rozróżnienia tych dwóch grup powstały role jakie nadajemy określonym grupom w systemie :



5.3.7 API udostępniane przez serwer

Framework Play! zbudowany jest na bazie podejścia RESTful, dlatego też z samej swojej natury udostępnia odpowiednie metody, które mogą być wołane tak z wewnątrz jak i z zewnątrz aplikacji. Na tym etapie konieczne było zrealizowanie podstawowej implementacji wtyczki. Nasza wtyczka korzysta z HTTPClient'a na licencji Apache i wysyła proste żądania Http. Serwer udostępnia następujące API :

```
GET /download/:projectId/:orderId
GET /status
POST /upload
```

Na obecnym etapie możliwe jest wykonanie jakichkolwiek zapytań bez uprzedniej autentykacji.

5.4 Iteracja 4

W czwartej iteracji skupiliśmy się na rozbudowie aplikacji klienckiej (dostarczenie większości zakładanej funkcjonalności, ale tylko z interfejsem tekstowym), oraz rozbudowie aplikacji serwerowej tak, aby możliwe było przetestowanie odbierania rezultatów z serwera. Dodatkowo popracowaliśmy nad samym wyglądem aplikacji.

5.4.1 Interfejs aplikacji STAT

Pracując nad interfejsem aplikacji wzorowaliśmy się na programie Git. Ogólne użycie STAT, wygląda następująco:

```
stat <KOMENDA> [OPCJE...]
```

Na przykład :

```
stat add -s en -t pl,fr,it src/main/resources/messages.properties
```

Niemniej, proces stworzenia projektu wymaga w naszym przypadku zapytania użytkownika o kilka rzeczy, w związku z czym potrzebujemy do tego kreatora. W tej iteracji zajęliśmy się jedynie kreatorem działającym w trybie tekstowym, w późniejszym czasie ma się pojawić również konfigurator graficzny.

Przykładowa sesja konfiguracji wygląda następująco:

```
>  ./stat init -c
Select one of predefined patterns, or provide custom one
 1. Java properties (*.properties files)
 2. GNU gettext (*.po files)
 3. Custom pattern (advanced)
 > 3
 > **/src/main/resources/**/*.properties

Review list of found assets, and customize it if you need.
 1. src/main/resources/pl/edu/agh/messages.properties ? ->  ?
 2. src/main/resources/pl/edu/agh/messages_en.properties ? ->  ?
 3. src/main/resources/pl/edu/agh/messages_pl.properties ? ->  ?
Possible commands:
add  - add new file to list. E.g: "add ../foo/bar.txt"
rm   - remove files from list. E.g: "rm 1 3 8"
next - go to next step
 > rm 2-3
 1. src/main/resources/pl/edu/agh/messages.properties ? ->  ?
 >

Specify source languages for files (default: 1)
 1. Specify for all files
 2. Specify per file
 >
Source language:
 > en

Specify target languages for files (default: 1)
 1. Specify for all files
 2. Specify per file
 >
Target language:
 > en pl fr de ru

Specify encoding for assets (default: 1)
 1. Specify for all files
 2. Specify per file
 >
Provide encoding:
 > utf-8

Provide expression that describes installation path for translated files. Default: ${sourc
You can use following variables:
   ${lang}             - language code (e.g: en)
   ${source_file_path} - path to source file (e.g: "app/resources/messages")
   ${file_name}        - file name without extension (e.g: "messages")
   ${file_extension}   - file extension (e.g: "properties")
As file path separator symbol use: /
 >

Configuration completed!
```

Więcej szczegółów na temat użycia można znaleźć w dokumentacji użytkownika.

5.4.2 Ulepszenia w aplikacji serwerowej

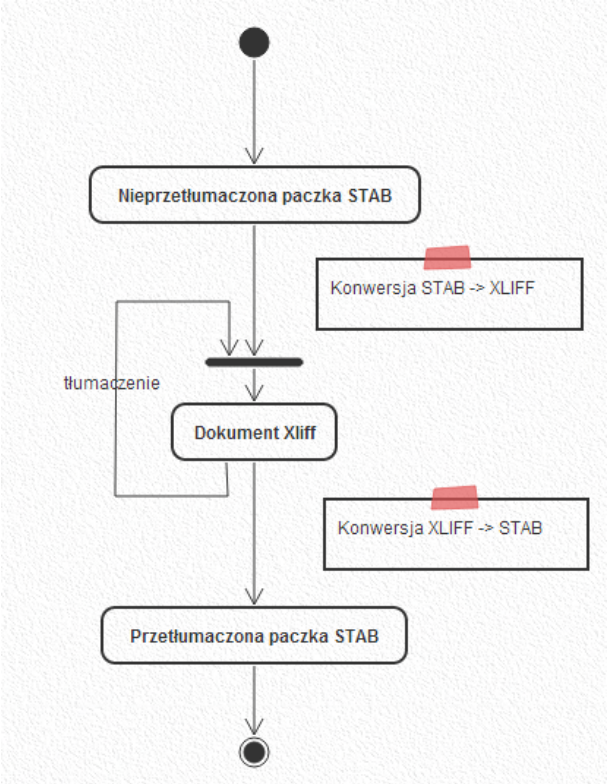
Najbardziej widoczną zmianą w aplikacji serwerowej było użycie Semantic UI. Jest to framework CSS ułatwiający tworzenie czytelnych, dobrze wyglądających i zgodnych ze standardami stron internetowych, podobny do znacznie

popularniejszego projektu Twitter Bootstrap.

Ponadto, w celu umożliwienia przetestowania współpracy obu aplikacji, dodano możliwość uploadu przetłumaczonej paczki (docelowo tłumaczenie odbywać się będzie bezpośrednio w aplikacji z użyciem edytora XLIFF).

5.5 Iteracja 5

W tej iteracji głównym zadaniem było skupienie się na procesie przetwarzania dokumentu xliff. Jego proces przedstawia się następująco :



Przetwarzanie dokumentu xliff to sztandarowe zadanie stworzonego przez nas serwera. Zgodnie z powyższym diagramem proces ten składa się z trzech kroków :

1. Konwersja z formatu STAB do formatu XLIFF
2. Tłumaczenie (z pomocą Edytora)
3. Konwersja z formatu XLIFF do formatu STAB

5.5.1 Edytor XLIFF

Zgodnie z założeniami całością edycji pliku XLIFF miał zająć się komponent napisany w języku javascript. Jego użycie wygląda następująco :

1. Definiujemy na stronie element, wewnętrz którego ma zostać osadzony edytor :

```
...
<div class="xliff editor">

</div>
...
```

2. Dostarczamy odpowiednich danych w obiekcie reprezentującym parametry edytora (prosty obiekt z czterema polami jak wyżej)

```
...
<script>
   $.ready(function() {
     data={content:" <xlf>test</xlf>",selector:".xliff.editor",toAddress:"/my/add
 ...
})
</script>
...
```

3. Wywołujemy odpowiednią funkcję :

```
...
<script>
   $.ready(function() {
     data=....
     xliffEditor(data);
   })
</script>
...
```

Dokładniejszy opis znajduje się w dokumentacji technicznej serwera. Edytor dostaje od serwera zwykłego stringa oraz parsuje go do obiektu xml. Z tego obiektu wyodrębnia tagi '<trans-unit>' i na ich bazie tworzy tabelkę zawierającą klucze oraz pola, w których można edytować wartość odpowiedniego klucza. Wciśnięcie przycisku 'Save' powoduje zapisanie obecnego stanu dokumentu w bazie.

5.5.2 Konwerter XLIFF

Z uwagi na to, że bazując wyłącznie na informacjach zapisanych w dokumencie XLIFF nie bylibyśmy w stanie odtworzyć wszystkich danych niezbędnych do utworzenia wynikowej paczki niezbędne jest zapamiętanie paczki bazowej (którą otrzymaliśmy w trakcie składanie zlecenia przez podmiot zewnętrzny). Posiadając dane na temat struktury paczki wejściowej, wzbogacone o informacje uzyskane w trakcie procesu tłumaczeniowego jesteśmy w stanie stworzyć paczkę w formacie STAB będącą gotową do zainstalowania w aplikacji klienckiej. Wszelkie pliki będące wynikiem tego procesu znajdują się w katalogu "target" w strukturze jak poniżej :

```
|-bundle.xml
|-refs
| |-fr_FR
| | |-msgs.properties
| |-pl_PL
|   |-msgs.properties
|-source
|   |-msgs.properties
|-target
```

5.5.3 Interfejs graficzny STAT

Kolejną wprowadzoną zmianą było dodanie (wczesnej wersji) graficznego konfiguratora projektu dla STAT.

Poza rzeczami widocznymi dla użytkownika, staraliśmy się napisać kod w taki sposób aby w miare możliwości uniknąć powtórzeń między kodem kreatora graficznego i kreatora tekstowego.

Więcej szczegółów na temat samego kreatora znajduje się w dokumentacji użytkownika aplikacji klienckiej.

5.5.4 Dodanie rewizji

Numer rewizji jest kolejnym atrybutem identyfikującym paczkę. Oddaje ona stan, w którym klient wysłał paczkę do serwera, ale nie otrzymawszy jeszcze przetłumaczonej wersji chciał jeszcze coś do niej dołączyć. Wtedy nie generowana jest paczka z innym numerem zamówienia (orderId) a zwiększana jest rewizja w ramach tego samego zamówienia.

Gdyby nasz serwer był prawdziwą aplikacją CAT powinien takie zamówienia (o tej samej rewizji) w jakiś sposób scalić. Takie zamówienie byłoby jedynie uaktualnieniem paczki. Ponieważ nie przewidywaliśmy takiej funkcjonalności serwer przechowuje paczki o wszystkich rewizjach, a na żądanie pobrania tłumaczenia zwraca przetłumaczone zamówienie (status 'FINISHED') o najwyższym numerze rewizji.

5.6 Iteracja 6

5.6.1 Repozytorium pluginów

Aby uniknąć ręcznego wpisywania URI wtyczki i uprościć dystrybucję wtyczek, stworzyliśmy ich repozytorium. Pod adresem "http://stat.repo:9000/list" udostępniana jest lista dostępnych pluginów w formie dokumentu JSON. Przykładowo:

```
[
   {
      "conf": "Url,Username,Password",
      "name": "product",
      "path": "vendor_1/product/1.0/plugin.jar",
      "vendor": "vendor_1",
      "version": "1.0"
   },
   {
      "name": "test",
      "path": "test/test/1.0/plugin.jar",
      "vendor": "test",
      "version": "1.0"
   }
]
```

Lista generowana jest automatycznie na podstawie plików znalezionych w katalogu roboczym repozytorium. Pliki zorganizowane są w katalogach o strukturze:

```
- Twórca
| |- Nazwa
|    |- Wersja
|       |- plugin.jar
|       |- plugin.desc
- Twórca
| |- Nazwa
|    |- Wersja
|       |- plugin.jar
|       |- plugin.desc
- Twórca
| |- Nazwa
|    |- Wersja
|       |- plugin.jar
|       |- plugin.desc
```

Plik "plugin.desc" jest w formacie pliku properties i można umieścić w nim dowolne dane, np:

```
jakas opcja=123
```

Zostanie ona dodana wtedy do listy informacji:

```
{
        "conf": "Url,Username,Password",
        "name": "product",
        "path": "vendor_1/product/1.0/plugin.jar",
        "vendor": "vendor_1",
        "version": "1.0",
        "jakas opcja": "123"
}
```

Parametr ćonfókreśla opcje konfiguracyjne które mają zostać przekazane do wtyczki w celu umożliwienia jej działania. Opcje oddziela się przecinkiem. W przypadku powyżej, wtyczka wymaga podania trzech parametrów: "Url", "Username"i "Password".

5.6.2 Przebudowa API rozszerzeń

Początkowo zakładaliśmy że API służyć będzie tylko do przesyłania paczek (początkowo nosiło nazwę TransportAPI). Nie mniej, po rozmowach z klientem uznaliśmy że powinno pełnić nieco więcej funkcji, w związku z czym przemianowaliśmy je na Integration API. Więcej informacji na temat ostatecznego wyglądu API można znaleźć w dokumentacji technicznej API.

5.6.3 Wymagania dotyczące danych klienta

Jednym z głównych założeń oraz wymagań stawianych przed aplikacją była maksymalna prostota protokołu wymiany informacji, który w minimalnym stopniu będzie angażowała potencjalnego użytkownika. Stąd początkowo wszelkie dane (projectId, orderId oraz clientId) miały dziać się po stronie klienta, a numer UUID miał zapewnić ich unikatowość. Jednak w momencie skonfrontowania tego pomysłu z rzeczywistymi potrzebami okazało się, że lepiej będzie wprowadzić mechanizm autoryzacji klientów. I tak podczas pierwszego użycia klient wprowadza swoje dane i zakłada wraz z projektem konto na serwerze, a informację o danych niezbędnych do logowania przesyłana jest na podany adres e-mail. Na tym etapie wynikła też potrzeba klienta odnośnie zlecania wyceny - klient nie musi się decydować na usługę, ale chce się zorientować ile można ona kosztować. Wszelkie podobne informacje otrzymuje on na swoje konto mailowe.

Powyższe zmiany wymagań pociągnęły za sobą gruntowną przebudowę IntegrationAPI, a więc i wtyczki do naszej aplikacji serwerowej. Całość opisana jest w dziale IntegrationAPI.

5.6.4 Wtyczka do Maven-a

W ramach iteracji powstała również wtyczka do Maven-a umożliwiająca wykonanie podstawowych operacji na projekcie. Więcej informacji w dokumentacji użytkownika wtyczki.

6 Testy

W ramach dbania o jakość tworzonego produktu:

- pisaliśmy testy jednostkowe sprawdzające działanie wycinków funkcjonalności
- pisaliśmy testy integracyjne sprawdzające działanie różnych elementów ze sobą
- korzystaliśmy z systemu Continuous Integration (https://drone.io/bitbucket.org/gaczm/inzynierka)
- wykorzystywaliśmy narzędzie Sonar w celu znajdywania potencjalnych błędów

6.1 Raporty pokrycia

Poniżej znajdują się raporty pokrycia testami jednostkowymi dla projektów STAT i konwertera stab2xliff i xliff2stab. Raporty zostały przygotowane przy pomocy narzędzia wbudowanego w środowisko IntelliJ IDEA.

6.1.1 STAT

Package	Class	Method	Line
pl.edu.agh.settings.v1	100% (2/ 2)	89.3% (25/ 28)	82.8% (48/ 58)
pl.edu.agh.integration	100% (3/ 3)	87% (20/ 23)	85.5% (65/ 76)
pl.edu.agh.wizard.cli	100% (11/ 11)	82.4% (42/ 51)	60.1% (184/ 306)
pl.edu.agh.settings	100% (1/ 1)	80% (8/ 10)	66.7% (20/ 30)
pl.edu.agh.wizard	100% (1/ 1)	66.7% (2/ 3)	81.8% (9/ 11)
pl.edu.agh.commands	100% (6/ 6)	60% (12/ 20)	50.6% (41/ 81)
pl.edu.agh.utils.cli	100% (2/ 2)	100% (10/ 10)	95.6% (43/ 45)
pl.edu.agh.generator	100% (8/ 8)	92.8% (64/ 69)	94.4% (236/ 250)
pl.edu.agh.utils	92.9% (13/ 14)	73.9% (65/ 88)	89.7% (541/ 603)
pl.edu.agh.wizard.gui.textview	0% (0/ 1)	0% (0/ 5)	0% (0/ 14)
pl.edu.agh.wizard.gui.form	0% (0/ 5)	0% (0/ 32)	0% (0/ 87)
pl.edu.agh.wizard.gui	0% (0/ 22)	0% (0/ 137)	0% (0/ 555)

Pakiet `gui` nie zawiera logiki, a jedynie tworzenie okien, formularzy itd i nie był testowany, stąd brak pokrycia.

6.1.2 Konwertery

Package	Class	Method	Line
pl.edu.agh.common.helper	100% (3/ 3)	63.2% (12/ 19)	80.4% (86/ 107)
pl.edu.agh.common.model.data	100% (23/ 23)	70.4% (69/ 98)	76.2% (125/ 164)
pl.edu.agh.common.model.exception	100% (1/ 1)	100% (1/ 1)	100% (2/ 2)
pl.edu.agh.common.utils	100% (1/ 1)	83.3% (5/ 6)	68.2% (15/ 22)
pl.edu.agh.stab2xliff.analyzer.converters	100% (7/ 7)	80.6% (29/ 36)	74.8% (77/ 103)
pl.edu.agh.stab2xliff.analyzer.exceptions	100% (1/ 1)	100% (1/ 1)	100% (2/ 2)
pl.edu.agh.stab2xliff.exceptions	100% (1/ 1)	50% (1/ 2)	50% (2/ 4)
pl.edu.agh.stab2xliff.utils	100% (1/ 1)	42.9% (3/ 7)	45.5% (5/ 11)

6.2 Testy akceptacyjne

Kluczowym zagadnieniem mającym na celu poprawę jakości naszej aplikacji było stworzenie testów akceptacyjnych. Ze względu na to że w ramach usługi współpracuje ze sobą kilka narzędzi, wykonane zostały w formie testów integracyjnych. Testami pokryte zostały obszary:

- integracja wtyczki ze stroną serwerową
- działanie aplikacji klienckiej jako całości (w tym współpraca z wtyczkami)

6.2.1 Integracja wtyczki ze stroną serwerową

Napisane przez nas w tej części testy integracyjne zakładają istnienie uruchomionej instancji bazy danych MongoDB pod adresem 127.0.0.1 i portem 27017. Do ich realizacji wykorzystaliśmy bibliotekę TestNG. Ponieważ każdy test powinien być niezależny ustalony, początkowy stan bazy danych (do której jest podłączony serwer) jest przywracany przed każdą wykonaniem każdej metody testowej.

Początkowy stan bazy

Nasza baza zakłada istnienie jednego projektu o statusie 'PROJECT', w którym znajduje się jedno zamówienie o statusie 'OPEN' oraz jednego klienta.

```
{
   "projectId" : "6880ce29-258d-4a90-9c68-8064b0ae291a",
   "clientId" : "52ce8f63ed5cf6fe8fbc5df9",
   "projectStatus" : "PROJECT",
   "orders" : [
      {
```

```
            "orderId" : "1513fd13-bf75-46f9-b2f1-c1231a3632fa",
            "status" : "OPEN",
            "revision" : 1,
            ...
        }
    ]
}

{
    "clientId" : "52ce8f63ed5cf6fe8fbc5df9",
    "login" : "C383462",
    "password" : "XrLFXSwo",
    "role" : "CLIENT",
    ...
}
```

Scenariusze testowe

Test 1

Cel: Pobranie statusu konkretnego zamówienia wraz z wykonaniem mapowania.

Given: Istnieje projekt z jednym zamówieniem o statusie 'OPEN'.

When: Pobieramy status zamówienia.

Then: Otrzymany status to 'NOT_STARTED'.

Test 2

Cel: Sprawdzenie poprawności danych wejściowych

Given: Posiadamy wadliwą paczkę - niezgodną ze standardem, mającą stworzyć nowy projekt.

When: Wysyłamy paczkę do serwera zlecając wycenę projektu.

Then: Otrzymujemy błąd spowodowany niepoprawnym formatem archiwum.

Test 3

Cel: Założenie nowego projektu oraz stworzenie nowego klienta

Given: Posiadamy poprawnie wygenerowaną paczkę mającą stworzyć nowy projekt.

And: Chcemy założyć nowe konto wprowadzając określone dane.

When: Wysyłamy paczkę do serwera zlecając wycenę projektu.

Then: W bazie istnieje jeden projekt o statusie 'QUOTE' i podanych przez nas identyfikatorach.

And: W bazie istnieje klient o wprowadzonych przez nas danych.

Test 4

Cel: Dodanie zamówienie do istniejącego projektu.

Given: Posiadamy projekt z jednym zamówieniem.

And: Posiadamy poprawnie wygenerowaną paczkę o id istniejącego projektu i nowym id zamówienia

When: Wysyłamy paczkę do serwera dodając zamówienie do projektu.

Then: Posiadamy w bazie jeden projekt o dwóch zamówieniach.

Test 5

Cel: Dodanie kolejny raz tego samego zamówienia do projektu.

Given: Posiadamy projekt z jednym zamówieniem.

And: Posiadamy poprawnie wygenerowaną paczkę o id istniejącego projektu oraz istniejącego dla niego zamówienia

When: Wysyłamy paczkę do serwera dodając zamówienie do projektu.

Then: Otrzymujemy błąd spowodowany istnieniem takiego zamówienia oraz projektu w bazie.

6.3 Działanie aplikacji klienckiej

Testy te miały za zadanie sprawdzić działanie całej aplikacji STAT (a nie tylko poszczególnych części), uruchamianej w taki sposób jak robi to końcowy użytkownik.

6.3.1 Scenariusze testowe

Test 1

Cel: Założenie projektu w danym katalogu

Given: Istnieje katalog zawierający pliki do tłumaczenia

When: Przeprowadzamy konfigurację

Then: Otrzymujemy prywatny katalog aplikacji z zapisanymi (wprowadzonymi przez nas) ustawieniami

Test 2

Cel: Wygenerowanie paczki

Given: Istnieje skonfigurowany projekt

When: Przeprowadzamy generację paczki

Then: Postała poprawna paczka i została przekazana wtyczce do przesłania

Test 3

Cel: Pobranie paczki

Given: Istnieje przetłumaczona paczka do odebrania

When: Pobieramy i instalujemy paczkę

Then: W katalogu projektu znalazły się wynikowe pliki

7 Zgodność rezultatu z początkowymi założeniami

Uzyskany rezultat realizuje przyjęte założenia i został zaakceptowany przez klienta. Projekt był dość znacznie rozpięty w czasie, co spowodawało zmiany w koncepcji systemu jak i w wymaganiach formułowanych przez klienta. Głównym kryterium stawianym przed naszym projektem była maksymalna prostota dostarczanej usługi, tak by nie zrazić potencjalnego użytkownika niezrozumiałymi opcjami konfiguracyjnymi czy też zagmatwanym przepływem wykonywanych akcji. Na samym początku klient oczekiwał dostarczenia prostej wtyczki do Maven'a, co niestety okazało się nie do końca możliwe. Oprócz takiej wtyczki (oferującej jedynie część funkcjonalności) dostarczony przez nas został interfejs konsolowy jak również graficzny interfejs użytkownika dla aplikacji klienckiej. Początkowo wszelkie dane takie jak 'projectID', 'orderID' i 'clientID' miały być generowane po stronie klienta, konieczne jednak okazało się wprowadzenie kont dla użytkowników oraz autoryzowanie ich zapytań. To wymaganie zmusiło nas do rozbudowy interfejsu integracyjnego i nieznaczego skomplikowania protokołu komunikacyjnego. Nie były to jednak zmiany na tyle poważne by znacząco rzutować na najistotniejsze dla nas "kryterium prostoty".

8 Minimalizacja ryzyka

Na początku projektu zakładaliśmy istnienie kilku czynników mogących w mniejszym lub większym stopniu utrudnić nam rozwijanie naszego projektu. W trakcie trwania prac projektowych staraliśmy się każdy z tych czynników możliwie minimalizować podejmując odpowiednie kroki :

- Niejasno określone wymagania lub ich błędne zrozumienie oraz analiza - częste spotkania z klientem, na których omawialiśmy nie tylko zakres przeprowadzanych prac, ale także zastanawialiśmy się jak dane wymaganie / rozwiązanie może wpływać na pozostałe prace oraz jakie trudności może sprawić w przyszłości
- Problemy w integracji ze sobą poszczególnych elementów składowych projektu - aplikacja kliencka oraz serwerowa były rozwijane równolegle, ale naszym priorytetem było jak najwcześniejsze stworzenie warstwy (interfejsu) integracyjnego. Bardzo prosta początkowo integracja, która bardzo wiele faktów zakładała, a pewne parametry jej nie były konfigurowalne była przez nas wdrażana od początkowych iteracji, dzięki czemu nie musieliśmy na końcu zespalać dwóch zupełnie odrębnie działających systemów
- Trudności wynikające z nieznajomości zastosowanych technologii - przed zdecydowaniem się na wybór technologii, w której chcemy realizować dany fragment systemu przeprowadzaliśmy wynikliwą analizę mającą przeprowadzić porównanie różnych możliwości pod względem wad i zalet oraz dopasowania do naszego projektu.
- Możliwość pojawienia się dodatkowych wymagań klienta w trakcie prac - pojawiające się wymagania były przez nas realizowane, nie było jednak ich wiele i nie spowodowały nagłego przyrostu prac.

9 Dalsze możliwe kierunki rozwoju

Obecna wersja aplikacji w związku z trzymaniem informacji o aktualnej wersji tłumaczenia w katalogu projektu, wymaga aby obsługiwana była przez jedną osobę. Aby tego uniknąć należałoby wprowadzić dodatkową warstwę (instalowaną przez dostawcę tłumaczeń) która mogłaby przejąć te obowiązki. Innym sposobem rozwiązania tego problemu byłoby uczynienie STAT-u usługą uruchamianą na serwerze i automatycznie wykonującą akcje pobierania/generowania paczek przy commitowaniu zmian do zarządzanych plików. Innym obszarem w którym możliwy byłby rozwój aplikacji, to integracja z systemami kontroli wersji. STAT mógłby sam po zainstalowaniu tłumaczeń wykonać skonfigurowaną akcję w danym VCS (czyli z reguły zacommitować zmiany).