



**MATEMATICKO-FYZIKÁLNÍ
FAKULTA**
Univerzita Karlova

BAKALÁŘSKÁ PRÁCE

Jindřich Cincibuch

Qubit - systém pro správu a tvorbu překladů

Katedra distribuovaných a spolehlivých systémů

Vedoucí bakalářské práce: Mgr. Pavel Ježek, Ph.D.

Studijní program: Informatika

Studijní obor: Obecná informatika

Praha 2017

Prohlašuji, že jsem tuto bakalářskou práci vypracoval(a) samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V dne

Podpis autora

Název práce: Qubit - systém pro správu a tvorbu překladů

Autor: Jindřich Cincibuch

Katedra: Katedra distribuovaných a spolehlivých systémů

Vedoucí bakalářské práce: Mgr. Pavel Ježek, Ph.D., Katedra distribuovaných a spolehlivých systémů

Abstrakt:

Cílem této práce bylo vytvořit softwarové řešení, které umožní překladatelským agenturám lépe spolupracovat s *freelance* překladateli, kteří nevlastní nákladné překladatelské nástroje.

Součástí softwarového řešení je informační systém s grafickou aplikací, která bude umožňovat zaměstnancům překladatelské agentury vzdáleně zadávat dokumenty k překladu a spravovat je. Další částí je grafická aplikace pro *freelance* překladatele s integrovaným grafickým editorem překladů, v němž budou překladatelé realizovat samotný překlad zadaných souborů. Softwarové řešení je zaměřeno především na podporu editování komerčního překladatelského formátu SDLXLIFF, tak aby ho bylo možné zpětně používat v originální aplikaci SDL Trados Studio.

V textové části práce je podrobně představena problematika překladů a formátů užívaných pro jejich sdílení. Dále je rozebrán návrh, analýza a samotný vývoj informačního systému a grafických uživatelských aplikací. Následně je popsána struktura implementovaného softwarového řešení.

Klíčová slova: sdlxliff tmx překladatelský editor wcf síťová aplikace informační systém

Title: Qubit - A System for Translations Management and Authoring

Author: Jindřich Cincibuch

Department: Department of Distributed and Dependable Systems

Supervisor: Mgr. Pavel Ježek, Ph.D., Department of Distributed and Dependable Systems

Abstract:

The goal of this thesis was to create a software solution that will allow translation agencies to better collaborate with *freelance* translators who do not own expensive translation tools.

One part of the software solution is an information system with a graphical application that will allow translation agency staff to remotely assign documents for translation and manage them. Another part is a graphic application for *freelance* translators with an integrated graphic translations editor, in which translators will translate specified files. The software solution is primarily designed to support the SDLXLIFF commercial translation format so that it can be reused in the original SDL Trados Studio application.

In the text part of the thesis we present in detail the issue of translations and formats used for their sharing. It also presents the design, analysis and development of the mentioned information system and graphical user applications. Subsequently, the structure of the implemented software solution is described.

Keywords: Sdlxliff tmx translation editor wcf network application information system

Chtěl bych poděkovat vedoucímu práce Mgr. Pavlu Ježkovi, Ph.D. za trpělivost a pomoc při vypracování této práce, zadavatelské překladatelské agentuře za poskytnutí zázemí a možnost spolupráce, a rodičům a Kláře za podporu během bakalářského studia.

Obsah

1	Úvod	3
1.1	Svět jazykových překladů	3
1.2	Řešený problém	5
1.3	Možná (ne)řešení	5
1.4	Požadavky na vlastní softwarové řešení	8
1.5	Cíle práce	9
2	Lokalizovatelné formáty	10
2.1	Segmenty a tagy	10
2.2	Formát SDLXLIFF	11
2.2.1	Struktura SDLXLIFF	12
2.2.2	Ukázka SDLXLIFF	13
2.3	Formát TMX	16
2.3.1	Struktura TMX	16
2.3.2	Fomátovací tagy v TMX	17
3	Otestování základní funkčnosti	19
4	Analýza řešení	21
4.1	Volba jazyka a prostředí	21
4.2	Architektura celého řešení	21
4.2.1	Serverová komponenta	22
4.2.2	Ukládání překládaných souborů	23
4.2.3	Klientské aplikace	23
4.3	Propojení aplikačního serveru a klientských aplikací	24
4.3.1	Práce s WCF	25
4.4	Zabezpečení uživatelů a jejich přihlašování	26
4.4.1	Autentizace	26
4.4.2	Autorizace	27
4.5	Společný formát překládaných souborů	28
4.5.1	Mapování SDLXLIFF a TMX	28
4.6	Klientské aplikace	30
4.6.1	Volba technologie	30
4.6.2	Překladačská aplikace	31
4.6.3	Aplikace pro projektového manažera	34
4.7	Datová vrstva	35
4.7.1	Databázové schéma	35
4.7.2	Přístup do databáze z kódu	37
4.8	Instalátor a aktualizace	38
5	Vývojová dokumentace	39
5.1	Struktura Visual Studio solution	39
5.2	Projekt MiddleTier	40
5.3	Shared	41
5.3.1	Adresář ServiceData	41

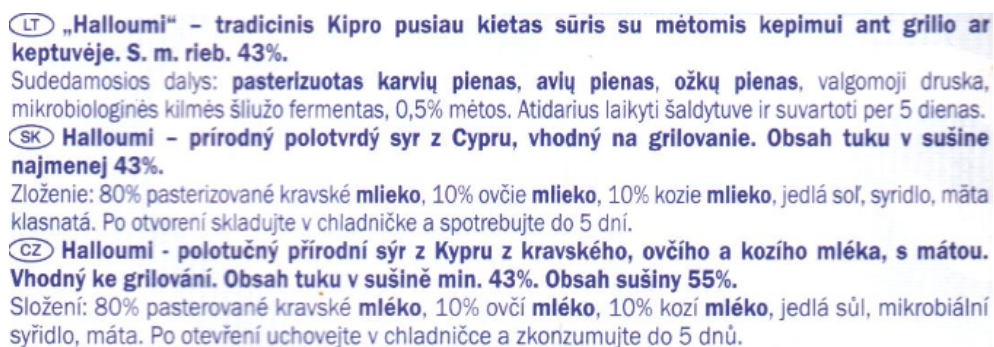
5.3.2	Další třídy	42
5.3.3	Třída <code>ServiceDelegate</code>	42
5.4	Formats	43
5.4.1	Třídy pro reprezentaci překladatelských dokumentů	43
5.4.2	Konvertory	45
5.5	Projekty pro klientské aplikace	46
5.5.1	<code>ProjectManagerClient</code>	46
5.5.2	<code>TranslatorClient</code>	46
5.5.3	Instalátory klientských aplikací	47
5.5.4	Race condition u modálního okna	48
6	Uživatelská dokumentace	49
6.1	Administrátorská dokumentace	49
6.1.1	Testovací nasazení na <i>localhostu</i>	52
6.2	Klientské aplikace	53
6.2.1	Instalace	53
6.2.2	Aplikace pro projektového manažera	53
6.2.3	Aplikace pro překladatele	56
	Závěr	58
	Seznam použité literatury	59
	Přílohy	60
A	Implementace	60
B	Testovací nasazení	60
C	Ukázkové soubory	60
D	Podpůrný software	61

1. Úvod

Tato bakalářská práce se věnuje vývoji aplikace, která bude zefektivňovat práci překladatelů a překladatelských agentur v oboru jazykových překladů. V následujících podkapitolách nejprve čtenáře uvedeme do tohoto oboru, poté podrobně představíme situaci, kterou by měla tato práce vyřešit a nakonec představíme cíle, které by měla naše práce splnit.

1.1 Svět jazykových překladů

Jelikož je překladatelský obor velice specifický, tak v této podkapitole nastíníme, jak překladatelský průmysl funguje. Je důležité zmínit, že hovoříme o překladatelském průmyslu, protože se zaměřujeme především na segment neliterárních překladů, jako jsou například překlady potvrzení, smluv, návodů, webů, etiket, reklamní grafiky apod., ale ne překlady literárních děl. V tomto segmentu překladů je velice důležité zachovat vizuální podobu původního dokumentu, protože se jedná například o již výše zmíněné etikety, kde text musí mít specifickou sazbu a velikost, příklad etikety můžeme vidět na obrázku 1.1.1.



Obrázek 1.1.1: Příklad vícejazyčné etikety.

V ideální situaci by práce překladatele mohla vypadat tak, že klient (například firma, která chce přeložit etikety ke svému výrobku) sám kontaktuje přímo překladatele, poté mu pošle elektronický dokument, který chce přeložit. Překladatel dokument přeloží (s tím, že zachová formátování a grafickou podobu původního dokumentu) a pošle ho klientovi zpět.

Reálně je ale situace složitější, protože musíme vzít v úvahu několik praktických problémů. Ne každý překladatel má vlastní webovou stránku, takže by se o něm klient ani nedozvěděl nebo je překlad rozsáhlý a pro jeho realizaci je potřeba celý tým překladatelů, korektorů a grafiků. Proto se objevuje další článek – překladatelská agentura – firma, která má na starosti komunikaci s klientem, vyhledání a najmutí překladatelů, korekturu, ošetření grafiky, soudní ověření překladu apod. Výsledkem je, že se překladatel může starat pouze o samotný překlad a klient obdrží spolehlivou službu.

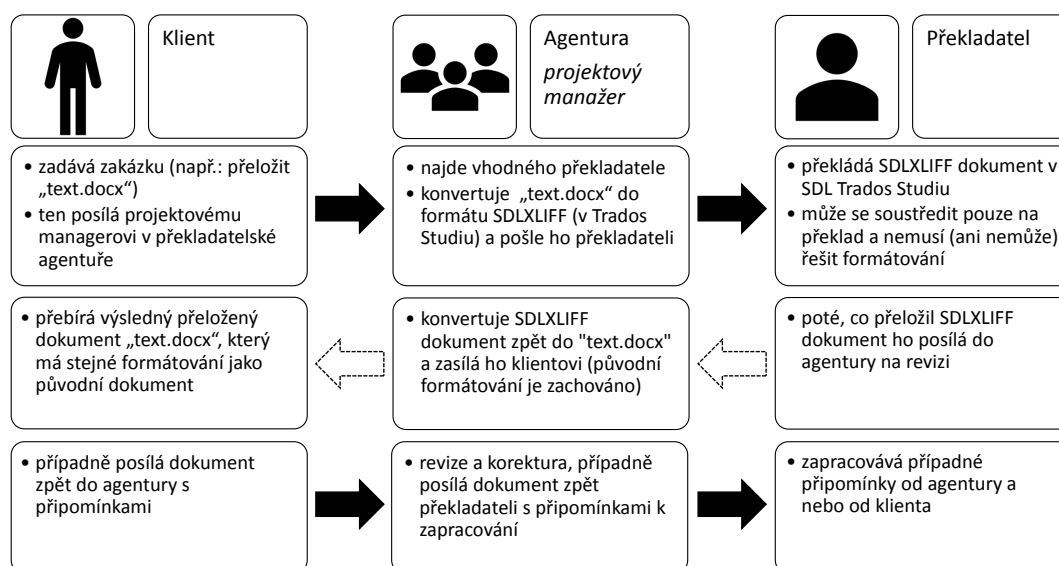
Překladačský workflow

V dnešní době překladačský workflow většinou vypadá tak, že klient kontaktuje překladačskou agenturu a zadá jí zakázku. Teprve překladačská agentura poté zadá práci reálným překladačům (tito překladače mohou být jak interní *in-house* překladače, tak externí *freelance* překladače). Zaměstnanec v překladačské agentuře, který obstarává na jedné straně komunikaci s klientem (zadavatelem zakázky) a na druhé straně komunikaci se samotným překladačem se nazývá projektový manager.

Aby se překladač mohl soustředit pouze na samotný překlad a nemusel řešit problémy technického rázu (například grafická úprava či formátování zdrojového dokumentu), projektový manager překladači zašle dokument ve speciálním formátu určeném pro přenos a sdílení lokalizovatelných dat (do tohoto formátu musí originální dokument zkonvertovat). K tomu projektový manager i překladač potřebují vhodný softwarový nástroj – takzvaný CAT (Computer Assisted Translation) program. Někdy překladačská agentura dostane již přímo od klienta soubory v lokalizovatelném speciálním formátu.

CAT program (Computer Assisted Translation) je zpravidla grafický desktopový software, který zvyšuje produktivitu celého procesu překladu. Nejpoužívanějším CAT programem v současnosti je **SDL Trados Studio** s více než 80% tržním podílem (viz studie překladačské komunity *ProZ* [13]), zároveň se jedná o jeden z nejdražších CAT programů, kdy roční licence pro jednoho překladače stojí 699€ a pro projektového manažera 2500€ (viz internetový obchod SDL [10]).

SDL Trados Studio umí převést překládaný elektronický dokument (podporována je většina běžně používaných elektronických formátů jako jsou například `.docx`, `.xlsx`, `.pdf`, `.ai` apod.) do univerzálního formátu SDLXLIFF – formátu pro sdílení lokalizovatelných dat (podrobněji o SDLXLIFF v kapitole 2). Tento `.sdlxliff` soubor potom překladač přeloží v unifikovaném SDLXLIFF editoru, který je také součástí SDL Trados Studia. Nyní přichází zásadní výhoda SDL Trados Studia (a formátu SDLXLIFF), které dokáže z tohoto `.sdlxliff` souboru zpětně vygenerovat **přeložený** elektronický dokument v původním formátu (`.docx`, `.xlsx`, `.pdf`, `.ai` apod.). Formát SDLXLIFF je také díky tržnímu podílu Trados Studia v překladačském průmyslu de-facto standardem pro sdílení lokalizovatelných souborů. Výše popsany postup překladačského workflow můžeme vidět na obrázku 1.1.2.



Obrázek 1.1.2: Překladatelský workflow.

1.2 Řešený problém

Překladatelské agentury často spolupracují s drobnými *freelance* překladateli, kteří jsou zpravidla levnější nebo se na některé málo frekventované jazykové kombinace nevyplatí agentuře zaměstnávat vlastního *in-house* překladatele.

Problém nastává ve chvíli, kdy v překladatelské agentuře interně používají SDL Trados Studio a zároveň chtějí spolupracovat s *freelance* překladateli, kteří nemají svojí vlastní licenci SDL Trados Studia. Objevují se tyto dvě překážky:

- Soubor ve formátu SDLXLIFF není možné editovat bez editoru s podporou SDLXLIFF (je samozřejmě možné ho editovat v běžném textovém editoru – SDLXLIFF je XML formát – to se ale neslučuje s efektivní prací překladatele).
- SDL Trados Studio kromě editoru také obsahuje projektový management a možnost jednoduchého sdílení lokalizovatelných dokumentů pomocí *cloudu*. Bez těchto funkcí by projektový manager musel soubory posílat mailem, což zvyšuje celou administrativní zátěž, jak pro projektového manažera, tak pro překladatele, a celkově zvyšuje náklady na překlad.

1.3 Možná (ne)řešení

Situace v překladatelské agentuře, pro kterou autor pracuje, vypadá tak, že *in-house* překladatelé mají licence SDL Trados Studia a úspěšně ho používají. Nyní ale tato agentura řeší výše zmíněný problém – jak umožnit snadnou spolupráci agentury s *freelance* překladateli, kteří nemají licenci SDL Trados Studia? Nyní představíme uvažovaná řešení, jejich možnosti a omezení:

Nepoužívat formát SDLXLIFF

To bohužel není možné, protože výše zmíněná překladatelská agentura často dostane soubory k přeložení již ve formátu SDLXLIFF od klienta (toto se stává v případě, že klientem je jiná překladatelská agentura). Dalším důvodem je to, že v agentuře interně používají SDL Trados Studio (jak již bylo zmíněno výše), pro které je SDLXLIFF „nativním“ formátem.

Nespolupracovat s *freelance* překladateli bez Trados licence

Není možné z ekonomických důvodů. *Freelance* překladatelé jsou většinou levnější.

Koupit *freelance* překladatelům Trados licence

Není možné z ekonomických důvodů. Spolupracujících *freelance* překladatelů je mnoho a zpravidla je spolupráce příliš krátká na to, aby se agentuře vyplatilo do nich investovat do takové míry jako do *in-house* překladatelů.

Použít existující komerční řešení

SDL Trados Studio má vestavěnou funkci „*Export to external review*“, která místo .sdlxliff souboru vygeneruje .docx dokument s tabulkou, do které překladatel dopíše svůj překlad. Podrobněji postup vypadá takto:

1. Projektový manager v SDL Trados Studiu vygeneruje z SDLXLIFF dokumentu .docx dokument se speciální tabulkou, kterou můžeme vidět na obrázku 1.3.1.
2. Tento .docx dokument je poté zaslán překladateli bez Trados Studia.
3. Překladatel do této tabulky v Microsoft Word dopíše svůj překlad a celý .docx dokument pošle e-mailem zpět projektovému managerovi.
4. Projektový manager .docx dokument zpětně nahraje do SDL Trados Studia, které do původního SDLXLIFF dokumentu doplní překlad z tabulky (zmíněné v bodu 1) z .docx dokumentu.
5. S tímto SDLXLIFF dokumentem už projektový manager může dále pracovat, například z něj vygenerovat původní překládaný dokument a ten poslat klientovi.

Segment ID	Segment status	Source segment	Target segment
1	Not Translated (0%)	"<0/>" auf Seite <1/>	"<0/>" auf Seite <1/>
2	Translated (100%)	"<2/>"	"<2/>"
3	Translated (100%)	"<3/>" auf Seite <4/>	"<3/>" a pagina <4/>
4	Translated (100%)	"<5/>"	"<5/>"
5	Translated (100%)	<6/> auf Seite <7/>	<6/> a pagina <7/>
6	Translated (100%)	"<9/>" auf Seite <10/>	"<9/>" a pagina <10/>
7	Translated (100%)	"<11/>"	"<11/>"
8	Translated (100%)	Seite <12/>	Pagina <12/>
9	Translated (100%)	<18/>.<19/>.<20/>.<21/>	<18/>.<19/>.<20/>.<21/>
10	Translated (100%)	<24/>.<25/>.<26/>	<24/>.<25/>.<26/>
11	Translated (100%)	<29/>.<30/>.<31/>.<32/>	<29/>.<30/>.<31/>.<32/>
12	Translated (99%)	<44/>Montagerichtlinien <45/>=<46/>	<44/>Istruzioni per il montaggio
13	Translated (99%)	<47/>EgoKiefer <48/>=<49/> Ausgabe 2016 <50/>	<47/>EgoKiefer <48/>=<49/> Edizione 2016 <50/>
14	Not Translated (0%)	EgoKiefer Montagerichtlinien	EgoKiefer Montagerichtlinien

Obrázek 1.3.1: Tabulka v .docx dokumentu vygenerovaném z SDL Trados Studia.

Tento postup prozatím používá již výše zmíněná agentura. Bohužel se ukazuje, že tento postup je v praxi dlouhodobě neudržitelný, a to ze dvou důvodů:

- Formát výsledného .docx dokumentu není robustní, protože v něm překladatel udělá nějakou nepovolenou změnu (stačí drobná změna tabulky, písma atd.), což se stává velmi často, tak se již tento dokument nepodaří zpětně nahrát do Trados Studia.
- Organizace práce se stává velmi náročnou, tyto .docx dokumenty se musí opakovaně posílat e-mailem mezi projektovým manažerem a překladatelem (např.: kvůli revizím či pravidelné kontrole průběhu práce). Ztrácí se přehled na odevzdávacími termíny a narůstá objem práce pro projektového manažera, protože si musí vést evidenci zadaných dokumentů k přeložení.

Vlastní softwarové řešení

Vzhledem k tomu, že ani jedno z předchozích řešení nesplňovalo požadavky výše zmíněné překladatelské agentury, tak bylo rozhodnuto vyvinout vlastní softwarové řešení agentury na míru.

Tímto řešením se má stát právě tato bakalářská práce, která by měla poskytnout této překladatelské agentuře systém pro zadávání překladů *freelance* překladatelům, a zároveň těmto překladatelům jednoduchý editor dokumentů .sdlxliff, který bude napojen na tento systém pro správu projektů.

1.4 Požadavky na vlastní softwarové řešení

Před samotným vývojem vlastního řešení bylo nutné zjistit, jaké požadavky by měl náš software splňovat. Na základě konzultací s zadavatelskou agenturou jsme zjistili, že aby bylo řešení pro agenturu použitelné v praxi, tak by mělo splňovat následující požadavky:

R1 Centrální systém

Řešení by mělo poskytnout centrální systém pro ukládání a správu překládaných dokumentů. To proto, aby agentura měla nad soubory i celým procesem překladu kontrolu. Předpokládá se, že jako datové úložiště bude použita databáze, která bude pod kontrolou IT oddělení agentury. Celý systém musí být veřejně přístupný z internetu, aby se mohli *freelance* překladatelé připojovat z celého světa. Budou zde uložena citlivá data, proto by přístupy měly být patřičně zabezpečené. O každém dokumentu by mělo být vedeno množství informací – datum vytvoření, přidělený překladatel, datum odevzdání, informace o dokumentu (např.: zdrojový a cílový jazyk), jméno projektového manažera, který dokument vložil a další. Co vše se bude evidovat, by mělo být rozhodnuto během vývoje a testování aplikace. Tato centrální část by měla běžet na operačním systému Microsoft Windows Server, protože IT zázemí zadavatelské agentury je celkově postaveno na řešeních od firmy Microsoft.

R2 Vzdálená správa

Řešení by mělo poskytnout grafickou aplikaci pro vzdálenou správu centrálního úložiště pro projektové managery. Toto rozhraní by mělo umožnit vkládat nové dokumenty a přidělovat je jednotlivým překladatelům, vytvářet nové uživatele, měnit informace. V případě příznivých časových možností, by v aplikaci mohla být možnost sdružovat jednotlivé dokumenty logicky do projektů a podpora pro práva a hierarchii uživatelů (např.: překladatel bude moci „vidět“ a editovat pouze dokumenty přiřazené přímo jemu, 3 úrovně práv projektových managerů). Tato aplikace by měla být kompatibilní s operačním systémem Windows 7 a vyšší.

R3 SDLXLIFF editor

Součástí řešení musí být grafický editor dokumentů ve formátu SDLXLIFF, který budou používat překladatelé bez SDL Trados Studia. Editované dokumenty musí být možné zpětně nahrát do SDL Trados Studia. Tato funkcionality umožní agentuře interně a při externí komunikaci stále používat formát SDLXLIFF (a využívat jeho výhod), ale zároveň bude moci využívat služeb menších překladatelů. Je nutné podotknout, že tato funkcionality je klíčová pro použití v praxi. Tento editor by měl být napojen na centrální úložiště (viz bod **R1** výše), aby se překladatel dostal k SDLXLIFF souborům pouze skrz tento editor – to zajistí konzistenci editovaných SDLXLIFF souborů. Tento editor by měl být kompatibilní s operačním systémem Windows 7 a vyšší.

R4 Podpora pro TMX

Dále by mělo být možné nahrávat a editovat také formát TMX (Translation Memory eXchange, viz specifikace TMX [2]), otevřený XML formát určený

pro přenos lokalizovatelných dokumentů. Pro zadavatelskou agenturu není tato funkcionality kritická (dostávají pouze malé množství zakázek ve formátu TMX), přesto by se ale hodilo mít možnost exportovat do tohoto formátu, protože na rozdíl od formátu SDLXLIFF je otevřený a akceptují ho i další komerční CAT programy.

1.5 Cíle práce

Tato bakalářská práce by měla naplnit níže uvedené cíle, které jsou shrnutím požadavků zmíněných v kapitole 1.4.

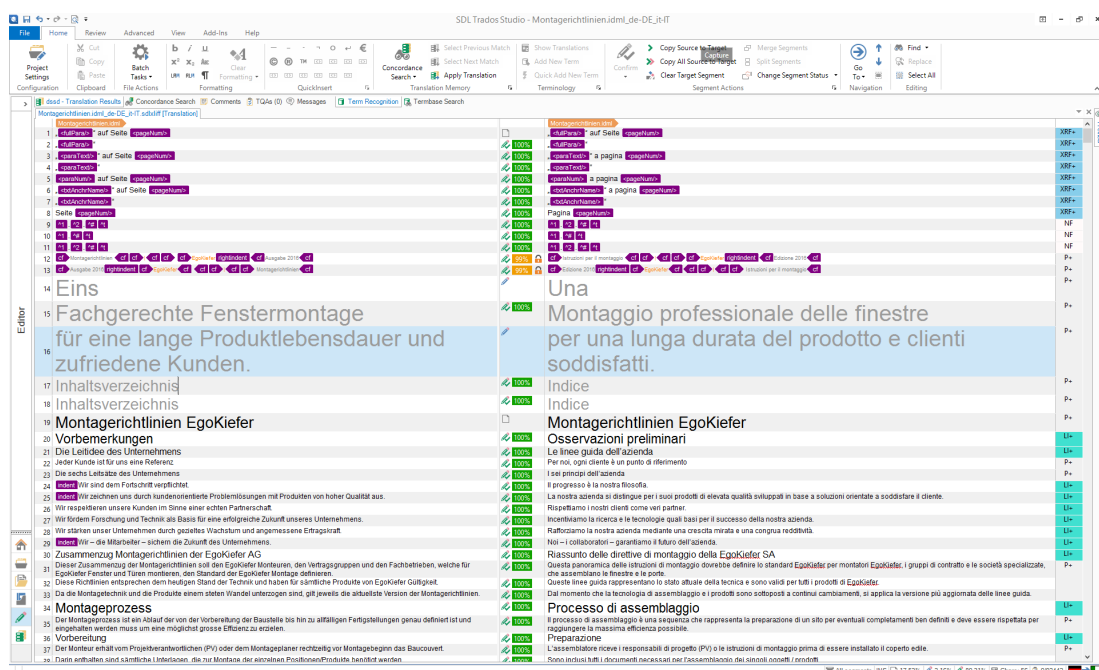
1. Naimplementovat centrální systém, který by měl:
 - (a) poskytnout úložiště a správu překládaných dokumentů,
 - (b) ukládat metainformace o uložených dokumentech,
 - (c) být veřejně dostupný z Internetu,
 - (d) poskytnout správu uživatelů a jejich práv v systému,
 - (e) podporovat autentizaci a autorizaci uživatelů,
 - (f) poskytnout podporu pro sdružování dokumentů do projektů.
 - (g) být kompatibilní s operačním systémem Microsoft Windows Server 2012.
2. Vytvořit klientskou aplikaci pro projektového manažera, která by měla:
 - (a) poskytnout vzdálenou správu centrálního systému,
 - (b) poskytnout možnost projektovému managerovi nahrávat nové dokumenty k přeložení a jejich přiřazení překladatelům,
 - (c) podporovat nahrávání dokumentů ve formátech SDLXLIFF a TMX.
 - (d) podporovat zpětný export nahraných (a v aplikaci editovaných) dokumentů do formátů SDLXLIFF a TMX, tak aby byly zpětně použitelné v programu SDL Trados Studio,
 - (e) být kompatibilní s operačním systémem Microsoft Windows 7/8/10.
3. Vytvořit klientskou aplikaci pro překladatele, která by měla:
 - (a) obsahovat editor překladů přiřazených přihlášenému překladateli,
 - (b) tento editor by měl podobné uživatelské rozhraní jako editor překladů v SDL Trados Studiu,
 - (c) zpětně přeložené dokumenty nahrávat do centrálního systému,
 - (d) být kompatibilní s operačním systémem Microsoft Windows 7/8/10.

2. Lokalizovatelné formáty

Vzhledem k tomu, že se dále v této práci zabýváme elektronickými formáty pro sdílení lokalizovatelných dokumentů, které nemusí být čtenáři pohybujícímu se mimo překladatelský obor známé, tak je v této kapitole představíme. Nejprve ukážeme, jak probíhá převod překládaného dokumentu do lokalizovatelných formátů.

2.1 Segmenty a tagy

Při převodu překládaného dokumentu do univerzálního lokalizovatelného formátu (ať už SDLXLIFF nebo TMX) je zdrojový text rozdělen SDL Trados Studií (obecně jakýmkoliv CAT programem, viz podkapitola 1.1, ale vzhledem k cílům práce, se budeme primárně zabývat SDL Trados Studií) na takzvané *segmenty*. Velikost segmentů je většinou konfigurovatelná, ale zpravidla platí to, že jedna věta ze zdrojového textu se promítne do jednoho segmentu. Toto rozložení na segmenty je uloženo v daném lokalizovatelném formátu a umožňuje dosáhnout překladateli větší efektivity.



Obrázek 2.1.1: Editor překladů v SDL Trados Studiu, v levém sloupci je zdrojový text, v pravém přeložený text. Vidíme, že se jedná o již přeložený dokument, protože pravý sloupec je celý vyplněn.

Překladatel poté v překladatelském editoru, který je součástí SDL Trados Studia, již překládá jednotlivé segmenty. Editor zobrazuje tabulku se dvěma sloupci, kdy každý řádek reprezentuje dvojici segmentů – v levém sloupci je překládaný segment (*zdroj*) a v pravém sloupci je přeložený segment (*cíl*), respektive do něj

překladač dopisuje svůj překlad. Jak tento editor vypadá v SDL Trados Studiu můžeme vidět na obrázku 2.1.1.

Na obrázku 2.1.1 si také můžeme všimnout, že se ve sloupcích neobjevuje jenom text, ale jsou zde i graficky vyznačené takzvané *tagy*, značky, které překladači říkají například, že slovo je ve zdrojovém textu vysázeno jiným fontem (v SDL Trados Studiu se po najetí myši na *tag* zobrazí *tooltip* s podrobnostmi o tom, co daný *tag* symbolizuje). Na obrázku 2.1.2 můžeme vidět podrobnější pohled na jeden z překládaných segmentů.



Obrázek 2.1.2: Podrobnější pohled jeden z **překládaných segmentů**. Vlevo je zdrojový segment, vpravo cílový (překládaný) segment.

Typicky se v lokalizovatelných souborech vyskytují tyto dva základní typy *tagů*:

Párové: Typicky se jedná o *tagy* označující použití nějakého stylu na text, který je těmito *tagy* obalen. Na obrázku 2.1.2 jsou to *tagy* **cf** (*otevírací část*) **cf** (*uzavírací část*). Jedná se *control formatting* *tagy* specifikující formátování obalovaného textu.

Nepárové: V angličtině se tyto *tagy* někdy nazývají *placeholder*, neboli zástupný prvek, který nám říká, že na tomto místě má být například obrázek, odrazka apod. Na obrázku 2.1.2 je to *tag* **anchorref** (ukotvení nějakého jiného grafického prvku v původním souboru).

Dále si na obrázku 2.1.2 můžeme všimnout, že překladač *tagem* **cf** (*tag* symbolizující formátování textu) obalil v cílovém (přeloženém) textu slova „There are“, tedy jiný počet slov a na jiném místě než v překládaném segmentu, protože *tagy* obaluje slova podle významu a kontextu. Zároveň překladač musí vědět o jaký typ *tagu* se jedná, protože zase pro *tag* **anchorref** je naopak žádoucí, aby byl na konci věty, protože tento *tag* pro změnu představuje ukotvení textu v originálním dokumentu. Nyní se podrobně podíváme, jak jsou uspořádány konkrétní formáty SDLXLIFF a TMX.

2.2 Formát SDLXLIFF

Formát SDLXLIFF je XML formát, vycházející z otevřeného formátu XLIFF 1.2, který ve svojí specifikaci (viz [8]) povoluje formát v některých sekcích rozšiřovat o svoje vlastní elementy a atributy. Bohužel firma SDL pro formát SDLXLIFF nikde neuveřejňuje specifikaci jejích vlastních rozšíření, takže jsme při zkoumání formátu SDLXLIFF byli odkázáni pouze na vlastní analýzu sady ukázkových souborů, kterou nám zaslala k tomuto účelu zadavatelská agentura. Pro podrobnosti týkající se formátu XLIFF odkážeme čtenáře na podrobnou specifikaci (viz [8]). V podkapitole 2.2.2 uvádíme ukázkou SDLXLIFF formátu, ve které bylo pro přehlednost vynecháno velké množství XML elementů, podelementů a atributů, které nejsou pro ukázání základní struktury zásadní. Zároveň také na obrázku

2.2.1 můžeme vidět, jak by se ukázka (pokud bychom doplnili všechny náležitosti SDLXLIFF) zobrazila v editoru SDL Trados Studio.

2.2.1 Struktura SDLXLIFF

V kořenovém elementu **xliff** se nachází jeden a více elementů **file** (ve všech testovaných **.sdlxliiff** souborech byl element **file** pouze jednou, proto předpokládáme, že SDL Trados Studio pro každý originální dokument vytváří právě jeden **.sdlxliiff** soubor). Element **file** obsahuje XML elementy **header** a **body**.

Element **header** obsahuje metainformace o souboru (jako je například jméno původního souboru, datum vytvoření apod.). Nás bude zajímat hlavně element **tag-defs**, což je element z rozšíření SDL, který obsahuje seznam XML elementů **tag**, které popisují jednotlivé formátovací tagy (viz podkapitola 2.1), které pak budou později použity v jednotlivých překládaných segmentech. Zde bychom chtěli čtenáře upozornit na možnou záměnu termínů – je třeba rozlišovat překladatelský formátovací *tag* a XML element **tag**.

Element **body** obsahuje seznam XML elementů **group**, kde každý z elementů **group** může obsahovat jeden a více XML elementů **trans-unit**, který obsahuje následující elementy:

seg-source: Tento element obsahuje jeden a více elementů **mrk**. Každý z nich reprezentuje překládaný segment (*zdroj*). Element **mrk** rozebíráme podrobněji níže.

target: Tento element také obsahuje elementy **mrk** stejně jako výše zmíněný **seg-source**, které ale reprezentují již přeložený segment (*cíl*).

sdl:seg-defs: Tento element se nemusí v SDLXLIFF vyskytovat. Pokud zde je, tak by měl obsahovat pro každý **mrk** element z **seg-source** (zmíněného výše) obsahovat jeden element **sdl:seg**, který v podelementech a atributech uchovává metainformace daném segmentu – například o tom jestli je zamčený, jestli byl označen jako již přeložený apod. Příslušející segment najdeme podle atributu **id**, který odkazuje na atribut **mid** v elementu **mrk** (viz níže).

mrk: Pokud element **mrk** obsahuje atribut **mtype** s hodnotou „seg“, pak takový element **mrk** reprezentuje jednu z částí překládaného segmentu. O kterou část se jedná poznáme podle toho jestli je obsažen v elementu **seg-source** (*zdroj*) nebo v elementu **target** (*cíl*). Tyto elementy **mrk** obsahují atribut **mid**, jehož hodnota je identifikátor, kterým mezi sebou můžeme spojit elementy **mrk** z elementů **seg-source** a **target** a případně ještě metadata z **sdl:seg-defs**. Obsahem elementu **mrk** jsou již samotná data – tedy text a případné formátovací tagy (viz 2.1). V SDLXLIFF jsou formátovací tagy reprezentovány následujícími XML elementy:


g: Tento XML element představuje párový tag. Obsah tohoto elementu (text a možné další formátovací tagy) je pak obalen formátovacím párovým tagem. O jaký tag se jedná zjistíme pomocí atributu **id** (viz

níže). Otevírací část tohoto elementu by editoru SDL Trados Studio byla reprezentována například jako  a zavírací část jako .

- x:** Tento XML element reprezentuje nepárový formátovací tag, například odrážku, či obrázek. O jaký tag se jedná zjistíme pomocí atributu `id` (viz níže). V editoru SDL Trados Studio by byl reprezentován například jako `anchoref`.

Tyto XML elementy (`g`, `x`) – reprezentující formátovací tagy – obsahují atribut `id`, což je identifikátor, kterým k sobě dokážeme přiřadit jednotlivé tagy mezi zdrojem a cílem a zároveň je klíčem do seznamu definic formátovacích tagů (zmíněných výše, v elementu `header`).

Je třeba poznamenat, že k rozšíření SDL neexistuje specifikace a i v případech, kdy SDLXLIFF využívá standardních elementů ze specifikace XLIFF, tak je nepoužívá přesně v souladu se specifikací:

- Například si můžeme všimnout, že hned první segment v ukázce 2.2.2 (element `mrk` s `id` „186“) je obalen elementem `g`, což je v rozporu s popisem segmentace ve specifikaci formátu XLIFF. Podle specifikace (viz [8]), by se měly tagy typu `g` pokud obalují text, který přesahuje více elementů `mrk`, vždy na konci segmentu uzavřít a v další segmentu otevřít. Vypozorovali jsme, že takovýto segment se v editoru SDL Trados Studio projeví jiným formátováním. Ne zobrazí se formátovací tagy typu , ale text je například kurzívou – jako můžeme vidět na obrázku 2.2.1, kde jsou slova „Wichtig:“ a „Importante:“ opravdu kurzívou.
- Dále SDLXLIFF také často obsahuje zdrojové segmenty, které neobsahují žádný text, pouze formátovací tagy, o těchto jsme nezjistili k čemu slouží. Editor SDL Trados Studio takové segmenty vůbec nezobrazuje.
- Další neznámou jsou elementy `mrk`, které nemají atribut `mtype`, případně u nich má jinou hodnotu než „seg“. Tyto elementy mohou obalovat další elementy `mrk`. V SDL Trados Studio editoru jsme neobjevili žádný rozdíl oproti segmentům, které nejsou takovými `mrk` obaleny.

2.2.2 Ukázka SDLXLIFF

V ukázce si můžeme všimnout, že zde máme pouze jeden element `group`, který obsahuje pouze jeden element `trans-unit`, v němž máme tři segmenty. Segment s `mid` „187“ obsahuje nepárový tag a text „ein“. Podle hodnoty `id` tagu („221“) najdeme příslušnou definici v XML elementu `tag-defs` a zjistíme, že se jedná o nepárový tag, který představuje obrázek (*image*). Nyní již uvádíme samotnou ukázkou a na obrázku 2.2.1 je jak by byla reprezentována v editoru SDL Trados Studio, pokud bychom jí doplnili na validní SDLXLIFF soubor (ukázka není kompletním SDLXLIFF souborem z demonstračních důvodů, jak jsme již zmínili na začátku 2.2).



Obrázek 2.2.1: Reprezentace ukázky 2.2.1.

Ukázka 2.2.1: **Formát SDLXLIFF** – v ukázce jsou autorem doplněné komentáře pro lepší orientaci (v reálných SDLXLIFF souborech se nevyskytují).

```
<?xml version="1.0" encoding="utf-8"?>
<xliff xmlns:sdl="http://sdl.com/FileTypes/SdlXliff/1.0"
  version="1.2"
  sdl:version="1.0"
  xmlns="urn:oasis:names:tc:xliff:document:1.2">
  <file original="C:\path\file.idml"
    source-language="de-DE"
    target-language="it-IT">
    <header>
      <reference>
        <internal-file form="base64">
          string containing the original file encoded in base64
        </internal-file>
      </reference>
      <!-- Definice použitých formátovacích tagů -->
      <tag-defs xmlns="http://sdl.com/FileTypes/SdlXliff/1.0">
        <tag id="123">
          <bpt name="Italics"
            word-end="false">&lt;cf style=&quot;Italics
              ↪ &quot;&gt;&lt;/bpt>
          <bpt-props>
            <value key="node">&lt;CharacterStyleRange
              ↪ AppliedCharacterStyle=&quot;CharacterStyle/Italics
              ↪ &quot;&gt;&lt;/value>
          </bpt-props>
          <ept name="Italics"
            word-end="false">&lt;/cf&gt;&lt;/ept>
          <fmt id="2"/>
        </tag>
        <tag id="555">
          <bpt name="cf" word-end="false">&lt;cf
            ↪ nfa=&quot;true&quot;&gt;&lt;/bpt>
          <bpt-props>
            <value key="node">&lt;CharacterStyleRange
              ↪ FontStyle=&quot;46 Light Italic&quot;&gt;&lt;/value>
          </bpt-props>
          <ept name="cf" word-end="false">&lt;/cf&gt;&lt;/ept>
        </tag>
        <tag id="221">
          <ph name="image" word-end="false"
            ↪ seg-hint="IncludeWithText">&lt;image
            ↪ id=&quot;u6217&quot;/&gt;&lt;/ph>
```

```

        <props>
          <value key="uniqueid">u1660_88</value>
        </props>
      </tag>
      <!-- Další elementy tag popisující jednotlivé tagy -->
    </tag-defs>
  </header>
  <body>
    <group>
      <trans-unit id="7575ffac-018a-4351-a7c5-cbbbc8f907bb">
        <seg-source>
          <g id="132">
            <!-- Zdrojový segment. Příslušný cílový segment
            ↪ nalezneme podle hodnoty atributu mid. -->
            <mrk mtype="seg" mid="186">Wichtig:</mrk>
          </g>
          <mrk mtype="seg" mid="187">
            <!-- Nepárový tag, podle id můžeme najít jeho přesnější
            ↪ definici a také ho spojit -->
            <x id="221"/> ein</mrk>
            <mrk mtype="seg" mid="188">zwei <g id="555">drei</g>
          </mrk>
        </seg-source>
        <target>
          <g id="123">
            <mrk mtype="seg" mid="186">Importante:</mrk>
          </g>
          <mrk mtype="seg" mid="187">
            <x id="221"/> una</mrk>
            <mrk mtype="seg" mid="188">due <g id="555">tre</g>
          </mrk>
        </target>
        <sdl:seg-defs>
          <!-- Metainformace o segmentu. Příslušný segment nalezneme
          ↪ podle hodnoty atributu id (u segmentu mid).-->
          <sdl:seg id="186" locked="true" conf="Translated" />
          <sdl:seg id="187" locked="true" conf="Translated" />
          <sdl:seg id="188" locked="true" conf="Translated" />
        </sdl:seg-defs>
      </trans-unit>
      <!-- Element může obsahovat více elementů trans-unit -->
    </group>
    <!-- Další elementy group... -->
  </body>
</file>
</xliff>

```

2.3 Formát TMX

Formát TMX je XML formát pro sdílení lokalizovatelných dokumentů. Na rozdíl od formátu SDLXLIFF (respektive XLIFF) podporuje více jazykových variant, ne jen pouze dvě, jako SDLXLIFF/XLIFF. My se budeme zabývat verzí TMX 1.4, která je v současnosti nejpoužívanější verzí TMX.

2.3.1 Struktura TMX

Nyní shrneme základní strukturu `.tmx` souborů, která by měla čtenáři stačit pro rozsah této práce (pro úplnou specifikaci formátu TMX viz [2]). Na konci této kapitoly uvádíme ukázkou `.tmx` souboru (viz 2.3.1). Kořenový element `tmx` obsahuje dva elementy, `header` a `body` (odsazení v následujícím výpisu odpovídá zanoření jednotlivých elementů):

header: V attributech tento element obsahuje různé metainformace. V ukázkovém souboru můžeme vidět například, že zdrojovým jazykem je francouzština (hondota atributu `srclang` je `fr`) nebo typ rozdělení na segmenty

prop: Obsah tohoto elementu je na libovůli tvůrce `.tmx` dokumentu. V atributu `type` může tvůrce popsat o jaký typ informace zde ukládá. Například v ukázkovém souboru 2.3.1 zde vidíme pouze text, ale tvůrce dokumentu, by sem mohl vložit libovolný xml podstrom.

body: Obsahuje samotná data, tedy seznam segmentů (viz podkapitola 2.1). Každý segment je reprezentován elementem `tu`. V tomto seznamu jsou elementy uvedeny ve stejném pořadí jako v původním dokumentu. V ukázkovém souboru je pouze jeden element `tu`.

tu: Tento element, představuje jeden segment (v angličtině někdy také *translation unit*). Obsahuje dva a více `tuv` elementů, takzvaných jazykových variant (*translation unit variant*). Také může obsahovat libovolně elementů `prop`.


prop: Obsah tohoto elementu je na libovůli tvůrce `.tmx` dokumentu. V atributu `type` může tvůrce popsat jaká informace se zde ukládá. Například v ukázkovém souboru 2.3.1 je zde mapování mezi identifikátory formátovacích tagů (kterým se věnujeme v podkapitole 2.3.2 dále) a jejich typy.


tuv: Představuje jednu z jazykových variant příslušného segmentu (nadřazeného elementu `tu`). Jazyk varianty je určen hodnotou atributu `xml:lang`. Obsahuje právě jeden element `seg`.

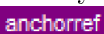
seg: V tomto elementu už je samotný text, jazykové varianty a případné formátovací tagy, kterým se věnujeme níže v podkapitole 2.3.2.

2.3.2 Formátovací tagy v TMX

V TMX jsou formátovací tagy (představené v kapitole 2.1) reprezentovány následujícími XML elementy:

bpt (***Begin paired tag***): Tento XML element představuje otevírací část párového tagu. Jedná se o takzvaný *self-closing* element, kdy nemá žádné podelementy ani neobsahuje žádný text. Formátovací tag reprezentovaný tímto elementem by se v editoru SDL Trados Studio zobrazil takto .

ept (***End pair tag***): Tento XML element představuje uzavírací část párového tagu. Také se jedná o *self-closing* element, jako u **bpt**. Formátovací tag reprezentovaný tímto elementem by se v editoru SDL Trados Studio zobrazil takto .

ph (***Placeholder***): Tento XML element reprezentuje nepárový formátovací tag, například odrážku, či obrázek. Také se jedná o *self-closing* element. Formátovací tag reprezentovaný tímto elementem by se v editoru SDL Trados Studio zobrazil takto .

Tyto výše zmíněné XML elementy (reprezentující formátovací tagy) mohou být rozšířeny XML atributy uvedenými níže:

x (***External matching***): Tento atribut se vykytuje pouze u elementů **bpt** a **ph**. Jeho hodnotou je identifikátor podle kterého k sobě můžeme přiřadit příslušné formátovací tagy (neprezentované výše zmíněnými elementy) napříč jednotlivými jazykovými variantami (reprezentované elementy **tuv**), ale stále v rámci jednoho překládaného segmentu (elementu **tu**). Přestože se podle specifikace TMX jedná o povinný atribut, tak se v některých vzorových souborech tento atribut neobjevuje. Analýzou všech takovýchto souborů jsme zjistili, že pokud se tento atribut neobjeví, tak v ostatních jazykových variantách se příslušný *tag* neobjevuje.

i (***Internal matching***): Tento atribut se vyskytuje pouze u párových *tagů* **bpt** a **ept**. Hodnota atributu opět slouží jako identifikátor, kterým spojujeme **bpt** a **ept** tagy v rámci jedné jazykové varianty (elementu **tuv**). Hodnota tohoto tagu je tedy unikátní pouze v rámci této jedné jazykové varianty. Text (a případné další formátovací tagy), který se nachází mezi spojenými **bpt** a **ept** XML elementů (pomocí atributu **i**), je pak jakoby obalen párovým formátovacím tagem. O tom o jaký typ formátovacího tagu se jedná určuje atribut **type** (viz níže), který je v elementu **bpt**.

type (***Type***): Tento atribut slouží k určení typu formátovacího tagu. Hodnotou může být přímo samotný typ. Ve standartu TMX je například *bold*, *italic*, *arial*, *indent* apod., ale principiálně hodnotou tohoto tagu může být cokoliv – například identifikátor, který odkazuje do mapování definic tagů, jak můžeme vidět v ukázce 2.3.1. Umístění tohoto mapování záleží na tvůrci **.tmx** dokumentu, v ukázce je umístěno v elementu **prop** v elementu **tu**.

Ukázka 2.3.1: **Formát TMX** – v ukázce jsou autorem doplněné komentáře pro lepší orientaci (v reálných generovaných souborech se nevyskytují).

```
<?xml version="1.0" encoding="UTF-8"?>
<tmx version="1.4">
  <header segtype="sentence" srclang="fr">
    <prop type="PropertyKey">Jakýkoliv obsah, ten, kdo vytváří soubor,
      ↪ zde může mít například další nestandardizované informace,
      ↪ například podrobný popis tagů apod.</prop>
  </header>
  <body>
    <tu>
      <prop type="tags">
        <tag type="tagtype:124" value="bold"/>
        <tag type="tagtype:22" value="image"/>
      </prop>
      <!-- Francouzská jazyková varianta-->
      <tuv xml:lang="fr">
        <seg>
          <!-- Otevírací tag (podle atributu type zjistíme, že se
            ↪ jedná o tučnou sazbu (bold))-->
          <bpt i="2" type="tagtype:124" x="3"/>
          <!-- Nepárový tag (podle atributu type zjistíme, že
            ↪ představuje obrázek (image))-->
          <ph type="tagtype:22" x="2"/>
          Oui
          <!-- Uzavírací tag, příslušející otevírací najdeme podle
            ↪ identifikátoru v atributu i-->
          <ept i="2" />
        </seg>
      </tuv>
      <!-- Německá jazyková varianta, ještě nebyla přeložena - element
        ↪ seg je prázdný-->
      <tuv xml:lang="de">
        <seg></seg>
      </tuv>
      <!-- Anglická jazyková varianta-->
      <tuv xml:lang="en">
        <seg>
          <bpt i="2" type="tagtype:124" x="3"/>
          <ph type="tagtype:22" x="2"/>
          Yes
          <ept i="2" />
        </seg>
      </tuv>
    </tu>
  </body>
</tmx>
```

3. Otestování základní funkčnosti

Ještě předtím, než jsme začali se samotným vývojem, bylo třeba zjistit, jestli je vůbec v našich silách naimplementovat editor formátů SDLXLIFF, protože je to pro zadavatelskou agenturu klíčová funkcionalita (dle cílů 2c a 2d), bez které by ani nemělo smysl celou aplikaci vyvíjet.

Od zadavatelské agentury jsme tedy dostali sadu ukázkových souborů, které bychom měli zvládnout nahrát do našeho editoru, upravit a zpětně zase nahrát do SDL Trados Studia, tak aby nich bylo možné opětovně vygenerovat původní elektronické dokumenty a zároveň projevíli námi provedené změny.

Toto vyžadovalo manuální analýzu SDLXLIFF souborů, protože pro formát SDLXLIFF bohužel neexistuje veřejně dostupná specifikace. Níže v krátkosti popíšeme, jak jsme během této analýzy postupovali. Samotné výsledky této analýzy jsou shrnuty v kapitole 2. V dalším textu předpokládáme, že je čtenář s touto kapitolou již seznámen.

Protože naším cílem (viz cíl 3a) je editor SDLXLIFF souborů, museli jsme zjistit jak vlastně původní zabudovaný editor v SDL Trados Studiu zobrazuje tyto soubory a jak reaguje na změny v těchto souborech. To vyžadovalo ruční analýzu sady ukázkových SDLXLIFF souborů, kdy jsme měli vedle sebe otevřený `.sdlxliff` soubor v SDL Trados Studiu a zároveň v Notepad++ (*plain-text* editor). Zjišťovali jsme, jak SDL Trados Studio graficky reprezentuje jednotlivé segmenty a formátovací tagy (viz podkapitola 2.2) a jak reaguje na změny v `.sdlxliff` souboru.

Zjišťovat všechna specifika formátu SDLXLIFF by bylo velice náročné vzhledem k tomu, jaké všechny informace musí obsahovat (například musí obsahovat celý originální původní dokument – kódovaný pomocí *base64* – kvůli funkci zpětného generování). Naší výhodou bylo, že jsme nepočítali s tím, že budeme `.sdlxliff` soubory sami generovat, pouze budeme editovat již existující. Stačilo nám tedy přesně imitovat chování editoru v SDL Trados Studiu a proto jsme mohli studovat pouze část formátu SDLXLIFF. Přesto tato analýza byla pracná, protože jsme museli ručně porovnávat, jak se změny provedené v editoru SDL Trados Studia projeví v samotném `.sdlxliff` souboru.

Nevýhodou tohoto manuálního přístupu je, že nemůžeme zaručit, že náš program dokáže zpracovat všechny možné `.sdlxliff` soubory, protože je možné, že v naší testovací sadě se nějaká vlastnost formátu SDLXLIFF neprojeví. Další nevýhodou je, že pokud se v budoucnu (v dalších verzích SDL Trados Studia) firma SDL rozhodne změnit formát SDLXLIFF nebo ho začne interpretovat jiným způsobem, tak náš editor bude potřebovat také změnit, aby zohlednil tyto úpravy.

S těmito nevýhodami se budeme muset smířit minimálně do doby, než firma SDL vydá specifikaci formátu SDLXLIFF, jinak jsme odkázáni na tento postup zpětného inženýrství.

Výše zmíněnou ruční analýzu jsme si ulehčili, tím, že vyvinuli prototypový

program, do kterého jsme postupně načetli jednotlivé testovací soubory. V tomto programu jsme automaticky – ale v souladu s zjištěními z předchozí ruční analýzy – měnili obsah SDLXLIFF souborů a tak jsme částečně imitovali práci překladatele. Tento postup nám umožnil urychlit testování naší schopnosti správně editovat formát SDLXLIFF, aniž bychom museli tyto změny provádět ručně přímo v textovém editoru. Upravené soubory jsme poté zpětně ručně nahrávali do SDL Trados Studio.

V rámci toho testování základní funkčnosti jsme se ujistili, že dokážeme napodobit chování editoru SDL Trados Studio a tedy jsme mohli začít s vývojem samotné aplikace, jehož průběhu se budeme věnovat v následující kapitole.

4. Analýza řešení

V této kapitole se budeme zabývat podrobnou analýzou vývoje celého softwarového řešení, které budeme dále označovat jako systém. Každá z podkapitol se zaměřuje jeden z okruhů, se kterým jsme se během vývoje zabývali, na jaké problémy jsme narazili a jak jsme je vyřešili.

4.1 Volba jazyka a prostředí

Protože jsme podle povahy projektu očekávali, že budeme pravděpodobně pracovat s databází, XML soubory a vytvářet grafické uživatelské rozhraní (viz cíle práce [1a](#), [2c](#) a [3a](#)), tak jsme potřebovali zvolit vhodný programovací jazyk a prostředí, které by byli pro tyto účely vhodné.

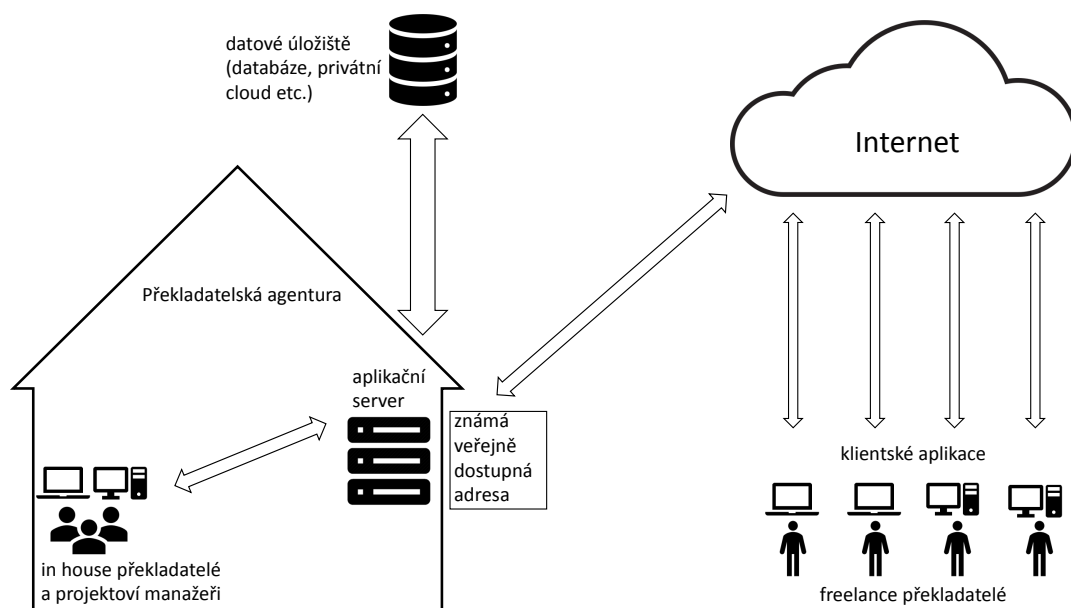
Nakonec jsme jako programovací jazyk zvolili jazyk **C#** [\[6\]](#) na základě autorovy zkušenosti s tímto jazykem, množstvím podpůrných knihoven, a protože všechny části systému (dle cílů práce [1g](#), [2e](#) a [3d](#)) by měly běžet na systémech Windows, se kterými je celá platforma **.NET** [\[7\]](#) (jejíž je jazyk **C#** součástí) kompatibilní.

Vývojovým nástrojem bylo zvoleno Microsoft Visual Studio, kvůli své integraci s jazykem **C#** a **.NET** frameworkem a protože se jedná o nejpoužívanější a jedno z nejvyspělejších IDE (Integrated Development Enviroment - integrované vývojové prostředí) pro vývoj v jazyce **C#**. Pro správu kódu budeme využívat nástroj **git**.

4.2 Architektura celého řešení

Na začátku vývoje bylo nutné rozhodnout, jaká bude architektura celého systému. Vzhledem k cílům práce (viz cíle práce [2](#) a [3](#)) je zřejmé, že bude potřeba vyvinout jednu grafickou aplikaci pro freelance překladatele a jednu pro projektové manažery. Tyto aplikace budeme dále nazývat jako klientské aplikace.

Dále potřebujeme mít serverovou část, která se bude starat o správu uložených dat a poskytovat veřejný přístup k nim pro klientské aplikace. Na základě výše uvedeného, jsme se rozhodli zvolit server–klient architekturu, kdy v roli serveru zde bude vystupovat centrální úložiště (které budeme dále označovat také jako serverovou komponentu) a roli klientů budou hrát klientské aplikace. Nyní se podrobněji popíšeme rozdělení na jednotlivé části celého systému, toto je ilustrováno na obrázku [4.2.1](#), na který se budeme odkazovat ve zbytku této kapitoly.



Obrázek 4.2.1: Diagram architektury systému

4.2.1 Serverová komponenta

Serverovou komponentu jsme se rozhodli rozdělit na tyto dvě části požadované funkce:

Datové úložiště Tato část bude realizována pomocí databáze. Půjde o takzvaný „hloupý“ *datastore*, který by měl obsahovat, co nejméně business logiky. Budou zde uloženy informace o uživatelích, jejich autentizační údaje, informace o zadaných překladech apod. Tato část je na obrázku 4.2.1 označena jako databáze, a také ji budeme tak v následujících kapitolách označovat.

Aplikační server Tato část bude realizována jako aplikace, která poběží na veřejně dostupném serveru na platformě Windows Server (viz cíl 1c). Tato aplikace bude zajišťovat všechny přístupy z klientských aplikací k datům z výše zmíněného datového úložiště. Bude zajišťovat autentizaci a autorizaci uživatelů a jejich akcí, konzistenci vkládaných dat apod. Tato část je na obrázku 4.2.1 označena jako aplikační server, a také ji tak budeme v následujících kapitolách označovat.

Pro rozdělení serverové komponenty na výše zmíněné části jsme se rozhodli z těchto důvodů:

- Díky tomu, že samotná data budou v databázi, tak zadavatelská agentura získá přímý přístup k těmto datům – například přes databázového správce (například Microsoft SQL Server Management Studio) – a tedy nebude závislá pouze na přístupu skrze klientské aplikace, což je zejména užitečné případě pádu či nedostupnosti aplikačního serveru.
- Databáze nám poskytne spolehlivou platformu pro uchovávání dat, na kterou se pak budeme moci spolehnout při implementaci aplikačního serveru.

Zpřehlední se celý systém – každá komponenta má svůj daný účel – a tím se ulehčí jeho vývoj a zlepší udržitelnost.

- Přestože databáze považujeme za spolehlivý prostředek uchování dat, tak naopak pro implementaci „business logiky“ považujeme za vhodnější použít výše zmíněný .NET Framework (viz 4.1), který poskytuje větší programátorský komfort (množství podpůrných knihoven, možnost použití *source control* apod.).

4.2.2 Ukládání překládaných souborů

Dále jsme se museli rozhodnout, jak budeme nakládat se samotnými překládanými soubory, jejichž správu máme dle cíle 1a poskytovat. Jedním z požadavků zadavatelské agentury je co největší kontrola nad celým procesem překladu. To zahrnuje i kontrolu nad všemi daty (včetně samotných překládaných souborů). Proto jsme se rozhodli pro takzvanou *cloudovou* variantu, kdy se i překládané soubory budou nacházet v centrálním datovém úložišti (viz 4.2.1).

Nevýhodou zvoleného přístupu je nutnost internetového připojení pro práci. Klientská aplikace bude muset udržovat spojení se serverovou komponentou, kam bude zapisovat všechny změny provedené uživatelem. Toto bude limitující zejména pro překladatele, jejichž práce je typicky dlouhodobějšího charakteru. Další nevýhodou je zvýšené využití pásma internetového připojení – klientská aplikace si bude překládaný soubor stahovat při každém otevření.

Naopak výhodou je snadnější správa souborů jak pro autora práce, tak pro samotnou agenturu, protože existuje pouze jedno místo, kde se soubory nacházejí a tedy není třeba řešit řadu problémů spojených se synchronizací apod.

4.2.3 Klientské aplikace

Klientské aplikace budou dvě – jedna pro překladatele (tu budeme dále označovat jako překladatelskou aplikaci) a jedna pro projektového manažera (tu budeme označovat jako manažerskou aplikaci). Rozhodovali jsme se mezi dvěma možnostmi implementace:

Webová aplikace: Výhodou je snadná přenositelnost. Nevýhodou je náročnost implementace kvalitního překladatelského editoru jako webové aplikace (s přihlédnutím k autorovým zkušenostem s webovým vývojem).

Desktopvá aplikace: Nevýhodou bude vazba na existenci .NET Frameworku na cílové platformě (vzhledem k naší předchozí volbě této technologie, viz 4.1), tato nevýhoda ale není zásadní, protože všechny naše cílové platformy .NET Framework podporují.

Rozhodli jsme klientské aplikace implementovat jako desktopové aplikace, přestože by výše zmíněná volba *cloudové* varianty ukládání souborů nahrávala spíše webovému řešení.

Důvodem volby desktopové varianty bylo to, že je do budoucna plánováno rozšíření překladatelské aplikace o „*cachování*“ překládaných souborů, což by umožnilo *offline* práci překladatelů a vyřešilo problémy uvedené v podkapitole 4.2.2 výše. Toto rozšíření se bude lépe implementovat v desktopové variantě než ve webové variantě.

4.3 Propojení aplikačního serveru a klientských aplikací

Jednou ze zásadních součástí v našem systému je síťová komunikace mezi aplikačním serverem a klientskými desktopovými aplikacemi, proto bylo nutné zvolit vhodné technické řešení. Nabízí se několik možností jak zajistit komunikaci mezi těmito dvěma komponentami. Nyní představíme technologie nad kterými jsme uvažovali:

Vlastní protokol

Implementace vlastního komunikačního protokolu by nám umožnil pokrýt všechny naše požadavky. Na druhou stranu navrhnout a implementovat spolehlivý síťový protokol není triviální záležitost, která je zcela mimo rozsah této práce.

.NET Remoting

.NET Remoting je technologie vzdáleného zveřejnění objektů a volání jejich metod (viz dokumentace na webu firmy Microsoft [5]) mezi dvěma procesy (ať už v rámci jednoho počítače nebo vzdáleně po síti). Výhodou je programátorský komfort, kdy v klientském kódu volání vzdálené metody vypadá v zásadě stejně jako místní volání. .NET Remoting je vhodný pro *tightly coupled* aplikace, kdy potřebujeme, aby jedna aplikace běžela distribuovaně.

WCF (Windows Communication Foundation)

WCF [4] je nástupnická technologie nahrazující .NET Remoting (firma Microsoft – tvůrce .NET Remotingu – doporučuje pro další vývoj použít WCF místo .NET Remotingu, viz dokumentační stránky [5]). WCF má stejný programátorský komfort jako .NET Remoting, zmíněný výše. Oproti němu má však WCF lepší podporu zabezpečení komunikace (například, aby .NET Remoting využíval protokolu HTTPS, tak vyžaduje hostování serverové aplikace v IIS – Internet Information Services, web server od firmy Microsoft). WCF také podporuje možnost zveřejnit aplikační rozhraní služby jako webové API, tedy je možné volat metody skrze webový prohlížeč, nejenom v kódu. Tuto možnost pravděpodobně nevyžijeme, ale volba této technologie nám neuzavírá možnost v budoucnosti rozšířit náš aplikační server o webové API.

Na základě porovnání výše jsme se rozhodli pro WCF. Jedná se vhodné řešení pro *klient-server* architekturu, což je naše situace. Velkou výhodou WCF je podpora pro autentizaci klientů, v případě, že použijeme některý ze zabezpečených protokolů (například HTTPS). V následující podkapitole se budeme zabývat, tím, co bylo třeba udělat pro fungování WCF. Také nám WCF umožňuje v budoucnosti vyvíjet další klientské aplikace, které by využívali stávající API, a to i klidně v jiných jazycích než je C#.

4.3.1 Práce s WCF

Když jsme začínali pracovat s WCF, tak jsme nejprve naimplementovali dvě prototypové aplikace, kde jedna simulovala chování serveru a jedna chování klienta. V těchto aplikacích jsme testovali základní funkčnost WCF frameworku.

Ve WCF se serverová komponenta implementuje, tak že serverem zveřejňované aplikační rozhraní je definováno rozhraním (nyní máme mysli konkrétní konstrukt z jazyka C#), ve kterém definujeme jednotlivé zveřejňované metody.

Protože jsme vyvíjeli dvě klientské aplikace, rozhodli jsme se pro každou z nich mít jedno rozhraní, místo jednoho, které by obsahovalo definice všech metod, které budou aplikace potřebovat. Každé z těchto rozhraní bude obsahovat metody, které bude naše klientská aplikace potřebovat. Překladač bude typicky potřebovat metody, které mu dovolí stahovat a nahrávat soubory, které mu budou přiřazeny, případně u nich měnit některé metainformace (například možnost označit soubor jako přeložený). Projektový manažer bude potřebovat metody k přidání a editaci souborů, uživatelů a projektů.

Výhodou je, že v případě přidání dalších metod do rozhraní, není třeba znova kompilovat klientský kód, tedy je zachována zpětná kompatibilita. Nutnost další kompilace je vyžadována pouze při změně signatur nebo implementace stávajících metod na serveru.

Během naší s WCF práce na projektu jsme narazili na tyto dva problémy:

- Jména parametrů volané metody musí být stejná jak na straně volajícího, tak i na straně serveru. V případě, že se jméno parametru liší, tak na straně serveru tento parametr dostane výchozí hodnotu (přestože se přenese po síti, hodnota je zahozena až na straně severu, ve chvíli kdy WCF framework překládá přijatou zprávu na místní volání metody). Tato situace se velice špatně odhaluje, protože problému si můžeme všimnout například až později při pohledu do databáze, kdy v ní máme jiné než očekávané hodnoty. Tento problém se může objevit i přestože máme jedno rozhraní sdílené mezi kódem aplikačního serveru a klientských aplikací. Dojde k tomu například v situaci, kdy máme již aplikační server odladěný a nasazený na staging serveru a pouze vyvíjíme klientskou aplikaci a v rámci úprav změníme v rozhraní i jména parametrů.

- Ve chvíli, kdy jsme klientskou aplikaci ladili, tak bylo nutné mít v ladícím režimu spuštěný i server, protože informace o tom, že na straně serveru vznikne výjimka se u klienta projeví pouze výjimkou `FaultException`, která v sobě nenese původní výjimku.

4.4 Zabezpečení uživatelů a jejich přihlašování

Protože vyvíjený systém bude uchovávat překládané soubory, což jsou citlivá data, tak je potřeba zabezpečit přístup k nim (viz cíl 1e). WCF Framework přímo poskytuje podporu pro zabezpečené spojení mezi komunikujícími stranami, pokud použijeme vhodný *binding*. Ve WCF se konfigurace použitého transportního protokolu (TCP, HTTP, HTTPS, NamedPipes apod.), kódování jednotlivých zpráv (například dle různých standardů *Web Services*) a další nastavení chování frameworku nazývá souhrnně *binding*. My jsme potřebovali použít některý z *bindingů*, který podporuje zabezpečenou komunikaci. Rozhodli jsme se použít *wsHttpBinding*, protože se jedná o Microsoftem doporučovaný *binding* pro zabezpečenou komunikaci v Internetu. *wsHttpBinding* šifruje na úrovni jednotlivých zpráv, které kóduje pomocí standardu Web Services a podporuje autentizaci klientů pomocí jména a hesla.

4.4.1 Autentizace

Uživatele autentizujeme pomocí přihlašovacího jména (dále budeme označovat jako login) a hesla. Uživatel při spuštění klientské aplikace zadá login a heslo, tyto údaje jsou bezpečně přeneseny pomocí *wsHttpBindingu* na server, kde je ověříme. V souladu s bezpečnostními zásadami uživatelská hesla neuchováváme v *plaintextové* podobě. Hesla hashujeme pomocí kryptografické funkce PBKDF2 (*Password-Based Key Derivation Function 2*), která je určena právě k tomuto účelu (viz standard od společnosti *RSA Laboratories* doporučující PBKDF2 viz [9]). Každé heslo je navíc hashováno společně s náhodně zvoleným (pro každé heslo jiným) řetězcem nazývaným *salt*, který zajistí, že i kdyby dva uživatelé měli stejné heslo, tak výsledný hash bude jiný. Další parametrem PBKDF2 je počet iterací, který určuje náročnost výpočtu hashe hesla. Tento počet iterací je možné v budoucnu zvyšovat s tím, jak poroste výpočetní kapacita. Hash hesla a *salt* řetězce, *salt* samotný a počet iterací ukládáme do databáze (viz kapitola 4.7 věnující se databázové části) ke každému uživateli zvlášť a tedy při každém volání služby jsme schopni uživatele autentizovat porovnáním hashů. V současné verzi aplikace hesla hashujeme pouze na straně serveru, což ochrání uživatelská hesla pouze v případě úniku naší databáze. Tento přístup ovšem už nechrání uživatelská hesla v případě, že by se útočníkovi podařilo získat privátní klíč serveru a dešifrovat komunikaci mezi klientskou aplikací a aplikačním serverem. Do budoucna by bylo jistě vhodné hashovat uživatelská hesla také v klientské aplikaci, abychom tomuto předešli.

Abychom ve WCF mohli používat šifrování, bylo třeba vytvořit pro server vlastní certifikát. To, že komunikujeme skutečně s naším serverem poznáme v

klientské aplikaci porovnáním přijatého veřejného klíče od serveru s klíčem uloženým přímo v klientské aplikaci. Dále je také dát pozor pokud přecházíme na verzi .NET frameworku 4.6.1 a vyšší ze starší verze (což se nám stalo), tak je třeba do konfiguračního souboru přidat tyto řádky:

```
<AppContextSwitchOverrides
value=
"Switch.System.IdentityModel.DisableMultipleDNSEntriesInSANCertificate=true"
/>
```

Pokud zde nejsou, tak u certifikátů s více SAN (*Subject Alternative Name*) záznamy WCF framework v klientské aplikaci odmítne certifikát a vyhodí výjimku `MessageSecurityException`, přestože naše ověření identity aplikačního serveru probíhá na základě kontroly veřejného klíče (implementujeme vlastní `X509CertificateValidator`).

4.4.2 Autorizace

Také chceme mít možnost autorizovat jednotlivá volání metod, protože máme dva druhy uživatelů – *překladaatele* a *projektové manažery*. Překladaatelé mohou stahovat a nahrávat pouze jim přiřazené soubory. Navíc chceme v aplikaci rozlišovat tyto tři úrovně projektového manažera:

1 Super

Manažer na úrovni **Super** může vše. Tedy mazat, vyvážet a editovat všechny projekty v systému a tvořit, mazat a editovat všechny typy úrovně uživatelských účtů.

2 Advanced

Manažer na úrovni **Advanced** může mazat, vyvážet a editovat všechny projekty v systému, ale může vytvářet pouze nové překladaatelské účty.

3 Junior

Manažer na úrovni **Junior** může vidět všechny projekty a soubory v systému, ale editovat a mazat může pouze soubory a projekty, které sám vytvořil. Nemůže vyvážet žádné uživatelské účty.

WCF obsahuje podporu pro autorizaci – metody, které chceme kontrolovat, anotujeme atributem `[PrincipalPermission(SecurityAction.Demand , Role = "ADMIN")]`, kdy v parametru atributu `Role` je řetězec (v příkladu je to řetězec "ADMIN"), který popisuje roli, kterou musí volající mít pro úspěšné autorizování volání. Tuto roli bychom měli například uloženou v databázi.

Tato autorizace bohužel není pro naše účely dostatečná, protože nedovoluje mít možnost autorizovat jednotlivá volání metod i na základě hodnot předaných parametrů. Naším cíle je mít možnost například zamítnout požadavek od překladaatele na stažení jemu nepřijíženého souboru nebo zamítnout požadavek na vytvoření nového uživatelského účtu od projektového manažera úrovně Junior. Proto musíme provádět autorizaci teprve až na začátku volané metody, tedy ve chvíli kdy máme k dispozici předané parametry. V současnosti je tento přístup,

kdy je všechna logika autorizace v kontrolním kódu (role ani oprávnění neukládáme do databáze) dostačující. V případě růstu počtu úrovní manažerů (rolí) a jednotlivých oprávnění (akcí k autorizování), by bylo vhodné přejít na autorizaci pomocí rolí, oprávnění a aktivit, která je sice složitější ale o to flexibilnější.

4.5 Společný formát překládaných souborů

Vzhledem k tomu, že `.sdlxliff` soubory v sobě obsahují binárně serializovaný původní dokument (viz kapitola 2.2) a tedy mohou být velmi velké, přestože se v nich nemusí vyskytovat mnoho textu k přeložení. Naopak formát TMX je úspornější, protože obsahuje pouze překládaný text a formátovací tagy. Protože díky *cloudovému* stylu ukládání souborů (viz 4.2.2) si bude překladatelská aplikace stahovat soubor při každém spuštění, bylo by velice neefektivní pokaždé po síti stahovat `.sdlxliff` soubor pouze kvůli textu. Proto jsme se rozhodli použít TMX jako editační formát, kdy pro každý SDLXLIFF soubor bude také v databázi uložen příslušný (z něj vygenerovaný) `.tmx` soubor. Tento `.tmx` soubor bude obsahovat pouze text a definice formátovacích tagů – tedy pouze to nejn nutnější, co potřebuje překladatel k práci. Tento `.tmx` soubor bude editován překladatelem a původní `.sdlxliff` zůstane netknutý. Pouze ve chvíli kdy bude projektový manažer chtít vygenerovat přeložený `.sdlxliff` soubor, tak manažerská aplikace ze serveru stáhne oba soubory (`.tmx` i `.sdlxliff`) a do `.sdlxliff` souboru doplní cílový text z `.tmx` souboru.

4.5.1 Mapování SDLXLIFF a TMX

Aby výše zmíněný postup fungoval museli jsme namapovat na sebe `.sdlxliff` soubory a `.tmx` soubory. Mapování podrobně popíšeme z pohledu převodu SDLXLIFF formátu do formátu TMX. Toto mapování jsme odvodili z našeho chápání formátu SDLXLIFF, tak aby výsledný `.tmx` soubor co nejvíce odpovídal grafickému zobrazení `.sdlxliff` souborů SDL Trados Studiu. Mapování je zachyceno na ukázce 4.5.1. Dále předpokládáme, že se čtenář již seznámil s kapitolami 2.3 a 2.2, které se věnují podrobnému popisu obou XML formátů.

Synovské zdrojové `mrk` elementy – tedy ty obsažené v `seg-source` elementu se mapují na jeden `tuv` element v `.tmx` souboru. Mapují se pouze ty `mrk` elementy, které obsahují nějaký přeložitelný text, pokud jsou prázdné nebo obsahují pouze formátovací tagy, tak je ignorujeme. Tento `tuv` element má pak v `.tmx` nastaven zdrojový jazyk. Podobně jsou mapovány případné cílové `mrk` elementy, tedy ty obsažené v elementu `target`, také se mapují na `tuv` element, ale tentokrát mají tyto `tuv` elementy nastaven jazyk cíle.

Dále je nutné převést reprezentaci formátovacích tagů v `.sdlxliff` do `.tmx`. Element `x` se mapuje na element `ph`. Atribut `id` mapujeme na atribut `type`. Otevírací část XML elementu `g` se mapuje na `bpt` a jeho uzavírací část na `ept`. Tyto výsledné `bpt` a `ept` elementy nesmíme zapomenout v spárovat pomocí atributu `i`, v ukázce si můžeme všimnout, že jako identifikátor jsme použili řadové číslovky, ale principiálně můžeme použít jakýkoliv řetězec (viz 2.3). Stejně tak jsme museli

spolu napárovat i formátovací tagy napříč jednotlivými elementy `tuv` pomocí atributu `x`, zase si v ukázce můžeme všimnout, že jsme použili zase řadové číslovky, stejně tak bychom mohli ale například použít hodnotu atributu `id` z původního `.sdlxliiff` dokumentu. Také bychom chtěli připomenout, že to, že v ukázce mají atributy `x` a `i` stejnou hodnotu nemá žádný důvod – tyto atributy mají zcela odlišný význam (viz podkapitola 2.1) – jedná se pouze o shodu náhod.

Ukázka 4.5.1: Převod segmentu z SDLXLIFF do TMX

```
<!-- vstupní SDLXLIFF -->
...
<seg-source>
  <mrk mtype="seg" mid="337">
    V domě <g id="390">jsou</g> tři pokoje.<x id="391"/>
  </mrk>
</seg-source>
<target>
  <mrk mtype="seg" mid="337">
    <g id="390">There are</g> three rooms in the house.<x id="391"/>
  </mrk>
</target>
...
<!-- výstupní TMX -->
...
<tu>
  <tuv xml:lang="cz">
    <seg>
      V domě <bpt type="390" x="1" i="1"/>jsou<ept i="1"/> tři pokoje.<ph
        ↳ type="391" x="2"/>
    </seg>
  <tuv xml:lang="en">
    <seg>
      <bpt type="390" x="1" i="1"/>There are<ept i="1"/> three rooms in the
        ↳ house.<ph type="391" x="2"/>
    </seg>
  </tuv>
</tu>
...
```

Během tohoto převodu jsme se museli vypořádat se některými odpozorovanými vlastnostmi formátu SDLXLIFF, které jsme zmiňovali v podkapitole 2.2.

- Elementy `mrk`, které neobsahují žádný text, ignorujeme.
- Jako segmenty chápeme pouze ty elementy `mrk`, které mají atribut `mid` a atribut `mtype` s hodnotou „seg“. Ostatní elementy `mrk` ignorujeme – respektive ignorujeme pouze element, ale jeho obsah (jak text, tak podelementy) dále interpretujeme. Můžeme si to představit tak, jako kdybychom tento element v XML dokumentovém stromu nahradili jeho potomky.
- V případě, že jsou elementy `mrk` obaleny nějakými tagy (takovou situaci jsme mohli vidět v kapitole 2.2, kde v ukázce 2.2.1 je takto obalen segment, jehož atribut `mid` má hodnotu „186“), tak tyto tagy ignorujeme. Výhodou je, že při spojování SDLXLIFF a TMX (které bylo rozebráno v úvodu této pod-


kapitoly 4.5) zůstanou tyto obalovací tag zachovány – nahradíme původní segment za námi nově vygenerovaný segment.

4.6 Klientské aplikace

Dle cílů 2 a 3 jsme chtěli vyvinout dvě grafické klientské aplikace. Aplikaci pro projektového manažera a aplikaci pro překladatele. Do obou aplikací se bude možné přihlásit jménem a heslem. V aplikaci pro projektového manažera bude jedna obrazovka ve které zobrazíme seznam projektů (a souborů v nich), dále by zde měla být tlačítka a dialogy pro přidání nového souboru, stažení přeloženého souboru ve formátu SDLXIFF nebo TMX, smazání souboru, přidání nového projektu, přidání nového uživatele apod. V překladatelské aplikaci by se po přihlášení měla zobrazit tabulka se seznamem souborů přidělených danému překladateli. Z této tabulky si překladatel vybere soubor, který chce přeložit a ten si otevře v grafickém překladatelském editoru. Dle cíle 3b, by tento editor by měl ideálně být podobný překladatelskému editoru v SDL Trados Studiu (jeho vzhled můžeme vidět na obrázku 2.1.1) – tedy seznam dvojic segmentů (každá dvojice na jednom řádku), kdy zdrojový segment bude v levém sloupci a nepůjde editovat a cílový segment bude v pravém sloupci a bude editovatelný. Zároveň budeme chtít nějak vizuálně odlišit formátovací tagy od ostatního textu.

4.6.1 Volba technologie

Pro tvorbu grafického rozhraní se na platformě .NET nejběžněji používají frameworky *Windows Forms* a *WPF* (*Windows Presentation Foundation*). Výše zmíněné požadavky na funkci grafického editoru týkající se správy souborů a uživatelů jsou běžné pro spoustu aplikací a jejich implementace by měla být možná v obou grafických frameworkích. Naopak ohledně překladatelského editoru jsme museli zvážit možnosti, které nám grafické frameworky nabízejí, protože se nejedná o standardní komponentu. Naše požadavky na překladatelský editor byly následující:

- Budeme potřebovat komponentu, která bude reprezentovat *cílový* segment. Bude možné v ní zobrazit a zároveň **editovat** text a překladatelské formátovací tagy. Tyto tagy by měly být reprezentovány vizuálně odlišným způsobem od textu. Ideální by bylo možné zobrazit překladatelský tag podobně jako v SDL Trados Studiu (například takto ).
- Zdrojový segment bude reprezentován podobným způsobem jako *cílový* (viz výše), pouze nebude povolena editace textu a tagů.
- Budeme potřebovat komponentu, která bude obsahovat předem neznámé množství položek. Každá z položek bude obsahovat výše zmíněné komponenty pro reprezentaci *zdrojového* a *cílového* segmentu – tedy celá položka reprezentuje jednu řádku v překladatelském editoru. Těmito položkami by mělo být možné rolovat. Také by měla tato komponenta být schopná zvládnout obsluhovat řádově tisíce položek (v běžném překládaném souboru je

typicky od stovek po nízké desítky tisíc segmentů k přeložení).

Nyní popíšeme možnosti implementace překladatelského editoru v jednotlivých frameworkích:

WPF Jako komponenta pro editování textu se nabízí `RichTextBox`, ve kterém je možné editovat text a také přímo do textu vkládat (a také je z něj odstraňovat) libovolné grafické elementy (potomky třídy `UIElement`). Jednou z výhod WPF je snadná tvorba vlastních komponent, která by měla umožnit formátovací tagy reprezentovat pomocí nějaké vlastní komponenty, u které budeme mít plnou kontrolu nad jejím vzhledem. Pro zobrazení jednotlivých dvojic segmentů se hodí komponenta `ListBox`, která umožňuje zobrazovat libovolné množství námi definovaných grafických prvků. `ListBox` současně podporuje takzvanou *virtualizaci*, tedy by měl zvládnout obsloužit velké množství překládaných segmentů.

Windows Forms Ve *Windows Forms* se komponenta, která umožňuje editovat text a zároveň vkládat jiné než textové prvky také jmenuje `RichTextBox`. Její nevýhodou je to, že na rozdíl od `RichTextBoxu` ve *WPF* do textu umí vkládat pouze obrázky – tedy bychom museli naše formátovací tagy reprezentovat pomocí obrázků. V případě, že bychom chtěli mít pro každý z formátovacích tagů obrázek s podrobným popisem tagu (informace o tazích v *SDLXLIFF* bývají poměrně detailní, kromě toho, že se jedná například o tag `cf`, mohou obsahovat další informace o *fontu*, *velikosti* apod.), tak bychom museli tento obrázek generovat za běhu z popisu tagu – nebylo by možné použít předem připravené obrázky. Další nevýhodou *Windows Forms* je absence podpory pro zobrazování seznamu vlastních grafických prvků. Dále také ve *Windows Forms* není taková podpora pro *virtualizaci* při obsluze velkého množství položek (doporučený postup vývojářskou komunitou (viz odkaz [11]) je použít hostovaný `ListBox` z *WPF* – v tuto chvíli by postrádala volba *Windows Forms* smysl).

Na základě výše popsaných implementačních možností jsme se rozhodli zvolit grafický framework *WPF*, protože je pro naše plánované použití vhodnější.

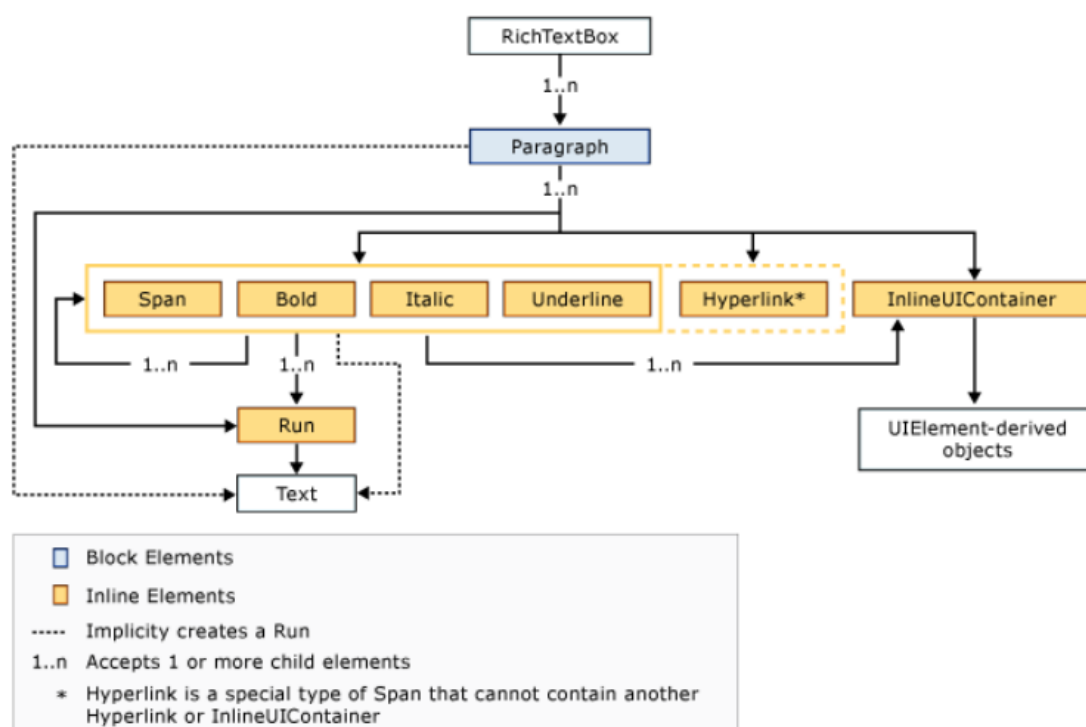
4.6.2 Překladatelská aplikace

Workflow v překladatelské aplikaci vypadá následovně. Po spuštění se zobrazí dialog se jménem a heslem a uživatel (překladatel) se přihlásí. Po přihlášení je překladateli zobrazena tabulka s jemu přidělenými soubory, z těch si jeden vybere a ten může upravovat v editoru. Toto můžeme vidět na konci této podkapitoly na obrázcích 4.6.3 a 4.6.4, které ukazují vzhled současné verze překladatelské aplikace.

Grafická reprezentace formátovacích tagů

Jako vhodnou komponentu pro zobrazování textu a formátovacích tagů jsme zvolili `RichTextBox` (jak již bylo zmíněno v podkapitole 4.6.1), protože do této komponenty lze vkládat mezi text i jakékoli potomky `UIElement`, což si nyní

podrobněji popíšeme. Na obrázku 4.6.1 je diagram obsahu (Content) komponenty `RichTextBox` na který se teď budeme odkazovat. `RichTextBox` obsahuje jeden a více instancí třídy `Paragraph`, které obsahují jeden a více instancí třídy `Run`, která reprezentuje samotný text. Vedle instancí třídy `Run` může `Paragraph` také obsahovat potomky třídy `UIElement` (zabalené do `InlineUIContainer`). Této vlastnosti jsme využili a formátovací tagy reprezentujeme pomocí třídy `Button`. Volba třídy `Button` je pouze dočasná a má demonstrovat to, že jsme schopni formátovací tagy rozumně reprezentovat a pracovat s nimi v editovatelném textu. V další verzi aplikace by bylo vhodné namísto `Buttonu` použít jinou komponentu – nejlépe přímo vytvořit vlastní komponentu pro reprezentaci tag. Text budeme reprezentovat standardně pomocí instancí třídy `Run`.

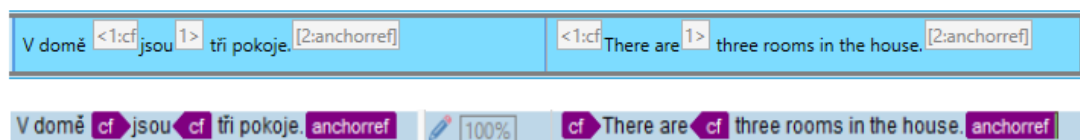


Obrázek 4.6.1: Diagram struktury obsahu komponenty `RichTextBox` (převzato z MSDN [3])

Dále jsme také chtěli dát překladateli možnost vkládat formátovací tagy ze zdrojového segmentu do cílového segmentu. Bohužel při kopírování pomocí klávesové zkratky `Ctrl+C` se zkopíruje pouze text, `InlineUIControler` reprezentující tagy je bohužel zahozen. Proto jsme do editoru přidali tlačítka a *drop-down* seznam, které umožní překladateli vybrat si formátovací tag ze zdrojového segmentu a vložit ho na aktuální pozici kurzoru v editovaném *cílovém* segmentu. Zde bylo nutné ošetřit všechny možné krajní pozice kurzoru, protože některých situacích se `RichTextBox` rozhodne přidat nový `Paragraph`, či rozdělit souvislý text na více `Runů` případně se nacházíme úplně mimo `Paragraph` (to se stane například ve chvíli, kdy překladatel zvolí celý obsah cílového segmentu a smaže ho) apod. Nepodařilo se nám zjistit, co přesně toto chování způsobuje.

Na obrázku 4.6.2 můžeme vidět srovnání toho, jak vypadá překládaný segment v našem editoru a jak vpadá v SDL Trados Studiu. Verze SDL Trados Studia vy-

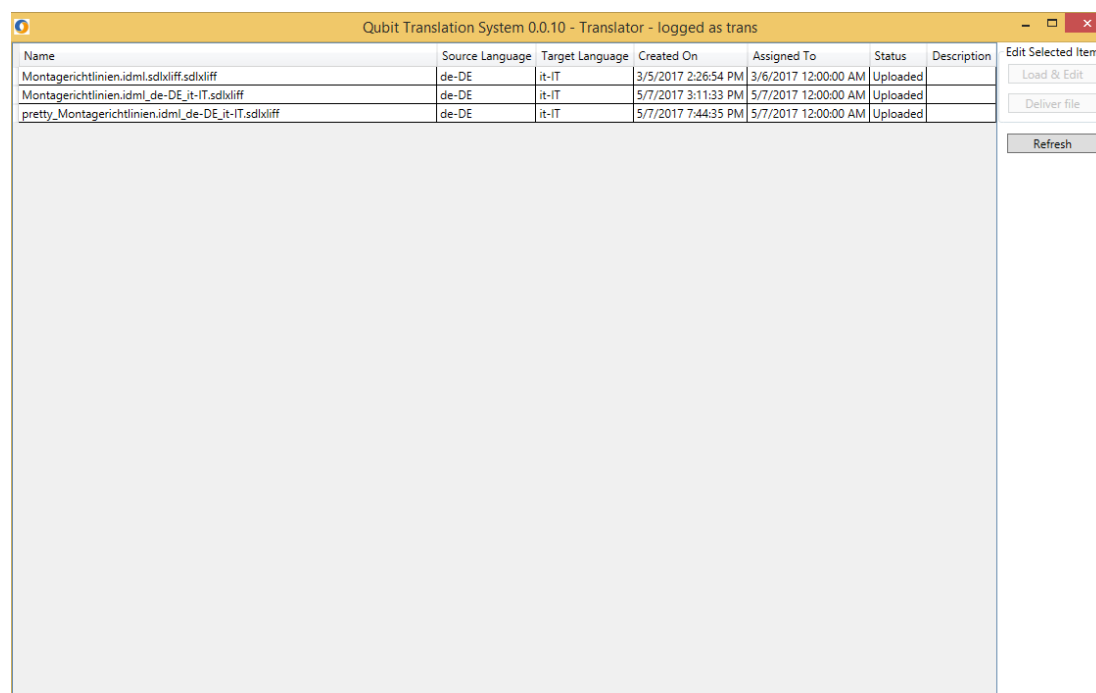
padá lépe, ale na druhou stranu v našem editoru je nyní možné velice jednoduše změnit vzhled formátovacích tagů – stačí nahradit použitý **Button** vlastní grafickou komponentou, jak již bylo zmíněno. Vylepšení grafického vzhledu editoru je určitě dalším krokem vývoje aplikace. Můžeme ale konstatovat, že základní funkcionality překladatelského editoru byla úspěšně implementována, tak aby se na ní dalo dále úspěšně stavět. Výsledný vzhled celého editoru můžeme vidět na obrázku 4.6.4.



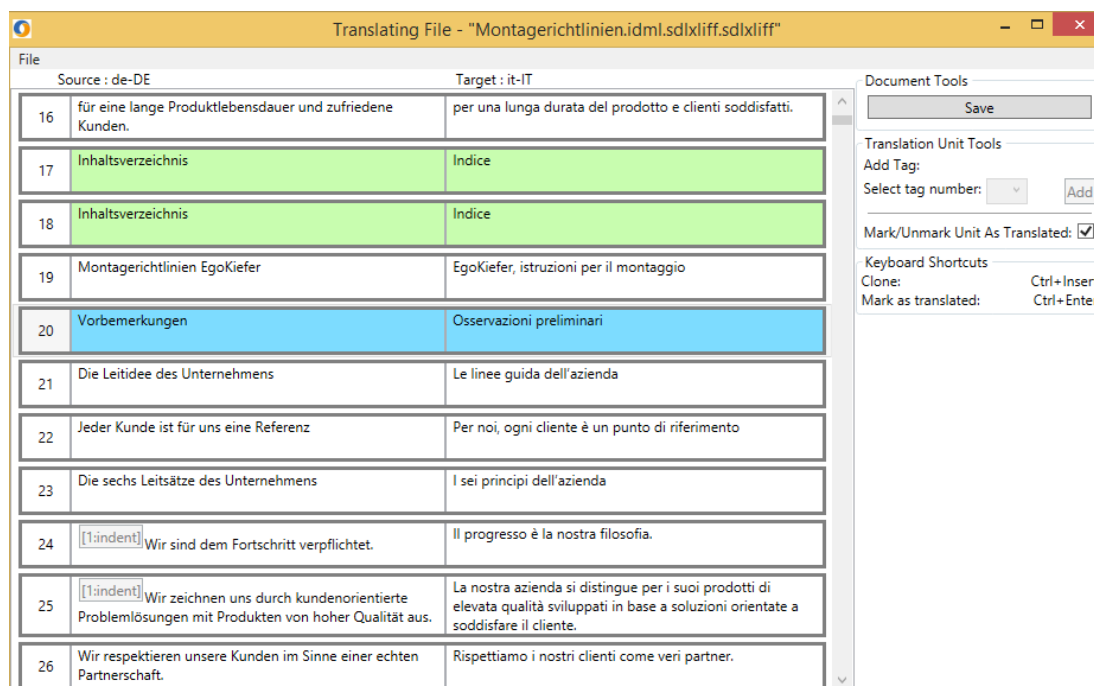
Obrázek 4.6.2: Porovnání vzhledu překládaného segmentu v našem editoru (*nahore*) a v SDL Trados Studiu (*dole*)

Dále bychom chtěli zmínit, že v překládaných souborech se objevuje velké množství překládaných segmentů – v řádu tisíců. V našem editoru každý překládaný segment prakticky znamená dvě instance třídy **RichTextBox**, které patří mezi grafické komponenty náročnější na systémové zdroje. Proto bylo nutné využít *virtualizačních* možností komponenty **ListBox** (ve které jsme **RichTextBoxy** zobrazovali), která dokáže načítat pouze zobrazované grafické položky s rozumnou odezvou.

Nyní na obrázcích 4.6.3 a 4.6.4 uvádíme základní hlavního okna překladatelské aplikace a překladatelského editoru.



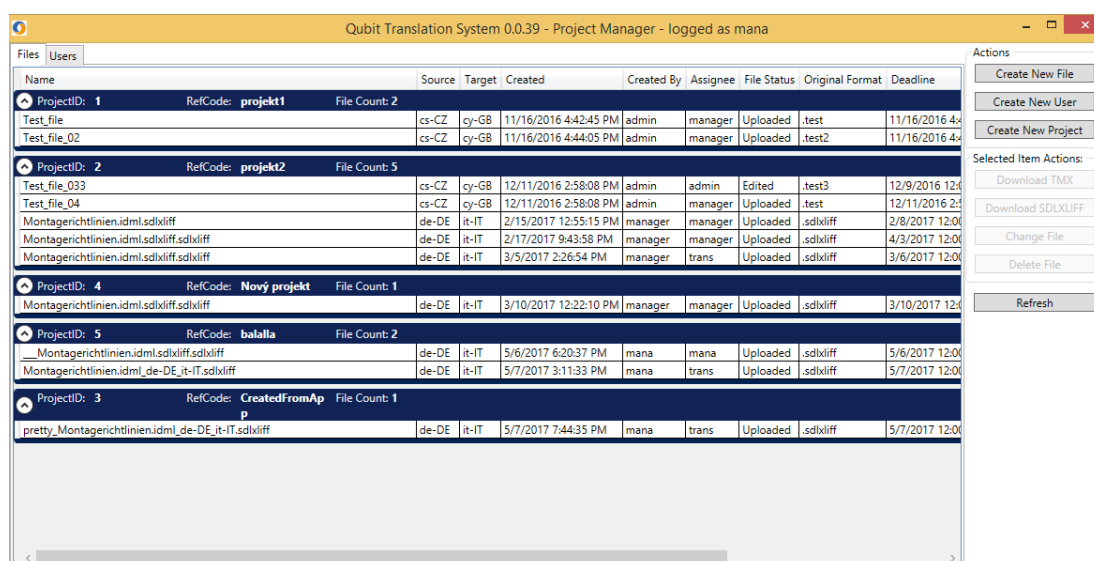
Obrázek 4.6.3: Hlavní okno aplikace pro překladaatele



Obrázek 4.6.4: Překladačský editor (*zelené* řádky jsou segmenty, které byly označeny jako přeložené, *modrý* řádek je právě editovaný)

4.6.3 Aplikace pro projektového manažera

Aplikace pro projektového manažera má podobný workflow jako aplikace pro překladače. Po spuštění se zobrazí dialog se jménem a heslem a uživatel se přihlásí. Projektovému manažerovi se zobrazí tabulka se všemi soubory sdruženými podle projektů a nabídkou možných akcí, jak můžeme vidět na obrázku 4.6.5, který uvádíme pro ukázkou vzhled současné verze aplikace.



Obrázek 4.6.5: Hlavní okno aplikace pro projektového manažera

4.7 Datová vrstva

V této kapitole se budeme zabývat implementací databázové vrstvy. V kapitole 4.2.1 jsme se rozhodli datové úložiště implementovat pomocí databáze. Dle zadání budeme ukládat tyto data:

1. Seznam všech uživatelů a informací o nich. Tedy jméno, login, heslo, datum vytvoření, práva uživatelů apod.
2. Seznam všech projektů a metadat s nimi spojených (datum vytvoření, přiřazené dokumenty, uživatel, který projekt vytvořil) apod.
3. Samotné překládané soubory a příslušné metainformace, tedy samotný soubor, datum vytvoření, přiřazený překladatel, datum předpokládaného dokončení apod.

4.7.1 Databázové schéma

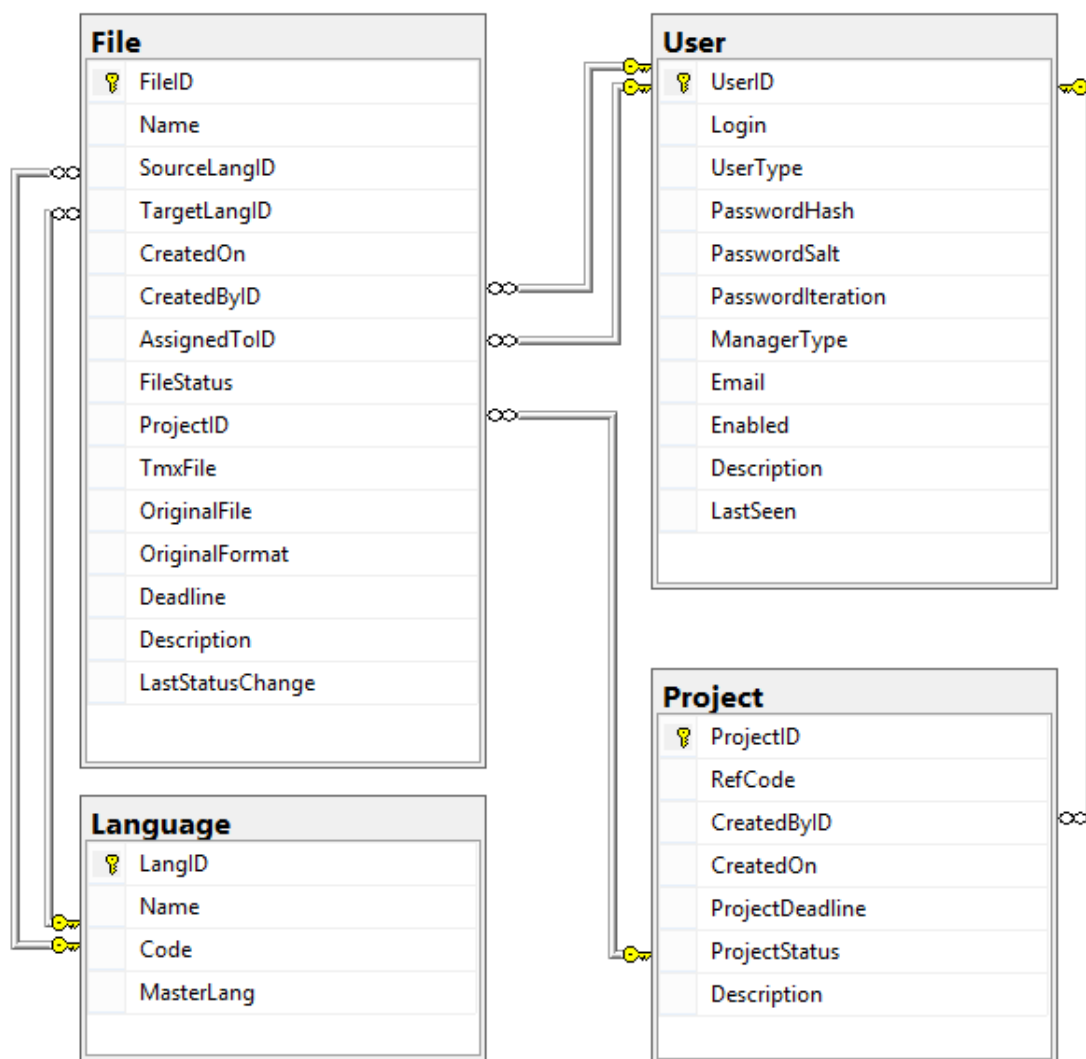
Na základě cílů a požadavků na ukládaná data viz výše, jsme rozhodli vytvořit databázové schéma, které můžeme vidět na obrázku 4.7.1.

Data uživatelů jsou v tabulce `User`. Uchováváme zde základní údaje (`Login`, `Email`, `Description`, `Enabled`, `LastSeen`) a autentizační údaje (`PasswordHash`, `PasswordSalt`, `PasswordIteration`), které jsme rozebírali v kapitole 4.4.1 a autORIZAČNÍ údaje (`UserType`, `ManagerType`), kterým jsme se věnovali v kapitole 4.4.2. Je nutné podotknout, že datové položky `LastSeen` a `Enabled` nejsou v aplikaci zatím používány.

V tabulce `Language` je seznam známých *locale*, jejich ISO kódů, jazykových kódů a jejich plných jmen.

V tabulce `File` jsou uloženy binárně samotné překladatelské soubory a metainformace o nich.

Tabulka `Project` slouží k logickému členění souborů na projekty.



Obrázek 4.7.1: Diagram databázového schématu

Volba databázové technologie

Vzhledem k tomu, že naše databázové schéma je relativně jednoduché a ne-
očekáváme vysoký počet požadavků, tak by pro naše účely měla stačit většina
databázových systémů. Volili jsme mezi dvěma nejpoužívanějšími komerčními da-
tabázemi *Oracle Database* a *Microsoft SQL Server*. Obě dvě existují v bezplatné
Express verzi, která je limitována výkonem. *Oracle Database* povoluje využití jed-
noho procesoru (bez ohledu na počet jader), 1GB RAM a 11GB dat. *Microsoft*
SQL Server povoluje maximálně čtyři jádra, 1GB RAM a 10GB dat – vidíme
tedy, že obě bezplatné verze nabízejí skoro stejné možnosti. V zadavatelské agen-
tuře používají technologie od firmy Microsoft, což byl jeden z důvodů proč jsme
se rozhodli zvolit *Microsoft SQL Server*. Také pokud by nám v budoucnosti ne-
dostačovala Express verze, tak placené verze *Microsoft SQL Serveru* jsou levnější
než ty od *Oracle*. Dále pro *Microsoft SQL Server* existuje grafický nástroj pro
správu databáze (*Microsoft SQL Server Management Studio*), který je uživatel-
sky přívětivější než *SqlDeveloper* používaný pro *Oracle* databázi.

4.7.2 Přístup do databáze z kódu

.NET framework v sobě obsahuje ovladače pro komunikaci s *Microsoft SQL Serverem*. Protože API těchto ovladačů není úplně programátorsky komfortní, tak jsme rozhodli použít nějaké z existujících řešení, které by nám měla práci s databázemi zjednodušit. Zvažovali jsme tyto dvě varianty:

- *Entity Framework* je vyspělý ORM (*Object Relation Mapping*) framework s velkým množstvím funkcí. *Entity Framework* si stáhne databázové schéma a poté nám podle něj vytvoří třídy v kódu. *Entity framework* také podporuje přístup *code-first*, kdy generuje za uživatele celé databázové schéma podle tříd v kódu. *Entity Framework* také zbavuje programátora také nutnosti psát SQL dotazy přímo do kódu, protože dotazy jsou generovány přímo za běhu aplikace. Nevýhodou je složitější konfigurace a možná neefektivita generovaných SQL dotazů. Je nutné podotknout, že firma Microsoft vydala novou verzi *Entity Frameworku* (nazvanou *Entity Framework Core*), která výrazně zjednodušuje nasazení. V době implementace této práce *Entity Framework Core* ještě nebyl ve stabilní verzi.
- *Dapper.NET* je takzvaný *micro* ORM framework. Na rozdíl od „velkého“ ORM frameworku nepodporuje migrace jednotlivých databázových schémat ani generování SQL dotazů. Tedy stále programátor musí psát SQL dotazy ručně do kódu. Výhodou *Dapper.NET* nicméně je automatická serializace vrácených výsledků z databáze přímo do objektů, což velice urychluje programátorskou práci. Také se programátor nemusí starat o *Dapper.NET* také podporuje parametrizaci SQL dotazů a tedy nesvádí programátora k tomu dělat svojí aplikaci zranitelnou vůči útokům typu *SQL injection*.

Nakonec jsme zvolili pro *Dapper.NET* a to hlavně kvůli snadnosti jeho nasazení do kódu, kdy stačí přidat příslušnou knihovnu. Ve chvíli, kdy pak potřebujeme nějaká data z databáze, tak stačí zavolat generickou metodu knihovny *Dapper.NET*, které předáme jako argument řetězec s SQL dotazem, objekt s parametry daného dotazu a pokud nám má databáze vrátit nějaká data, tak jako typový argument předáme POCO (*Plain Old CLR Object*) třídu, do které chceme výsledky serializovat. *Dapper.NET* na rozdíl od *Entity Frameworku* nevyžaduje žádné anotace u tříd ani použití *designeru*. Další výhodou *Dapper.NET* je efektivita, dle porovnání viz [1], je *Dapper.NET* až 10x rychlejší než *Entity Framework*.

Poučení

Zpětně hodnotíme použití *Dapper.NET* negativně, protože ladění kódu s SQL dotazy se ukázalo jako velice časově náročné a neflexibilní vůči jakýmkoliv změnám. Vyhýbání se psaní SQL dotazů přímo do kódu nás vedlo k tomu používat SQL *stored procedury*, což ve výsledku způsobilo přesun části *business logiky* i do datové části, čemuž jsme se chtěli na začátku práce vyhnout (viz 4.2.1). Je třeba také podotknout, že správa SQL kódu je velice náročná, protože se v zásadě nedá nijak rozumně uchovávat ve správě zdrojového kódu. Také se ukázalo, že dotazy do databáze provádíme většinou pouze sporadicky (na žádost uživatele), takže se zde neprojevila výhoda větší výkonosti *Dapper.NET*. Pokud by

se s vývojem aplikace mělo pokračovat, tak považujeme za vhodné znova zvážit použití *Entity Frameworku*, zbavit se SQL dotazů v kódu a přesunout jakýkoliv kód z databáze zpět na aplikační server. Zároveň by se mělo znovu zvážit použití *Entity Framework Core*.

4.8 Instalátor a aktualizace

Abychom mohli vyzkoušet fungování systému i v Internetu, tak jsme museli Aplikační server nasadit na veřejně dostupný server. Kvůli usnadnění testování jsme se rozhodli využít virtuálního hostingu u firmy *Forpsi s.r.o.*, kdy za měsíční poplatek uživatel získá přístup k virtuálnímu stroji a nemusí se tedy starat o konfiguraci hardware apod. Tento přístup je realizován pomocí RDP (Remote Desktop Protocol, protokol od firmy Microsoft určený ke vzdálenému přístupu k počítači s operačním systémem Windows). Technickými podrobnostmi nasazení (jako je například konfigurace portů na serveru, instalace certifikátu nebo rezervace adres) se zabýváme v implementační části v kapitole 6.1.

U aplikačního serveru není až tak důležitá snadnost nasazení, protože konfigurace a správa aplikačního serveru bude prováděna pravděpodobně zkušeným uživatelem, nejspíše IT administrátorem. Naopak pro grafické klientské aplikace je nutné aby jejich instalaci zvládl i běžný uživatel. Proto jsme se rozhodli obě grafické vybavit instalátorem. Zároveň jsme se rozhodli do každé z těchto klientských aplikací dodat komponentu, která vždy na začátku startu aplikaci zkontroluje jestli neexistuje novější verze aplikace a případně nabídne uživateli možnost aktualizace. Také jsme museli zavést číslování jednotlivých verzí aplikace, abychom je mohli od sebe rozlišit. Toto nám umožní rychle reagovat na změny a rychle opravovat chyby.

Naimplementovali jsme komponentu, která se při startu podívá jestli existuje novější verze a případně si z FTP serveru stáhne a spustí nový instalátor, který umí existující aplikaci aktualizovat. Podobný přístup k aktualizacím používá například i program *Notepad++*.

Existuje celá řada možností jak vytvořit instalátor. Vyvíjet vlastní instalátor považujeme za příliš náročné. Vybrali jsme proto rozšíření do Visual Studio *Installer Project*, který se snadno konfiguruje, ale hlavně podporuje aktualizaci již nainstalovaných aplikací. Jeho nevýhodou je, že je potřeba mít nějakou z vyšších verzí Visual Studio (*Professional* nebo *Enterprise*). Abychom nemuseli před každým vytvořením instalátoru ručně měnit na více místech verzi aplikace, implementovali jsme program, který před spuštěním generování upraví soubory *Installer Project* konfigurace, tak aby sedělo číslo verze instalace a programu.

5. Vývojová dokumentace

V této kapitole se budeme věnovat technickému popisu toho, jak vypadá vlastní implementace celého softwarového řešení. Po přečtení této kapitoly by měl mít čtenář přehled o struktuře celého systému. Souborová struktura se dá rozdělit na dvě části, na databázovou část, která obsahuje skripty pro vytvoření databáze a vložení počátečních dat a nachází se v adresáři `/dbscripts` (těm se věnujeme v 6.1) a hlavní část s kódem (které se budeme věnovat v této kapitole), což je Visual Studio *solution*, které se nachází v adresáři `/src/Qubit`.

5.1 Struktura Visual Studio solution

Projekty v *solution* se dají rozdělit do tří kategorií:

- Projekty jejichž výstupem jsou spustitelné assembly, které reprezentují jednotlivé funkční součásti celého systému:
 - **MiddleTier** je projekt pro aplikační server.
 - **ProjectManagerClient** je projekt pro aplikaci projektového manažera.
 - **TranslatorClient** je projekt pro aplikaci projektového manažera.
- Sdílené projekty (jejich výstupem jsou nespustitelné assembly (*class library*), která obsahuje třídy, rozhraní a algoritmy použité v aplikačních projektech zmíněných výše):
 - **Shared** je sdílený projekt, pro sdílenou *class library*, který obsahuje komponenty, které se používají v projektech **MiddleTier**,
 - **Formats** je sdílený projekt, který obsahuje třídy pro reprezentaci a transformaci formátů TMX a SDLXLIFF.
- Projekty pro tvorbu instalátorů:
 - **TranslatorSetupProject** slouží k tvorbě instalátoru překladatelské aplikace vzniklé zkompileváním projektu **TranslatorClient**.
 - **ProjectManagerSetupProject** slouží k tvorbě instalátoru manažerské aplikace vzniklé zkompileváním projektu **ProjectManagerClient**.
 - **AssemblyInfoUtil** neslouží přímo k tvorbě instalátorů, ale jedná se o podpůrnou utilitu při tvorbě instalátorů. Přečte z konfigurace jednoho z výše zmíněných *installer* projektů produktovou verzi a tu zapíše do **AssemblyInfo.cs** kompilované aplikace. Volání této utility je nastaveno jako *prebuild event* u projektů **ProjectManagerClient** a **TranslatorClient**.

V následujících kapitolách podrobněji představíme jednotlivé projekty.

5.2 Projekt MiddleTier

Zkompilováním projektu *MiddleTier* získáme spustitelnou assembly, která funguje jako aplikační server. Základní kostra je jednoduchá, spustí se hostující služba WCF frameworku a zaregistruje se v ní třída `RemoteService`, která implementuje rozhraní zveřejňované aplikačním serverem. Toto rozhraní se nachází v projektu *Shared* (viz níže), který je sdílený i s projekty pro klientské aplikace. Tento projekt závisí pouze na projektu *Shared*. Nyní popíšeme podrobněji některé z komponent, které jsou v tomto projektu důležité.

RemoteService

Třída `RemoteService` implementuje obě klientská rozhraní. V zásadě každá z metod v této třídě má následující strukturu – autorizace akce na základě identity volajícího a předaných parametrů (o autentizaci se stará WCF framework, viz 4.4.1) a provedení samotné akce, což je většinou buď zápis nějakých hodnot do databáze nebo naopak vrácení dat z databáze. Příklad takové metody, která slouží ke smazání souboru, můžeme vidět na ukázce 5.2.1. Pro komunikaci s databází používáme ORM framework *Dapper.NET*, jehož volbu rozebíráme v podkapitole 4.7.2.

Ukázka 5.2.1: Typická struktura metody ve třídě `RemoteService`

```
public ServiceResponse DeleteFile(int deletedFileId)
{
    if (!CanEditFile(deletedFileId))
    {
        Log.SecurityLog.Error("Caller \"" + Caller + "\" tried to delete file \""
            + deletedFileId + "\" without authorized access.");
        return ServiceResponse.Fail("Unauthorized access.");
    }
    int rowsaf;
    using (var c = new SqlConnection(ConnectionString))
    {
        rowsaf = c.Execute("DELETE FROM [File] WHERE FileID = @id",
            new { id = deletedFileId });
    }
    if (rowsaf == 1)
    {
        return ServiceResponse.SuccessResponse();
    }
    else
    {
        return ServiceResponse.Fail("Error happened, no record was deleted.");
    }
}
```

Při procházení samotných zdrojových kódů v solution si můžeme všimnout, že v současné verzi nejsou ještě některé metody implementovány (například automatické vygenerování uživatelského hesla jeho zaslání na zadaný uživatelský

e-mail), jedná se o metody rozšiřující základní funkcionalitu systému, které budou implementovány v budoucích verzích. Také by pro usnadnění budoucího vývoje bylo vhodné extrahovat autorizační funkcionalitu z `RemoteService` do samostatné třídy, abychom případně mohli používat ve třídě `RemoteService` různé autorizační strategie, aniž bychom jí museli měnit.

AuthenticationValidator

Třída `AuthenticationValidator` implementuje samotnou autentizaci – tedy počítá hash z přijatého hesla (pomocí standardní implementace hashovací funkce PBKDF2, viz podkapitola 4.4.1) a ten porovnává s hashem v databázi. Tuto třídu musíme do WCF frameworku zaregistrovat v souboru `App.config` v sekci `behaviours`, jak můžeme vidět v ukázce ze souboru z `App.config` uvedeném níže. Můžeme si všimnout, že je třeba použít celé jméno třídy a jméno assembly. Dále si můžeme všimnout, že zde také specifikujeme certifikát, který pak bude server používat pro svojí autentizaci.

```
...
<serviceCredentials>
  <userNameAuthentication
    userNamePasswordValidationMode="Custom"
    customUserNamePasswordValidatorType=
      "Qubit.MiddleTier.Security.AuthenticationValidator,MiddleTier"
  />
...
</serviceCredentials>
...
```

5.3 Shared

Projekt *Shared* je určen pro sdílení tříd mezi ostatními projekty. Jde o podpůrné třídy používané v obou klientských aplikacích nebo o třídy použité pro komunikaci mezi klientskými aplikacemi a aplikačním serverem a nebo o třídy modelující tabulky v databázi. V následujících kapitolách tyto části popíšeme podrobněji.

5.3.1 Adresář ServiceData

Adresář `ServiceData` obsahuje třídy, které představují datový model uložených v databázi a nebo jsou používány jako návratové hodnoty jednotlivých metod rozhraní (níže zmíněných rozhraní). Většinou se jedná o *POCO* (*Plain Old CLR Object*) třídy.

Zde bychom chtěli poznamenat, že nebylo úplně dobrým řešením použití některých těchto tříd jak pro modelování databázových tabulek, tak pro modelování typů vyměňovaných v rámci WCF komunikace. Jedinou výhodou je málo práce

na straně aplikačního serveru, který může v zásadě objekty z databáze rovnou předávat klientovi. Platíme za to ale cenu v tom, že tyto třídy, které díky jejich provázanosti s databázovým schématem nejsou snadno upravitelné, musíme používat i v kódu klientských aplikací (viz 5.5). To poté vede k ne úplně ideálnímu návrhu jiných komponent. Do budoucna by jistě stálo zvážit pečlivé oddělení tříd pro databázový model a tříd použitých v rozhraní komunikace mezi aplikačním serverem a klientskými aplikacemi.

5.3.2 Další třídy

Níže uvádíme výčet některých významných tříd, které také nacházejí v projektu *Shared*:

- **IPMService**
Rozhraní definující metody dostupné v překladatelské aplikaci. Sdílené mezi překladatelskou aplikací (*TranslatorClient*) a aplikačním serverem (*MiddleTier*).
- **ITranslatorService**
Rozhraní definující metody dostupné v manažerské aplikaci. Sdílené mezi manažerskou aplikací (*ProjectManagerClient*) a aplikačním serverem (*MiddleTier*).
- **CustomCertificateValidator**
Tato třída je používána v klientských aplikacích pro kontrolu identity serveru. Porovnáváme zde veřejný klíč certifikátu přijatého od serveru s klíčem, který si třída načítá ze souboru `App.config`. Dále se o tom také zmiňujeme podkapitole 6.1 v administrátorské dokumentaci, která se věnuje nasazení.
- **CurrentSession**
Tato statická třída je používána v klientských aplikacích. Jedná se o globální proměnou, ve které uchováváme informace o přihlášeném uživateli, přijatá data z aplikačního serveru atd. Tato třída je používána pouze v klientských grafických aplikacích (viz kapitola 5.5).
- **AppcastConfig**
Do této třídy se deserializuje soubor `appcast.json`, který obsahuje údaje o dostupných aktualizacích apod.

5.3.3 Třída *ServiceDelegate*

Tato statická třída je používána v klientských aplikacích při volání metod na aplikačním serveru. Je to *wrapper* okolo volání metod na *proxy* objektu WCF frameworku. Díky tomu potom není uživatelský kód plný `catch` bloků na výjimky WCF frameworku. Zároveň je tato třída hezky navržena, tak aby její použití v kódu bylo co nejpodobnější použití WCF *proxy* objektu. Standardní vzdálené volání ve WCF můžeme vidět v ukázce 5.3.1. Naopak vzdálené volání pomocí *ServiceDelegate* můžeme pro srovnání vidět v ukázce 5.3.2. Tuto třídu jsme objevili na komunitním QA fóru (viz [12]) a pouze jsme si jí drobně přizpůsobili

– rozlišujeme některé další WCF výjimky, přidali jsme opětovný pokus o připojení a byla přidána vlastní kontrola certifikátu (pomocí výše zmíněného `CustomCertificateValidator`).

Ukázka 5.3.1: Volání vzdálené metody pomocí *proxy* objektu

```
// zavolání metody Foo na proxy objektu
var ch = new ChannelFactory<IService>("endpoint");
IService proxy = ch.CreateChannel();
try
{
    string result = proxy.Foo(parameter);
}
catch (ChannelTerminatedException) {...}
catch (EndpointNotFoundException) {...}
// zde by bylo mnoho dalších catch bloků reagujících na WCF výjimky
...
```

Ukázka 5.3.2: Volání vzdálené metody pomocí třídy `ServiceDelegate`

```
// zavolání metody Foo pomocí třídy ServiceDelegate
// kód, který se stará o výjimky je schován uvnitř ServiceDelegate
string result = null;
Service<IService>.Use(service => result = service.Foo(parameter));
```

Před prvním použitím třídy `ServiceDelegate` je třeba zavolat metodu `InitializeFactory`, ve které předáváme proxy objektu přihlašovací údaje uživatele, volíme *endpoint* (definovaný v `App.config`) a zároveň registrujeme do WCF frameworku třídu `CustomCertificateValidator` (výše zmíněnou).

5.4 Formats

Projekt *Formats* obsahuje třídy, které reprezentují překladatelské soubory a třídy, které je načítají. Jedná se o *class library*, která je používána v obou klient-ských aplikacích. Nyní popíšeme jednotlivé třídy které se v tomto projektu nacházejí.

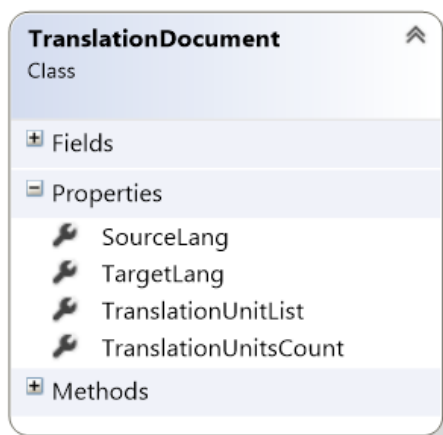
5.4.1 Třídy pro reprezentaci překladatelských dokumentů

Když načítáme dokument ve formátu SDLXLIFF nebo TMX, tak ho v kódu reprezentujeme pomocí třídy `TranslationDocument`. Tato třída není přímou reprezentací ani jednoho z XML formátů, ale oba na ní dokážeme převést. Nyní si je podrobněji představíme:

`TranslationDocument`

Překládaný soubor je reprezentován třídou `TranslationDocument`, která obsahuje seznam dvojic jednotlivých segmentů (*zdroj a cíl*), kde každá z

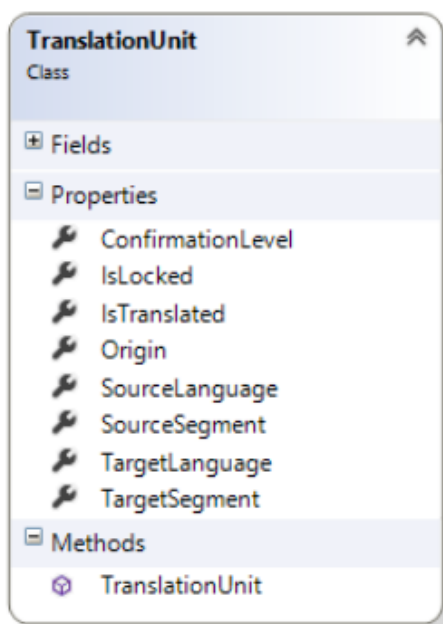
dvojic je reprezentována pomocí třídy `TranslationUnit`. Strukturu můžeme vidět na obrázku 5.4.1.



Obrázek 5.4.1: Struktura objektu `TranslationDocument`

`TranslationUnit`

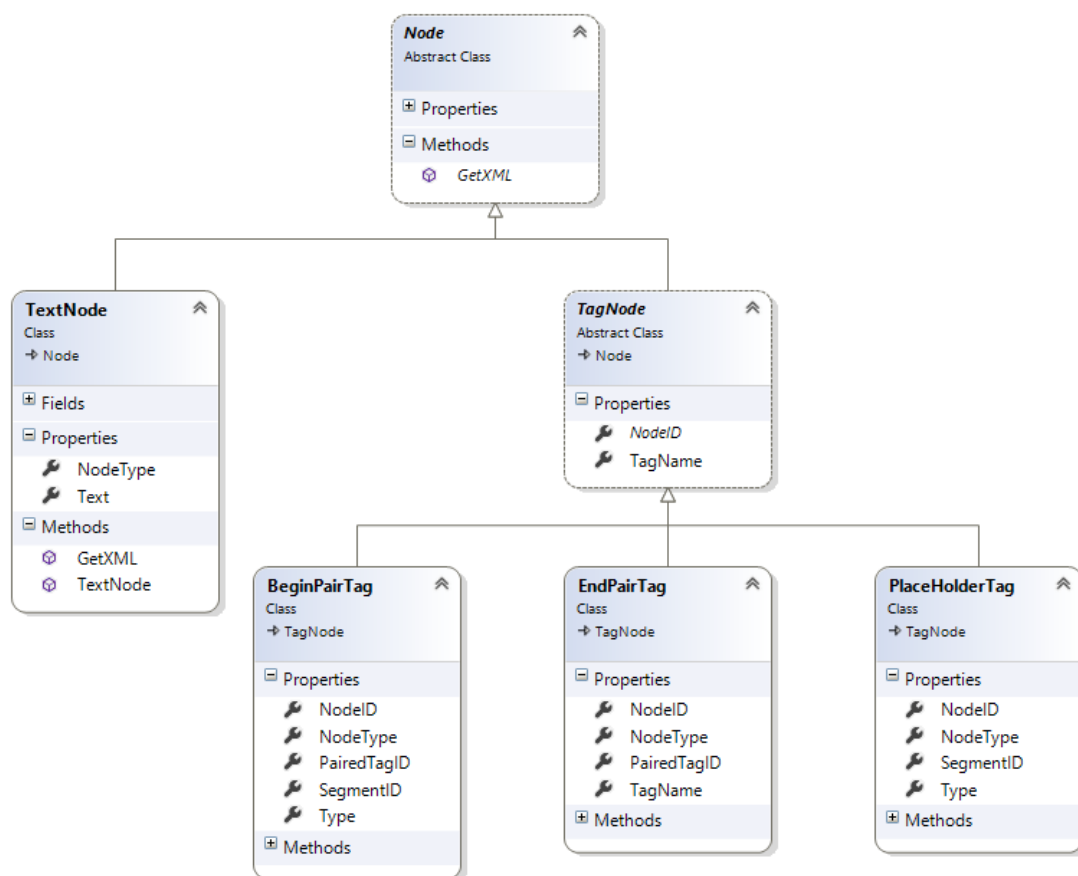
Tato třída reprezentuje dvojici segmentů, jeden zdrojový a jeden cílový, každý nich se reprezentován seznamem objektů `Node` (viz níže), které reprezentují formátovací tagy a text. Strukturu `TranslationUnit` můžeme vidět na obrázku 5.4.2.



Obrázek 5.4.2: Struktura objektu `TranslationUnit`

`Node`

Každý `Node` reprezentuje buď formátovací tag nebo samotný text, podle toho o jakého konkrétního potomka se jedná. Můžeme si všimnout, že hierarchie třídy `Node` (viz obrázek 5.4.3 níže) je podobná stylu reprezentace formátovacích tagů v TMX (viz kapitola 2.3).



Obrázek 5.4.3: Diagram hierarchie Node

5.4.2 Konvertory

Tyto třídy implementují algoritmy na načítání souborů v formátech SDLXLIFF a TMX do výše zmíněné třídy `TranslationDocument`.

TMXToTranslationDocumentConvert

Pomocí třídy `TMXToTranslationDocumentConvert` načítáme `.tmx` soubory do výše zmíněné struktury `TranslationDocument`. `TranslationDocument` je modelován podle struktury formátu TMX, díky čemuž je tento převod relativně jednoznačný. Třída je používána pouze v projektu *ProjectManagerClient*, při načítání souboru v SDLXLIFF formátu.

SDLXLIFFToTranslationDocumentConvert

Tato třída slouží k načtení `.sdlxliiff` souborů do výše zmíněné struktury `TranslationDocument`. Protože je `TranslationDocument` podobný struktuře TMX, jak již bylo zmíněno výše, tak se můžeme v zásadě omezit na mapování mezi SDLXLIFF a TMX, kterým se zabýváme v kapitole 4.5. Třída je používána pouze v projektu *ProjectManagerClient*, při načítání souboru v SDLXLIFF formátu.

SDLXLIFFUpdater

Tato třída načte originální SDLXLIFF dokument a přeložený TMX dokument (vytvořený z původního SDLXLIFFu) a poté z nich vygeneruje nový

SDLXLIFF dokument s doplněným překladem z TMX dokumentu. Je používána pouze v projektu *ProjectManagerClient*, právě pro toto slučování.

5.5 Projekty pro klientské aplikace

Klientské aplikace jsou reprezentovány dvěma projekty – *ProjectManagerClient* a *TranslatorClient*. Jedná se o grafické aplikace vyvinuté ve WPF. Obě mají relativně podobnou základní strukturu díky, tomu že i workflow v nich je podobný. Uživatel se přihlásí, poté se mu zobrazí seznam souborů a akce, které s nimi může provádět. Projektový manažer může projekty spravovat a přidávat nové naopak překladatel vidí pouze jemu přiřazené projekty a má jednu možnost – překládat. Nyní podrobněji představíme oba projekty.

5.5.1 ProjectManagerClient

Zkompilování toho projektu vznikne spustitelná aplikace pro projektového manažera. Po spuštění se připojí na aktualizací server a případně nabídne možnost aktualizace. Poté se zobrazí přihlašovací okno (*PMLoginWindow*, viz níže). Po úspěšném přihlášení si aplikace stáhne seznam uživatelů, souborů a projektů, které pak následně zobrazuje v hlavní obrazovce (*MainWindow*), ve které už může projektový manažer začít pracovat. Podrobný popis práce projektového manažera nalezneme v uživatelské dokumentaci (v kapitole 6.2.2). Níže uvádíme některé třídy z tohoto projektu:

PMLoginWindow – tato třída implementuje přihlašovací okno pro uživatele.

MainWindow – tato třída implementuje hlavní obrazovku manažerské aplikace.

Kromě seznamu souborů sdružených do projektů, je zde množství tlačítek, které umožňují vykonávat akce se zobrazenými soubory nebo otevírají některý z následujících dialogů níže.

ChangeFileDialog – dialog pro úpravy existujícího souboru.

CreateNewFileDialog – dialog pro vytvoření nového uživatele.

CreateNewProjectDialog – dialog pro vytvoření nového projektu.

CreateNewUserDialog – dialog pro přidání nového uživatele.

Dále zde jsou metody pro export souborů. Tyto metody používají již zmíněné komponenty ze sdílené assembly *Formats.dll*, která je výsledkem projektu *Formats*, viz 5.4.

5.5.2 TranslatorClient

Zkompilování toho projektu vznikne spustitelná aplikace pro překladatele. Po spuštění se připojí na aktualizací server a případně nabídne možnost aktualizace. Poté se zobrazí přihlašovací okno (*TranslatorLoginWindow*, viz níže), ve

kterém jsou zobrazeny projekty, které jsou přihlášenému uživateli přiřazeny a dostupné pro překlad. Jak již bylo zmíněno v 4.2.2, tak při každém otevření souboru se uživateli stáhne celý soubor z aplikačního serveru. Překladač je může poté překládat v integrovaném překladačském editoru. Podrobný popis workflow překladače nalezneme v uživatelské dokumentaci (v kapitole 6.2.3). Níže popíšeme třídy, které tvoří strukturu této aplikace.

TranslatorLoginWindow

Tato třída implementuje přihlašovací okno pro uživatele.

MainWindowTranslator

Tato třída implementuje hlavní obrazovku překladačské aplikace. Zde se překladači zobrazují jednotlivé přiřazené soubory a má zde možnost otevřít pro editaci.

EditFileWindow

Toto okno reprezentuje samotný editor překládaného souboru. Graficky je překládaný **TranslationDocument** (datová struktura pro reprezentaci překládaných dokumentů, viz 5.4), reprezentován **ListBoxem** ve kterém každá položka obsahuje dva **RichTextBoxy** – jeden pro zdroj a jeden pro cíl. Obsah těchto **RichTextBoxů** je kontrolován třídou **TUnitEditController**, která v podstatě převádí **TranslationUnit** na třídu **Paragraph**, který je poté vložen do příslušného **RichTextBoxu**. Tuto grafickou reprezentaci jsme podrobně rozebrali v sekci 4.6.2.

TUnitEditController

Tato třída obaluje třídu **TranslationUnit** (viz 5.4) a převádí její segmenty na **Paragraph**, který je možné pak vkládat do **RichTextBoxů** (jak bylo zmíněno výše). Převod do **Paragraphu** je následovný: objekty **TextNode** (viz 5.4.3) se transformují do **Runů** a objekty **Node**, které reprezentují nějaký formátovací tag převádíme na **Buttony** s příslušným textem. Jak již bylo řečeno v 4.6.2, třída **Button** jistě není nejvhodnějším prostředkem pro reprezentaci těchto tagů. V případě dalšího vývoje by zde měla být použita vlastní komponenta nebo alespoň **Border** s textem.

Zde bychom chtěli zmínit, že překladačská aplikace by si zasloužila jisté změny v návrhu. Bylo by vhodné kompletně vyčlenit překladačský editor do samostatné *assembly*, abychom editor mohli používat jako samostatnou komponentu, kterou by pak bylo možné například začlenit i do manažerské aplikace. V současnosti je totiž kód editoru zbytečně provázán s ostatními částmi překladačské aplikace.

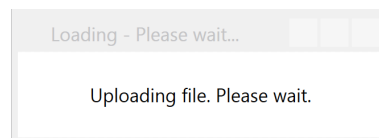
5.5.3 Instalátory klientských aplikací

Pro tvorbu instalátorů klientských aplikací používáme rozšíření do Visual Studio – *Visual Studio Installer Projects*. Toto rozšíření je k dispozici pouze pro verze Visual Studio vyšší než Community. Pro každou z klientských aplikací máme vlastní instalační projekt. Aby bylo možné aplikace nainstalovat i na počítač, kde nemá uživatel administrátorská práva, tak se aplikace instaluje do adresáře `C:\Users\uzivatel\AppData\Local\Qubit`.

Před vytvořením instalátoru, bychom měli změnit v *Properties* daného instalačního projektu položku verze (*Version*). Verze zde uvedená se totiž v rámci *prebuild eventu* kompilace samotné příslušné klientské aplikace propíše do souboru *AssemblyInfo.cs* kompilované aplikace. V této aplikaci se na začátku běhu podle této verze kontroluje, jestli je k dispozici nová verze. K tomuto slouží utilita *AssemblyInfoUtil.exe*, která je výsledkem projektu *AssemblyInfoUtil*. Dále je tato verze uvedena ve Windows nabídce „Add or remove programs“. Společně s touto verzí je nutné změnit i *ProductCode*, na základě něhož instalátor pozná, že má aktualizovat existující instalaci aplikace (pakliže taková existuje). Dále je také nutné měnit v klientských projektech v souboru *AssemblyInfo.cs* položky *AssemblyVersion* a *AssemblyFileVersion*, protože podle nich instalátor při opětovné instalaci pozná jestli má danou assembly nahradit.

5.5.4 Race condition u modálního okna

V rámci vývoje klientských aplikací jsme se také setkali s následujícím problémem. Některé operace v našich aplikacích mohou trvat déle, typicky se jedná o komunikaci se serverem či načítání většího souboru zároveň je ale možné, že v některých případech tato operace proběhne takřka hned. Tyto operace nechceme provádět v rámci kontextu *UI* (User Interface) vlákna, aby aplikace *nezamrzala*, spouštíme je tedy v rámci *Tasku*. Zároveň ale nechceme povolit uživateli možnost klikat na další tlačítka ani startovat jiné operace, chceme mu pouze zobrazit *pop-up* modální okno (viz obrázek 5.5.1), které ho informuje o tom, že probíhá operace. Zároveň ale aplikace stále reaguje na akce jako je například změna velikosti okna, zmenšení do lišty apod.



Obrázek 5.5.1: *Pop-up* modální okno informující o déletrvající operaci

Tedy po tom co vypustíme *Task* s dlouhodobější operací, tak chceme zobrazit výše zmíněné modální okno pomocí metody *ShowDialog()* (která zajišťuje modalitu zobrazovaného okna), a ve chvíli, kdy *Task* doběhne, tak toto modální okno zavřít (pomocí *Dispatcheru*, protože okno zavíráme z jiného než *UI* vlákna) a aktualizovat nějaké grafické prvky. Objevuje se nám zde totiž *race-condition*. Pokud *Task* vypustíme před otevřením modálního okna může se stát, že doběhne ještě dříve než se modální okno celé inicializuje a zavolá na něm *Close*, tak vyhodí výjimku. Stejně tak není možné vypustit *Task* až po zobrazení modálního okna, protože metoda *ShowDialog()* je blokuující metoda. Řešením bylo použití *eventu ContentRendered*, na třídě *Window*. Tento *event* nastane ve chvíli, kdy je dané okno inicializováno. Do tohoto *eventu* přidáme delegáta, který bude vypouštět náš dlouhotrvající *Task*. Tím budeme mít zajištěno, že *Task* bude vypuštěn až ve chvíli, kdy je modální okno již kompletně zobrazeno a tedy i připraveno na zavření.

6. Uživatelská dokumentace

V této kapitole jsou návody pro instalaci a použití klientských grafických aplikací. Dále je zde administrátorská dokumentace, která popisuje nasazení celého systému na vlastní infrastrukturu.

6.1 Administrátorská dokumentace

Abychom mohli aplikaci úspěšně nainstalovat budeme muset splnit následující:

- Je potřeba počítač s veřejnou IP adresou (pokud chceme, aby byl systém přístupný i z internetu) a operačním systémem Windows Server 2012 (řešení bylo testováno na Windows Server 2012, ale pravděpodobně by mělo fungovat i na klasických desktopových Windows). Na tomto systému by měl být *.NET Framework 4.6.1* a vyšší. Na tomto počítači poběží aplikační server.
- Dále potřebujeme databázový server Microsoft SQL Server, který by měl být přístupný pro systém popsany výše. Nemusí být dostupný z internetu.
- Pro distribuci aktualizací je třeba FTP server (s veřejnou IP adresou, pokud chceme aby bylo možné aktualizace distribuovat po Internetu).

Dále je nutné postupně provést následující kroky:

1 Databázový server

Na databázovém serveru spustíme postupně v uvedeném pořadí SQL skripty (nalezneme je ve složce `/dbscripts`):

- (a) `create_schema.sql` – vytvoří databázové schéma a zároveň vytvoří **databázový účet** „qubit_user“, přes který se bude aplikační server připojovat k databázi. Skript pro tento **databázový účet** nastaví heslo „hero“, které bychom měli změnit.
- (b) `seed_data.sql` – vloží seznam jazyků a vytvoří prvního uživatele (máme na mysli uživatele naší aplikace) s úrovní práv Super, aby bylo možné se k aplikaci poprvé přihlásit. Přihlašovací údaje pro tohoto uživatele jsou „manager“ a heslo je „heslo“.
- (c) `test_data.sql` – tento skript na rozdíl od dvou předchozích není povinné spouštět, ovšem pro vyzkoušení aplikace to doporučujeme. Vytvoří testovací projekty, soubory a uživatele, aby bylo možné aplikaci vyzkoušet. Přidané projekty také můžeme nalézt v příloze C. Přihlašovací údaje přidáných uživatelů jsou uvedeny v komentářích tohoto SQL skriptu.

2 Aplikační server

Pro nasazení aplikačního serveru na počítač s operačním systémem Windows Server 2012 (tento budeme dále nazývat jako *cílový počítač*) je nutné udělat následující:

- (a) Musíme vytvořit vlastní certifikát a vložit ho do certifikačního úložiště cílového počítače. My jsme použili nástroj *SelfCert* (viz příloha D), který přímo podporuje i vložení certifikátu do certifikačního úložiště Windows. V souboru *App.config* v projektu *MiddleTier* je poté třeba nastavit, kam jsme tento certifikát vložili a jak jsme ho pojmenovali, aby ho byl WCF framework schopen načíst. Například v ukázce níže vidíme, že aplikační server bude hledat certifikát podle jména „Test“ v certifikačním *storu* *LocalMachine\TrustedPeople*. Důležité je, aby zde byl pouze jeden certifikát s takovým jménem. Veřejný klíč tohoto certifikátu budeme později vkládat do *App.config* klientských aplikací, abychom mohli ověřit identitu serveru.

```
...  
<serviceCertificate findValue="Test"  
storeLocation="LocalMachine"  
storeName="TrustedPeople"  
x509FindType="FindBySubjectName"/>  
...
```

- (b) V *App.config* nastavíme do *endpointů* veřejnou IP adresu cílového počítače. V zkompilevaném aplikačním serveru (v příloze B) je jako adresa nastaven *localhost*.

```
...  
<service>  
<endpoint address="http://IP_ADDRESS:26718/trans">  
...  
...
```

- (c) V *App.config* v sekci *connectionStrings* upravíme *connection string* s jménem *db*. Stačí zde pouze zadat adresu našeho databázového serveru a heslo, které jsme nastavili v kroku 1 výše.
- (d) Na počítači na kterém aplikační server poběží musíme spustit příkazy:
- ```
netsh http add urlacl url=http://+:port/trans/ user=pc\usr
netsh http add urlacl url=http://+:port/pm/ user=pc\usr
```
- kde:

„**pc**“ je jméno či adresa stroje na které aplikační server poběží.

„**usr**“ je Windows login uživatele pod kterým aplikační server poběží.

„**port**“ je námi zvolený port na kterém bude služba WCF poslouchat. My jsme zvolili například 26718 (jak můžeme vidět v kroku 2b). Je možné, že bude třeba otevřít tento port ve firewallu.

- (e) Zkompilujeme projekt *MiddleTier* (viz 5.2) a na cílovém počítači spustíme soubor *MiddleTier.exe* v adresáři:
- ```
/csharp/Qubit/MiddleTier/bin/Release.
```
- (f) Aplikační server nyní běží.

3 Klientské aplikace

Pro vytvoření instalátorů klientských aplikací, které můžeme už distribuovat koncovým uživatelům (*překladačům a projektovým manažerům*) je třeba udělat následující:

- (a) V souborech `App.config` pro obě klientské aplikace v sekci pro konfiguraci WCF `endpointu` nastavíme IP adresu a port aplikačního serveru z kroku **2b**. V zkompileovaných klientských aplikacích (v příloze **B**) je jako adresa nastaven *localhost*.

```
...  
<client>  
<endpoint address="http://IP_ADDRESS:26718/trans">  
...  
...
```

- (b) Dále v `App.config` pro obě klientské aplikace v sekci `appSettings` nastavíme pod klíč `appCastUrl` FTP adresu souboru `appcast.json` s aktualizacími údaji (podrobnosti aktualizace jsou rozvedeny v podkapitole **5.5.3**).

```
...  
<appSettings>  
<add key="appCastUrl"  
  ↪ value="ftp://FTP_SERVER_IP/appcast.json"/>  
...
```

- (c) Dále v `App.config` pro obě klientské aplikace v sekci `appSettings` nastavíme pod klíč `pk` hodnotu hexadecimální reprezentaci veřejného klíče použitého na serveru, jak již bylo zmíněno v kroku **2a**. Tuto hodnotu si pak načítá třída `CustomCertificateValidator`, která kontroluje identity serveru a nachází se v projektu *Shared* (jak bylo popsáno v podkapitole **5.3**).

```
...  
<appSettings>  
<add key="pk" value="Hex value of the server's public key"/>  
...
```

- (d) V obou klientských projektech (*ProjectManagerClient* a *TranslatorClient*) můžeme nastavit produktovou verzi aplikace v `AssemblyInfo.cs` (toto je podrobně rozvedeno v **5.5.3**).
- (e) Zkompilujeme projekty (*ProjectManagerSetupProject* a *TranslatorSetupProject*) pro tvorbu instalátorů. jejich výstupem jsou instalátory, které již můžeme již distribuovat uživatelům.

4 Aktualizační FTP Server

Pro správné fungování aktualizací funkcionality je třeba provést následující. Na FTP server nahrajeme instalátory klientských aplikací z bodu **3e**

a cesty k těmto souborům vložíme do souboru `appcast.json` (viz ukázka níže), zároveň zde uvedeme správná čísla verzí klientských aplikací. Tento soubor poté zveřejníme na adrese, kterou jsme použili v bodě **3b**.

```
{
  "MinVersionPM" : "0.0.0",
  "LatestVersionPM" : "0.0.38",
  "MinVersionTranslator" : "0.0.0",
  "LatestVersionTranslator" : "0.0.9",
  "PMInstallerLocation" :
    ↪ "ftp://FTP_SERVER_IP/ProjectManagerSetup.msi",
  "TranslatorInstallerLocation" :
    ↪ "ftp://FTP_SERVER_IP/TranslatorSetup.msi"
}
```

Jako FTP Server jsme používali *BabyFTP Server* (viz příloha **D**). Jedná *lightweight* implementaci FTP s minimem počáteční konfigurace a pro naši potřebu je naprosto dostačující.

5 Hotovo

Nyní by měl být celý systém funkční. Pravděpodobně se nejprve budeme chtít přihlásit jako administrátorský uživatel uvedený v kroku **1** a vytvořit ostatní reálné uživatele. Poté si již překladatelé a projektoví manažeři už jenom mohou nainstalovat aplikaci a začít pracovat. Popis instalace a uživatelská dokumentace klientských aplikací se nachází ve složce `/data/userdoc`.

Tento postup vypadá relativně složitě. Na druhou stranu, po počátečním nastavení, kdy se některé korky provedou pouze jednou, je už nasazení dalších verzí či změny nastavení podstatně jednodušší. v následující podkapitole uvádíme postup pro testovací nasazení.

6.1.1 Testovací nasazení na *localhostu*

Pokud by chtěl čtenář vyzkoušet nasadit naši aplikaci aniž by cokoliv kompiloval, tak může použít buď zkompilevané aplikace nebo instalátory klientských aplikací viz příloha **B**.

Stále **je nutné provést většinu kroků** z administrátorské dokumentace (viz kapitola **6.1** výše). Je možné vynechat pouze některé kroky, kde se nastavují hodnoty v `App.config`.

Dále je nutné mít na paměti následující:

- Stále **je nutné** v `App.config` aplikačního serveru nastavit *connection string* použitého databázového serveru.
- Zkompileovaný aplikační server bude zveřejňovat rozhraní na *localhostu*.
- Zkompileovaný aplikační server hledá certifikát se jménem „Test“ ve *storu LocalMachine\TrustedPeople*. Důležité je, aby zde byl pouze jeden certifikát s takovým jménem.

- Zkompilované klientské aplikace očekávají, že aplikační server se bude prokazovat certifikátem `/install/cert.pfx` (viz příloha B). Tento certifikát musíme na aplikačním serveru importovat do správného *storu*, aby ho aplikační server našel (podrobně rozebráno v administrátorské dokumentaci výše).

6.2 Klientské aplikace

V této části je uživatelská dokumentace pro samotné koncové uživatele našeho systému.

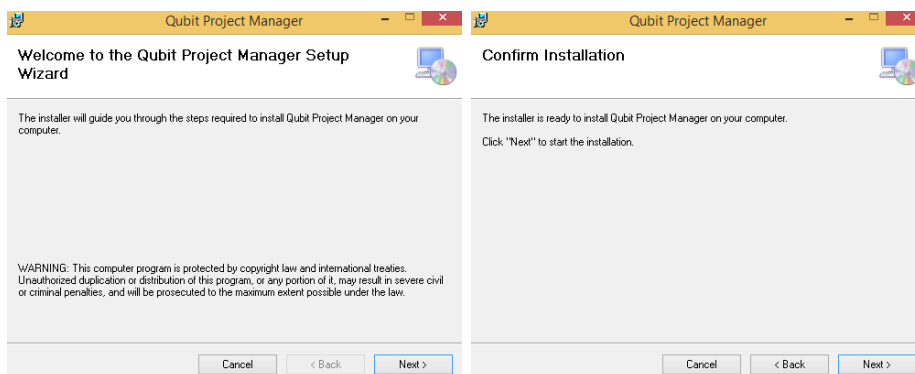
6.2.1 Instalace

Instalátory obou klientských aplikací jsou si velmi podobné. Pro každou klientskou aplikaci máme vlastní instalátor:

`ProjectManagerSetup.msi` – instalátor manažerské aplikace.

`TranslatorSetupProject.msi` – instalátor překladatelské aplikace.

Podle instalované aplikace dvakrát poklepeme na příslušný instalační soubor, který nám zaslal IT administrátor naší překladatelské agentury. Instalace se sestává pouze ze dvou oken (která můžeme vidět níže na obrázku 6.2.1), kdy v obou klikneme na tlačítko **Next**.

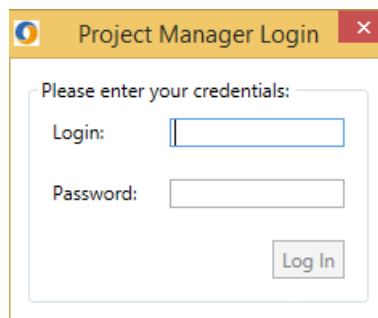


Obrázek 6.2.1: Instalační obrazovky

Je třeba poznamenat, že pokud bychom chtěli použít předpřipravené instalátory z přílohy B (adresář `/install`), je třeba mít na paměti to, že v nich zabalené aplikace očekávají specifické nasazení systému popsané v 6.1.1

6.2.2 Aplikace pro projektového manažera

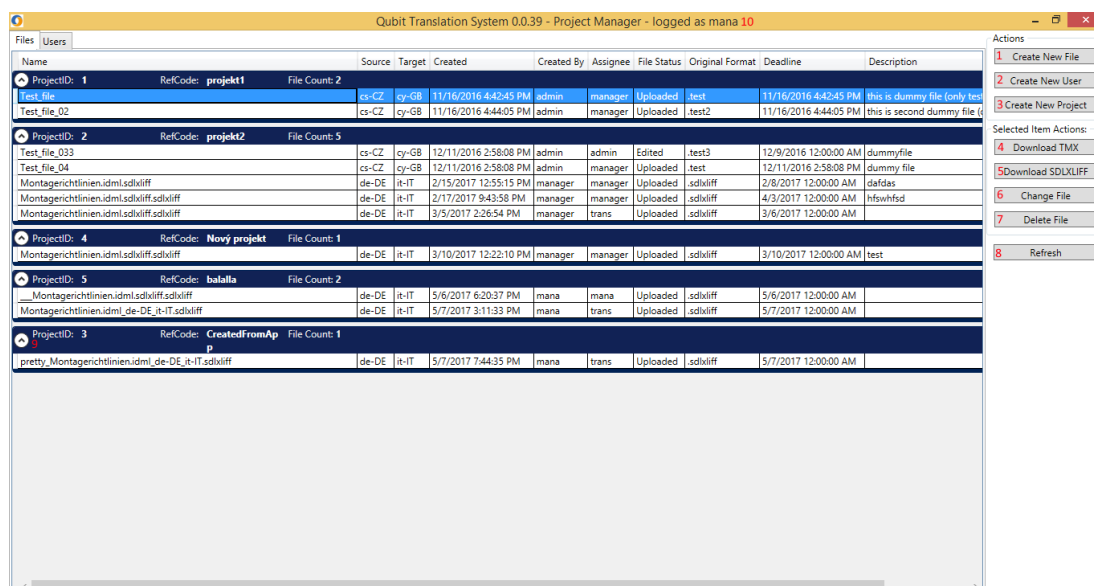
Práce s manažerskou aplikací vypadá následovně. Nejprve se zobrazí přihlašovací okno (viz obrázek 6.2.2). Tam vyplníme přidělený login a heslo. Přihlašovací údaje by nám měl přidělit administrátor.



Obrázek 6.2.2: Přihlašovací dialog

Hlavní obrazovka

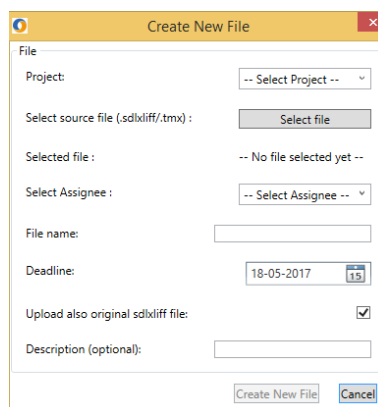
Po úspěšném přihlášení se zobrazí hlavní obrazovka (viz obrázek 6.2.3).



Obrázek 6.2.3: Hlavní obrazovka manažerské aplikace

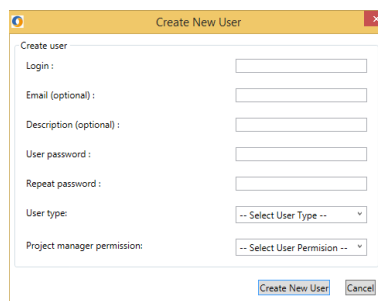
Na obrázku 6.2.3 jsou jednotlivé prvky očíslovány červenými číslicemi. Nyní vysvětlíme co jednotlivé číslice znamenají:

- Po kliknutí se objeví nový dialog pro přidání nového souboru (viz obrázek 6.2.4). V současnosti jsou podporovány formáty SDLXLIFF a TMX. Pro vyzkoušení je možné zkusit nahrát některé z testovacích souborů z přílohy C.



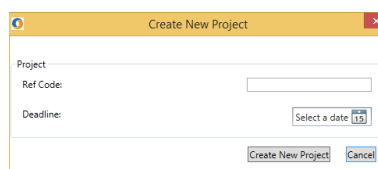
Obrázek 6.2.4: Dialog pro přidání nového souboru.

2. Po kliknutí se objeví nový dialog pro přidání nového uživatele.



Obrázek 6.2.5: Přidání nového uživatele.

3. Po kliknutí se objeví dialog pro přidání nového projektu.



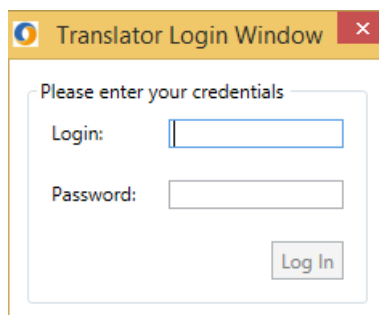
Obrázek 6.2.6: Přidání nového projektu.

4. Slouží pro stažení přeloženého TMX souboru. Jak mohou vypadat výsledné TMX soubory, které vznikají při použití této funkcionality je možné pro ilustraci najít v příloze C.
5. Slouží pro stažení přeloženého SDLXLIFF souboru. Jak mohou vypadat výsledné SDLXLIFF soubory, které vznikají při použití této funkcionality je možné pro ilustraci najít v příloze C. Je možné je porovnat s originálními soubory (z přílohy C), ze kterých vznikly.
6. Otevře dialog pro změny informací o právě zvoleném souboru.
7. Smaže vybraný soubor.
8. Zaktualizuje zobrazené údaje z cloudu.

9. Sbalí/rozbalí soubory v příslušném projektu.
10. Zde se zobrazuje verze aplikace a jméno právě přihlášeného uživatele.

6.2.3 Aplikace pro překladatele

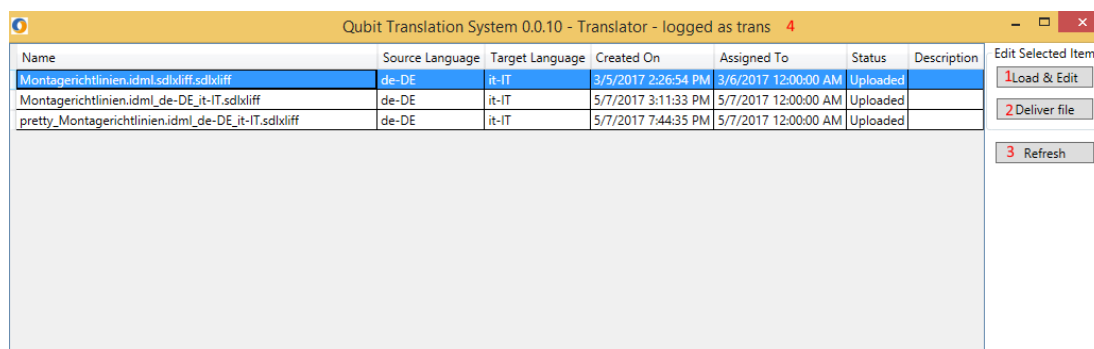
Práce v překladatelské aplikaci vypadá následovně. Nejprve se zobrazí přihlašovací okno (viz obrázek 6.2.7). Tam vyplníme přidělený login a heslo. Přihlašovací údaje by nám měl přidělit administrátor.



Obrázek 6.2.7: Přihlašovací dialog

Hlavní obrazovka

Po úspěšném přihlášení se zobrazí hlavní obrazovka (viz obrázek 6.2.8) se soubory přiřazenými přihlášenému překladateli.



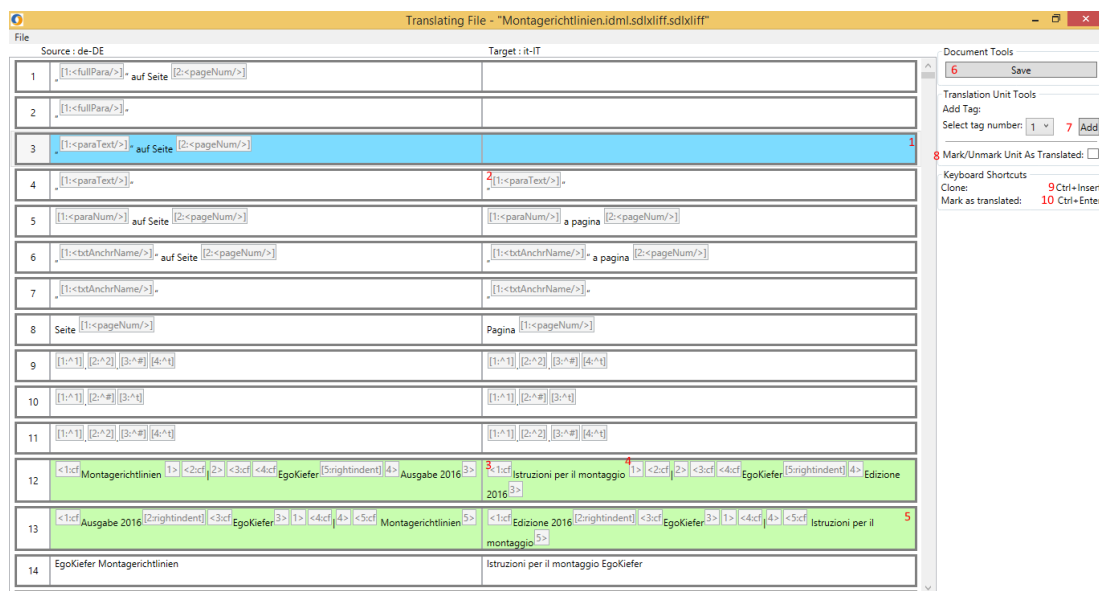
Obrázek 6.2.8: Hlavní obrazovka překladatelské aplikace

Na obrázku 6.2.8 jsou jednotlivé prvky očíslovány červenými číslicemi. Nyní vysvětlíme co jednotlivé číslice popisují:

1. Otevře právě vybraný soubor (zamodřený řádek v tabulce) překladatelském editoru (viz níže).
2. Označí právě vybraný soubor jako přeložený, podle toho pak projektový manažer pozná, že je překlad už hotový.
3. Zaktualizuje zobrazené údaje s cloudem.
4. Zde se zobrazuje verze aplikace a jméno právě přihlášeného uživatele.

Překladatelský editor

Potom co jsme v předchozí obrazovce zvolili, který soubor budeme chceme editovat, tak se nám otevře v překladatelském editoru, který můžeme vidět na obrázku 6.2.9, kde jsou jednotlivé prvky očíslovány červenými číslicemi. Nyní vysvětlíme co jednotlivé číslice popisují:



Obrázek 6.2.9: Hlavní obrazovka manažerské aplikace

1. Modrá řádka označuje právě editovaný segment.
2. Takto je reprezentován nepárový formátovací tag.
3. Takto je reprezentovaná párová otevírací část formátovacího tagu.
4. Takto je reprezentovaná párová uzavírací část formátovacího tagu.
5. Zelená řádka znamená, že řádka již byla přeložena.
6. Uloží soubor do cloudu.
7. V rolovacím menu si vybereme formátovací tag z právě vybraného překládaného (zdrojového) segmentu (můžeme si všimnout, že se nám v rolovacím menu nabízí pouze čísla tagů, které nachází v právě vybraném zdrojovém segmentu) a poté pomocí tlačítka „Add“ se tento tag přidá do právě vybraného cílového segmentu na místo aktuální pozice kurzoru.
8. Označí/odznačí právě překládaný segment za přeložený (řádka je zelená).
9. Klávesová zkratka *Ctrl+Insert* zkopíruje vše z zdrojového segmentu do cílového segmentu – týká se právě vybrané (modré) řádky.
10. Klávesová zkratka *Ctrl+Enter* označí právě vybranou řádku jako přeloženou a skočí na další nepřeloženou řádku.

Závěr

V závěru této práce zhodnotíme, jak se nám podařilo naplnit cíle práce, které jsme vytyčili v podkapitole 1.5. Vyvinuli jsme softwarové řešení sestávající se z databázové části, grafické aplikace pro projektové manažery, grafické aplikace pro překladatele s integrovaným překladatelským editorem a aplikačního serveru, který kontroluje přístup těchto uživatelských aplikací k databázi.

Databázový systém uchovává informace o uživateli. Do systému je možné pomocí aplikace pro projektové manažery nahrávat soubory ve dvou běžně používaných překladatelských formátech (otevřený TMX a komerční SDLXLIFF).

Součástí překladatelské aplikace je grafický překladatelský editor, ve kterém je možné tyto soubory upravovat. Tento editor podporuje zobrazení a editaci formátovacích tagů. Námi testované SDLXLIFF soubory lze po úpravách zpětně nahrávat do originální aplikace SDL Trados Studio.

Obě klientské aplikace dovolují vzdálený přístup k aplikačnímu serveru přes Internet. Tato komunikace je šifrována a přístupy k aplikačnímu serveru jsou chráněny přihlašovacím jménem a heslem.

Celý systém je možné nasadit na vhodnou infrastrukturu – databázovou část na Microsoft SQL Server, aplikační server na Windows Server s veřejnou IP adresou a klientské aplikace lze instalovat na počítače s operačním systémem Windows 7 a vyšší. Přestože je zde mnoho možností, jak aplikaci vylepšit (viz odstavec níže), tak považujeme vytyčené cíle práce za splněné. Zadavatelská agentura je s výsledkem spokojena a je plánováno po určitých změnách a vylepšeních (z nichž jsou některé zmíněny níže) nasadit toto řešení do praxe.

Možné pokračování

V budoucnu bychom rádi pokračovali rozšiřováním a vylepšováním celého systému. Níže uvádíme další možná vylepšení:

- Vylepšit grafický vzhled překladatelského editoru, nejlépe reprezentovat formátovací tagy pomocí vlastního grafického prvku (jak již bylo zmíněno v kapitole 4.6.2).
- Nahradit *Dapper.NET Entity Frameworkem*, jak jsme již zmiňovali v 4.7.2.
- Přidat podporu pro *offline* práci překladatelů.
- Automatizovat nasazování nových verzí aplikačního serveru, aby se zjednodušil proces popsáný v podkapitole 6.1.
- Přidat podporu pro další překladatelské formáty.
- Vylepšit překladatelský editor, aby byl schopen vyhledávat a nahrazovat stoprocentní shody – překladatel nemusí stejný text překládat vícekrát.
- Umožnit lokalizaci klientských aplikací, nyní jsou texty zobrazované uživateli bohužel přímo v kódu, což je nepřehledné.

Seznam použité literatury

- [1] JONES, M. Dapper vs Entity Framework vs ADO.NET Performance Benchmarking. URL <https://www.exceptionnotfound.net/dapper-vs-entity-framework-vs-ado-net-performance-benchmarking/>.
- [2] LOCALISATION INDUSTRY STANDARDS ASSOCIATION (LISA) (2005). TMX 1.4b Specification. URL <http://www.ttt.org/oscarStandards/tmx/>.
- [3] MICROSOFT. RichTextBox Overview. URL [https://msdn.microsoft.com/en-us/library/ee681613\(VS.95\).aspx](https://msdn.microsoft.com/en-us/library/ee681613(VS.95).aspx).
- [4] MICROSOFT. Windows Communication Foundation. URL [https://msdn.microsoft.com/en-us/library/dd456779\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/dd456779(v=vs.110).aspx).
- [5] MICROSOFT (2011). .NET Remoting. URL [https://msdn.microsoft.com/en-us/library/72x4h507\(v=vs.100\).aspx](https://msdn.microsoft.com/en-us/library/72x4h507(v=vs.100).aspx).
- [6] MICROSOFT (2016). C# Language Specification. URL <https://github.com/ljw1004/csharpspec/blob/gh-pages/README.md>.
- [7] MICROSOFT (2017). .NET Framework. URL <https://www.microsoft.com/net>.
- [8] ORGANIZATION FOR THE ADVANCEMENT OF STRUCTURED INFORMATION STANDARDS (OASIS) (2008). XLIFF version 1.2. URL <https://docs.oasis-open.org/xliff/v1.2/os/xliff-core.html>.
- [9] RSA LABORATORIES (2012). PKCS #5 v2.1: Password-Based Cryptography Standard.
- [10] SDL STORE (2017). Sdl Store Website. URL <http://www.sdl.com/store/>.
- [11] STACKOVERFLOW.COM. Complex UI inside ListBoxItem. URL <https://stackoverflow.com/questions/15532639/complex-ui-inside-listboxitem>.
- [12] STACKOVERFLOW.COM. What is the best workaround for the WCF client using block issue? URL <https://stackoverflow.com/questions/573872/what-is-the-best-workaround-for-the-wcf-client-using-block-issue>.
- [13] TABOR, J. (2013). CAT tool use by translators: what are they using? URL <https://prozcomblog.com/2013/03/28/cat-tool-use-by-translators-what-are-they-using/>.

Platnost všech odkazů byla ověřena 17.05.2017.

Přílohy

A Implementace

Tato příloha obsahuje zdrojové kódy celého systému

- Visual Studio solution se všemi projekty a zdrojovými kódy se nachází ve složce `/source/Qubit`.
- SQL Skripty pro tvorbu databáze jsou ve složce `/dbscripts`:
 - `create_schema.sql` – skript pro tvorbu databáze na Microsoft SQL Serveru
 - `seed_data.sql` – skript pro vyplnění tabulky `Languages` a vložení prvního uživatele
 - `test_data.sql` – nepovinný skript pro vložení počátečních dat do databáze, vloží data, která můžeme nalézt i v příloze [C](#).

B Testovací nasazení

Tyto přílohy jsou potřeba v případě, že čtenář rozhodne pro testovací nasazení popsané v podkapitole [6.1.1](#):

- Předpřipravený certifikát pro použití v testovacím nasazení (viz níže) je v souboru `/install/cert.pfx`.
- Připravené instalátory klientských aplikací jsou v adresáři `/install`:
 - `ProjectManagerSetup.exe` – instalátor manažerské aplikace
 - `TranslatorSetup.exe` – instalátor klientské aplikace
- Zkompilované klientské aplikace a aplikační server jsou v adresáři `/compiled`, respektive jejich podadresářích.
 - `ProjectManagerClient` – zde je výstup projektu *ProjectManagerClient* (manažerská aplikace).
 - `TranslatorClient` – zde je výstup projektu *TranslatorClient* (překladačská aplikace).
 - `MiddleTier` – zde je výstup projektu *MiddleTier* (aplikační server).

C Ukázkové soubory

Ukázkové soubory se nachází ve složce `/example_data`, kde jsou roztríděné do těchto podsložek:

- `orig` – obsahuje originální dokumenty a z nich vytvořené SDLXLIFF dokumenty pomocí nástroje SDL Trados Studio
- `generated` – SDLXLIFF a TMX soubory vygenerované naší aplikací ze souborů z adresáře `orig`

D Podpůrný software

Podpůrné knihovny a aplikace, které by se mohli hodit pro nasazení software se nacházejí ve složce `/tools`..

- `SelfCert.exe` – utilita pro tvorbu certifikátů
- `babyftp.exe` – FTP server, který lze použít pro zveřejnění aktualizací.