Nástroj pro zpracování logovacích souborů

David Šupa

Bakalářská práce
2017

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky
akademický rok: 2016/2017

ZADÁNÍ BAKALÁŘSKÉ PRÁCE
(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**David Šupa**
Osobní číslo:	**A14145**
Studijní program:	**B3902 Inženýrská informatika**
Studijní obor:	**Informační a řídicí technologie**
Forma studia:	**prezenční**
Téma práce:	**Nástroj pro zpracování logovacích souborů**
Téma anglicky:	**A Software Tool for Processing Log Files**

Zásady pro vypracování:

1. **Analyzujte vstupní funkční a nefunkční požadavky aplikace pro zpracování logovacích souborů.**
2. **Navrhněte a popište vhodné metody nebo technologie pro efektivní práci s rozsáhlými logovacími soubory.**
3. **Vytvořte specifikaci uživatelského rozhraní a veškerých funkcí aplikace.**
4. **Implementujte aplikaci dle vytvořené specifikace.**
5. **Otestujte aplikaci na vhodných logovacích souborech.**

Rozsah bakalářské práce:

Rozsah příloh:

Forma zpracování bakalářské práce: **tištěná/elektronická**

Seznam odborné literatury:

1. **MCCONNELL, Steve. Dokonalý kód: umění programování a techniky tvorby software. Brno: Computer Press, 2005. ISBN 80-251-0849-X.**
2. **NAGEL Christian. Professional C# 5.0 and .NET 4.5.1. Indianapolis, 2014. ISBN 978-1-118-83294-3.**
3. **User Interfaces in C#: Windows Forms and Custom Controls. Berkeley, CA: Apress, 2002. ISBN 9781430208372.**
4. **CLEARY, Stephen. Concurrency in C cookbook: [asynchronous, parallel, and multithreaded programming]. ISBN 1449367569.**
5. **WATSON, Ben. Writing high-performance .NET code. S.l: Ben Watson, 2014. ISBN 9780990583431.**

Vedoucí bakalářské práce: **Ing. Tomáš Dulík, Ph.D.**
Ústav informatiky a umělé inteligence

Datum zadání bakalářské práce: **24. února 2017**

Termín odevzdání bakalářské práce: **24. května 2017**

Ve Zlíně dne 24. února 2017

doc. Mgr. Milan Adámek, Ph.D.
děkan

prof. Ing. Vladimír Vašek, CSc.
ředitel ústavu

Název bakalářské/diplomové práce:

Prohlašuji, že

- beru na vědomí, že odevzdáním diplomové/bakalářské práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb. o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších právních předpisů, bez ohledu na výsledek obhajoby;
- beru na vědomí, že diplomová/bakalářská práce bude uložena v elektronické podobě v univerzitním informačním systému dostupná k prezenčnímu nahlédnutí, že jeden výtisk diplomové/bakalářské práce bude uložen v příruční knihovně Fakulty aplikované informatiky Univerzity Tomáše Bati ve Zlíně a jeden výtisk bude uložen u vedoucího práce;
- byl/a jsem seznámen/a s tím, že na moji diplomovou/bakalářskou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- beru na vědomí, že podle § 60 odst. 1 autorského zákona má UTB ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- beru na vědomí, že podle § 60 odst. 2 a 3 autorského zákona mohu užít své dílo – diplomovou/bakalářskou práci nebo poskytnout licenci k jejímu využití jen připouští-li tak licenční smlouva uzavřená mezi mnou a Univerzitou Tomáše Bati ve Zlíně s tím, že vyrovnání případného přiměřeného příspěvku na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše) bude rovněž předmětem této licenční smlouvy;
- beru na vědomí, že pokud bylo k vypracování diplomové/bakalářské práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tedy pouze k nekomerčnímu využití), nelze výsledky diplomové/bakalářské práce využít ke komerčním účelům;
- beru na vědomí, že pokud je výstupem diplomové/bakalářské práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá. Neodevzdání této součásti může být důvodem k neobhájení práce.

Prohlašuji,

- že jsem na diplomové/bakalářské práci pracoval samostatně a použitou literaturu jsem citoval. V případě publikace výsledků budu uveden jako spoluautor.
- že odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou totožné.

Ve Zlíně, dne 23.5.2017

.....
podpis diplomanta

ABSTRAKT

Cílem bakalářské práce je vytvoření softwarového nástroje za účelem zjednodušení práce s velkými logovacími soubory a zlepšení přehlednosti těchto souborů.

Úkolem teoretické části této práce je seznámení čtenáře s technologiemi a pojmy, které byly v rámci této práce použity.

Praktická část se zaměřuje na popis výsledné aplikace a ukázku co vše lze v rámci aplikace udělat.

Klíčová slova: .NET Framework, C#, Logování, WireFrame, Windows Forms.

ABSTRACT

The main point of this bachelor thesis is creation of software tool which allows the user to work with large log files with ease.

The goal of theoretical part of this work is to get the reader to familiarize with topic and technologies used in creation of the software tool.

Practical part shows how the software tool works and everything what can be done with it.

Keywords: NET Framework, C#, Logging, WireFrame, Windows Forms.

Děkuji Ing. Tomáši Dulíkovi, Ph.D., vedoucímu bakalářské práce, za odborné rady, připomínky a cenné nápady.

Dále bych chtěl poděkovat firmě Edhouse s.r.o., kde jsem strávil letní stáž a v rámci stáže, pracoval na tomto softwarovém nástroji. Především bych pak chtěl poděkovat vedoucímu mojí stáže Ing. Petru Müllerovi, za veškerou ochotu, užitečné rady a hlavně za trpělivost. Nakonec patří velký dík mé rodině, bez které bych nemohl studovat a jejíž podpora pro mne mnoho znamená.

Prohlašuji, že odevzdaná verze bakalářské práce a verze elektronická nahraná do IS/STAG jsou totožné.

OBSAH

ÚVOD

Logování, respektive pravidelné zaznamenávání konkrétních údajů, provází lidstvo již od pravěku, kdy se například zaznamenávaly měsíční fáze, čímž se zjišťovalo, kdy nastane další zima. Logování v počítačovém kontextu znamená automatické produkování záznamů o proběhnutí událostí v rámci jistého systému. Prakticky všechny softwarové aplikace a systémy produkují logovací soubory.

Tyto soubory mohou nabývat nezměrných velikostí, někdy až v rámci GB, což představuje několik milionů řádků informací. Procházení takovéhoto souboru bez pomoci chytrého software se může stát noční můrou. Většina aplikací neposkytuje uživateli možnost nějak efektivně zpracovávat tyto soubory a procházet jednotlivé záznamy. Z toho důvodu se většinou používají softwarové aplikace třetích stran, které takovéto procházení usnadňují.

V rámci této práce vznikl softwarový nástroj umožňující uživateli lépe pracovat s těmito soubory. Tento nástroj vznikl podle požadavků firmy Edhouse s.r.o. a byl doplněn několika dalšími vylepšeními vyžádanými vedoucím bakalářské práce.

Bakalářská práce „Nástroj pro zpracování logovacích souborů“ se stává ze dvou hlavních částí, z teoretické a praktické.

Teoretická část začíná výčtem požadavků, které by měla aplikace splňovat. Tyto požadavky tvoří základní kámen aplikace. Následuje uvedení do pojmu logování a popis typů logovacích souborů tak, jak jsou rozlišovány v informatice. Dále je čtenář seznámen s vývojovým prostředím .NET, konkrétně s funkcemi, které byly použity při tvorbě této aplikace a tvoří její převážnou část. Není opomenut ani programovací jazyk C#, ve kterém byla celá aplikace napsána. Předposlední kapitola teoretické části se zabývá popisem několika možných metod, které lze použít při práci s rozsáhlými logovacími soubory. A v poslední kapitole se nachází specifikace toho, jak by mělo vypadat uživatelské rozhraní aplikace pomocí tzv. *wireframe*.

První kapitolou praktické části je implementace uživatelského rozhraní podle specifikace, která se nachází v teoretické části. V této kapitole jsou ukázány a popsány všechny hlavní formuláře aplikace. U každého formuláře se nachází podrobný popis komponent, které jsou jeho součástí. Následuje kapitola s popisem hlavních tříd, které tvoří jádro aplikace, a také jejich metod. Praktická část je zakončena popisem ovládání aplikace při práci s vhodnými logovacími soubory.

I. TEORETICKÁ ČÁST

1 ANALÝZA VSTUPNÍCH POŽADAVKŮ APLIKACE

Aplikace, která je součástí této práce, byla zpracována podle následujících požadavků:

- Práce s více logovacími soubory najednou v přehledné tabulkové formě představuje hlavní požadavek.
- Možnost čtení neomezeně velkých logovacích souborů.
- Ztvárnění načtených dat formou grafů.
- Další požadavek na aplikace sestává z možnosti načtení záznamů událostí ve Windows, a to konkrétně systémové události a události aplikací.
- Aplikace by měla mít možnost zaznamenávat přístup do právě čteného *logovacího* souboru a v případě zápisu nové události do souboru aktualizovat právě zobrazený obsah.
- Možnost úpravy klíčových slov, podle kterých se vyhledává typ události v *logovacím* souboru bez nutnosti zásahu do kódu aplikace.
- Schopnost vytvářet profily, které obsahují adresy souborů, a tím umožnit jejich opětovné načtení rychle a snadno. Tyto profily ukládat do souboru s příponou *.xml*.
- Podbarvování řádků podle typů událostí a možnost definovat vlastní barvu pro danou událost u profilů úpravou *.xml* souboru, který obsahuje profily.
- Aplikace by také měla poskytnout možnost spojení několika právě otevřených souborů do jedné záložky.

2 LOGOVÁNÍ

Vytváření logů z aplikací slouží k zaznamenávání informací o průběhu programu, informací sloužících k ladění aplikace a také informací o výskytu chyb v aplikaci. Běžně také dochází k logování na serveru. Zde se sledují požadavky přicházející od uživatelů na server a zapisují se do souboru. *Logovací* soubory významně pomáhají při hledání příčin problémů, z toho důvodu dochází k jejich zaznamenávání.

2.1 Logovací soubory

Pod pojmem *logovací* soubor rozumíme v informatice soubor, který slouží k zaznamenávání událostí. Těmito událostmi může být například zaznamenávání událostí, které se projevily v operačním systému či v jiných softwarech atd. Běžně se *logovací* soubory dělí na *logy událostí* a *transakční logy*.

2.1.1 Event Logs

Event Logs neboli *logy událostí*, zaznamenávají průběh všech událostí v systému tak, aby mohly podat záznam o posloupnosti událostí, ze kterého lze pochopit aktuální chování systému a diagnostikovat problémy. Tvoří důležitou část při chápání chování složitých systémů. Převážně pak u aplikací, kde dochází k velmi malému počtu styků s uživatelem, například serverové aplikace.

Také kombinací záznamů z více zdrojů se může jevit jako velice užitečné. Tato kombinace spolu s využitím statické analýzy může ukázat spojitost mezi událostmi, které jinak vypadají zcela nezávisle.

Operační systém Windows klasifikuje události podle následujících pěti typů:

- *Information event* – popisuje úspěšné dokončení úlohy (např. instalace aplikace).
- *Warning event* – upozorňuje správce o potencionálním problému (např. nedostatek paměti).
- *Error message* – představuje závažný problém, který může vyústit až ke ztrátě funkčnosti.
- *Success audit* – tato událost indikuje dokončení události kontroly bezpečnosti (např. úspěšné přihlášení uživatele).
- *Failure audit* – jedná se o to samé jako *success audit*, nicméně v tomto případě událost popisuje neúspěšnou kontrolu (např. když uživatel zadá špatné heslo).

Logovací soubory zaměřeny na záznamy událostí mají téměř totožné složení obsahu. Tento obsah se může měnit v závislosti na aplikaci, pod kterou daný soubor spadá. Obsah většiny souborů vypadá následovně:

- *Časová známka* – jako první záznam bývá většinou časový údaj spolu s datem, kdy tato událost proběhla.
- *Typ události* – následuje typ události, který je většinou zapsán zkrácenou formou. Např. *info, debug, warn, error* a další.
- *Obsah události* – zde se nachází obsah události. Tedy zpráva popisující, co se stalo.

Tyto tři údaje jsou obsaženy ve všech logovacích souborech zaznamenávajících průběh událostí.

2.1.2 Transaction logs

Většina databázových systémů si uchovává takzvané *transakční logy*. Tyto *logy* většinou nejsou určeny k zaznamenávání posloupnosti událostí k jejich pozdější analýze a také nejsou tvořeny tak, aby byly pro člověka čitelné. Tyto záznamy slouží k uchovávání seznamu změn, které byly provedeny uloženým datům. To vše z důvodu případného pádu či jiného datového výpadku. *Transakční záznamy* poté slouží k obnovení dat do jejich původního stavu před jejich poškozením.

3 PROSTŘEDÍ .NET

Technologie *.NET* je jednou z nejvíce používaných a nejrychleji se rozvíjejících *platform*. Je určena pro vývoj a běh počítačových aplikací, se kterými se setkáváme neustále. Například systém Microsoft Windows, který tuto technologii uvedl v roce 2000 [1], ji využívá nejvíce v rámci osobních počítačů. Také v oblasti mobilních technologií si *.NET* získal své místo.

Jádrem celé platformy je právě sada programovacích rozhraní *.NET Framework*. Právě rozhraní *.NET* poskytuje spoustu služeb jako například správa paměti, typová a paměťová bezpečnost, vytváření sítí a zavádění aplikací. Poskytuje také datové struktury, které jsou jednoduché na použití a rozhraní *API*, které se abstraktně chovají jako operační systém Windows na nižší úrovni. Rozhraní také podporuje použití spousty programovacích jazyků jako např. *C#, F# a Visual Basic*.

Rozhraní *.NET* slouží k vývoji aplikací pro *Windows*, *Windows Phone*, *Windows Server* a *Microsoft Azure*. Skládá se z *obecného jazykového runtime* (*CLR*), který řídí provádění kódu napsaného v libovolném podporovaném programovacím jazyce. Nedílnou součástí rozhraní *.Net* jsou také knihovny tříd, které zahrnují širokou škálu funkčností a podpory mnoha průmyslových standardů. [2]

3.1 Rozdíl Windows Forms a WPF

Windows Presentation Foundation neboli *WPF* představuje nejnovější přístup ke grafickému uživatelskému rozhraní (*GUI*) používanému s platformou *.NET*. Bezhlavé použití nejnovějšího *WPF* je stejně hloupé, jako použití *Windows Forms* tam, kde se svými schopnostmi plně využije potenciál *WPF*.

3.1.1 Výhody WPF

3.1.1.1 Data Binding

Jednou z největších výhod je *data binding*. Jedná se o mechanismus určený pro synchronizaci dat a uživatelského rozhraní. Typickým příkladem je například aktualizace komponenty při změně obsahu listu. Zjednodušeně řečeno jde hlavně o synchronizaci mezi datovým zdrojem a cílem dat. Ve *WPF* lze tedy provést *data binding* pouhým zápisem *{Binding}*. [3]

3.1.1.2 Přizpůsobení UI a grafiky

Ve *WPF* se každá komponenta dá předělat podle vlastních představ. Lze zcela předělat vzhled komponenty a přepsat její chování. Jednoduše pouhou editací *XAML* šablony dané komponenty. Každý element ve *WPF* aplikaci je vektorovým objektem. Tedy přibližování a změny ve velikosti mohou být provázeny bez ztráty na kvalitě po vizuální stránce.

3.1.1.3 Čisté oddělení UI a logiky

Oddělení logiky programu od uživatelského rozhraní je jedním ze základních kamenů při tvorbě aplikace. Zajišťuje se tím nezávislost kódu na uživatelském rozhraní. Tedy můžeme provést změny v kódy bez nutnosti provedení změn v uživatelském rozhraní. Toto oddělení existuje i ve *Windows Forms*, nicméně zde je dotaženo k dokonalosti. Při tvorbě aplikace ve *Windows Forms* se generuje přeci jen část kódu v předem zvoleném jazyce. Znalost jazyka, ve kterém se daná aplikace tvoří nebo tvořila, je tedy nutností při úpravě kódu. U *WPF* je tomu naopak. Celý design uživatelského rozhraní může být proveden někým, kdo nemá nejmenší znalosti o daném programovacím jazyce, kterým je napsána daná aplikace. [4]

3.1.2 Výhody Windows Forms

3.1.2.1 Zaběhlá technologie

Jelikož se jedná o starší technologii, na které již bylo vyvinuto mnoho aplikací, můžeme s klidem říci, že je tato technologie bezpečná. Tedy veškeré její chyby, slabé stránky, ale i silné stránky jsou známy a můžeme se tedy rozhodnout, co využijeme pro naši aplikaci. Na internetu můžeme najít nepřeberné množství dokumentace, která se zde za ta léta nahromadila. Také můžeme najít veliké množství příkladů a řešených problémů.

3.1.2.2 Software třetích stran

Jelikož se jedná o zaběhlou technologii, nachází se na internetu nepřeberné množství kontrol, které byly vyvinuty třetími stranami a poskytnuty buď volně k šíření (*nugety*) či za jistý poplatek. Vyvstává tedy otázka: proč vytvářet něco, co již existuje, když se jedná o věc volně dostupnou? [5]

3.2 Windows Forms

Vytvářet aplikace, jenž používají textový výstup, respektive většinou vypisují údaje na konzoli počítače, se hodí spíše pro technické aplikace, jejichž výstup není obsáhlý. Nicméně dnešní komplikovanější aplikace, od nichž se očekává interaktivní reakce na uživatelské požadavky, či zobrazování grafů a podobně, lze jen ztěží provádět pomocí konzolových výpisů. V těchto případech se více hodí grafický výstup aplikace.

Prostředí *.NET* poskytuje pro vytváření grafického uživatelského rozhraní v rámci aplikací typu Windows jmenný prostor *Windows.Forms*. Tento jmenný prostor obsahuje rozsáhlé množství prvků, pomocí nichž je možno vytvořit či zpřehlednit a zjednodušit pro uživatele aplikaci.

Aplikace *Windows Forms* byly vždy používány při vývoji desktopových aplikací, od kterých se očekává, že veškerý výpočetní proces bude prováděn na straně klienta. To zahrnuje vše od vykreslení grafických aplikací až po systémy pro vkládání dat atd.

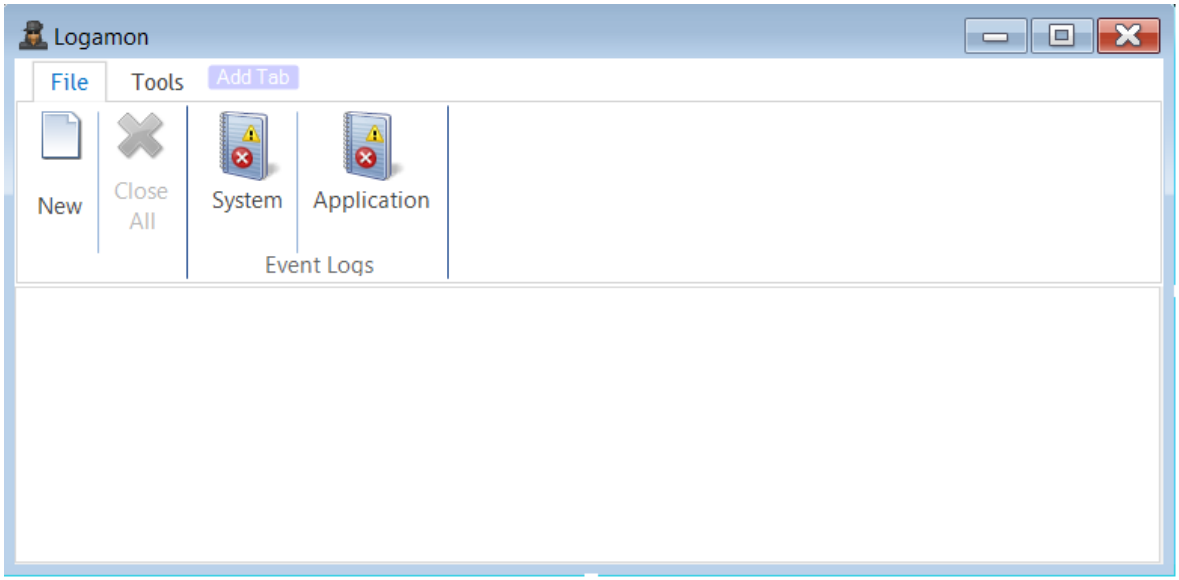
3.2.1 Co je to Form (formulář)

Formulář nabývá různých forem, vždy se ale jedná o prvek, který zabírá kus obrazovky. Většinou má tento formulář obdélníkový tvar. Formulář se používá k zobrazení informací a k příjmu vstupních údajů od uživatele. Může se jednat o standardní Windows okna, dialogová okna anebo zobrazování ploch pro grafické rutiny.

Tak jako všechny objekty v prostředí *.NET* jsou i formuláře instancemi tříd. Formulář, který je vytvořen pomocí *Windows Forms Designer*, se stává třídou. Při zobrazení instance tohoto formuláře za běhu programu se stává tato třída šablonou, která je použita pro vytvoření instance formuláře.

Prostředí *.NET* také umožnuje dědičnost z již existujících formulářů pro přidání funkcí či změnu chování již existujících funkcí. Při přidání formuláře k objektu lze rozhodnout, zda má daný formulář dědit od třídy *Windows.Forms* poskytovanou prostředím anebo může být zvolena dědičnost z existujícího formuláře, který byl už předtím vytvořen.

Formulář vytvořen v rámci projektu *Windows Forms* představuje hlavní způsob komunikace s uživatelem. Kombinováním různých sad ovládacích prvků a vhodně napsaným kódem, lze získat informace od uživatele a reagovat na ně. Také lze pracovat s existujícími daty či se dotazovat a zpětně zapisovat do systémových souborů a registrů. [6]



Obr. 1. Ukázka formuláře

3.2.2 Komponenty

Windows Forms poskytují mnoho komponent s tím, že každá z nich má své vlastnosti a zaměření. Následuje výpis několika komponent pro získání obecné představy.

3.2.2.1 ListView

Ovládací prvek *ListView Windows Forms* slouží k zobrazování seznamu položek. Toto zobrazení může být provedeno jedním ze čtyř možných způsobů. Položky mohou být zobrazeny formou dlaždic s velkými nebo malými ikonami reprezentujícími jednotlivé položky seznamu. Dále lze zobrazit položky formou vertikálního seznamu. Pro nejpodrobnější zobrazení lze využít detailního návrhu, kde máme možnost vidět nejen jednotlivé položky, ale také jejich podpoložky. Každá položka je zobrazena formou tabulky, kde hlavní položky jsou umístěny vertikálně pod sebou a podpoložky pro jednotlivé hlavní položky jsou umístěny vedle ve sloupcích. Detailní náhled je ideálním řešením při zobrazování například databáze uživatelů spolu s detaily ke každému uživateli.

3.2.2.2 TabControl

Účelem formuláře *TabControl* je zobrazení záložek, což si lze představit jako hromadu složek poskládaných v kartotéce. Aplikace využívající *TabControl* může vypadat jako například libovolný prohlížeč, kde má uživatel načteno zároveň více stránek a podle potřeby se mezi nimi přepíná. Záložky mohou obsahovat obrázky a další ovládací prvky. *TabControl*

lze použít k vytvoření víceúčelového dialogového okna, které se běžně nachází v operačním systému Windows.

3.2.2.3 UserControl

User control neboli uživatelská komponenta umožňuje vytvoření vlastní komponenty s vlastnostmi, které si definujeme sami. Základní představa použití této komponenty vychází ze stejného principu jako použití klasického tlačítka či *textboxu*. Tato komponenta umožňuje programátorovi návrh vlastní komponenty, která bude mít programátorem definované funkce. Typickým důvodem použití této komponenty je návrh komponenty, která bude následně využita vícekrát. Při jejím vícenásobném použití se pouze vytvoří nová instance této komponenty. Nejenže se tím ušetří spousta práce, ale také to umožňuje použití této komponenty ve chvílích, kdy není jisté, kolikrát ji bude potřeba použít.

Obecně se vývojáři zaměřují na komponenty typu *custom control* kvůli jednomu z těchto tří důvodů: [7]

- Za účelem vytvoření ovládacích prvků, které odstraňují nevýznamné detaily a jsou přizpůsobeny na míru konkrétnímu typu dat.
- Kvůli vytvoření ovládacích prvků, které poskytují zcela nové funkce nebo kombinují již existující funkce ovládacích prvků.
- Pro možnost vytvoření vlastních ovládacích prvků s originálním vzhledem či takovým vzhledem, který kopíruje již existující populární vzhled profesionálních aplikací.

Dalšími typy takzvaných *custom controls* jsou: [8]

- *User controls* – jedná se o nejjednodušší typ ovládacího prvku. Dědí ze třídy System.Windows.Forms.UserControl.
- *Inherited controls* – ty představují zdědění ovládací prvků, které jsou obecně výkonnější a flexibilnější. V základu se nachází zděděný existující ovládací prvek, jehož metody a vlastnosti jsou upraveny podle potřeb uživatele.
- *Owner-drawn controls* – vzhled těchto ovládacích prvků je vytvořen nově zcela od základu.

3.2.2.4 Chart Control

Jedná se o ovládací prvek typu graf. Umožňuje vytvářet jednoduché a vizuálně přesvědčivé grafy pro komplexní statické nebo finanční analýzy. Tato komponenta slouží k vytváření

grafů se všemi nezbytnostmi, jako jsou legendy, osy, názvy jednotlivých sérií a další prvky grafu. Každý prvek grafu odpovídá objektu.

Grafy lze vybrat z veliké škály dostupných stylů:

- *Column* – jedná se o klasický sloupcový graf. Osa x slouží k vypsání jednotlivých prvků grafu. Osa y pak slouží k vyobrazení číselného množství daného prvku definováním výšky sloupce.
- *Pie* – koláčový graf slouží k zobrazení, jak velkou část zabírají jednotlivé údaje z celkového počtu.
- *Point* – účel bodového grafu spočívá v zobrazení konkrétního bodu na rozhraní os x a y.
- *Line* – představuje typický graf, který zobrazuje hodnotu nějaké veličiny v závislosti na druhé veličině. Typickým příkladem může být zobrazení teploty v závislosti na čase.
- *Spline* – tento graf je stejným jako graf Line s tím rozdílem, že přechod mezi jednotlivými body není přímá čára. Graf se podle pozice následujícího bodu snaží čáru zakřivit tak, aby se vzhled více blížil ke skutečnému průběhu.

Komponenta nabízí celkem 36 různých stylů grafů, z nichž každý se hodí na zobrazení něčeho jiného.

3.2.2.5 BackgroundWorker

Komponenta *BackgroundWorker* umožňuje spouštět asynchronní operace. Existuje spousta běžně prováděných operací, jejichž výpočet může trvat dlouho. Pokud dochází ke spouštění těchto operací na hlavním vláknu, na kterém běží celá aplikace, pak může docházet k nepříjemnému zamrznutí této aplikace na dobu provádění výpočtu. Tomuto se dá předejít tak, že se časově náročná operace provede asynchronně na vedlejším vláknu. Po dokončení výpočtu se na hlavním vláknu provedou pouze změny typu aktualizace vizuální stránky komponenty.

U komponenty *backgroundWorker* dochází pouze k obsloužení tří událostí. První je událost *DoWork*. Do této události se zapisuje časově náročný kód, který je nutné provést asynchronně. V této části se nabízí možnost hlásit, kolik práce již bylo zpracováno.

Jako další událost následuje *ProgressChanged*. Tato událost se spouští v případě, že z události *DoWork* hlásíme množství zpracované práce.

Poslední událostí je *RunWorkerCompleted*. Kód v této události proběhne poté, co se dokončí práce v události *DoWork*. Do této události se umísťuje například aktualizace uživatelského rozhraní.

3.2.2.6 Ribbon

Komponenta *Ribbon* je moderní verzí klasického výběrového menu aplikace. Jejím hlavním účelem je poskytnout uživateli co nejrychlejší přístup k funkcím aplikace v co nejméně krocích. *Ribbon* tvoří příkazovou lištu umístěnou okolo vrchního okraje formuláře. Silnou stránkou této komponenty je možnost navrhnutí designu za účelem maximálního zjednodušení práce s aplikací. Hlavní síla této komponenty spočívá v možnosti vytvoření záložek, kde každá záložka obsahuje skupiny komponent, rozdělené podle jejich účelu. Záložkám lze přiřadit jméno tak, aby se definoval co nejpřesněji jejich obsah. Jednotlivé komponenty umístěny ve stejné skupině by měli mít společný význam, účel. To z toho důvodu, aby uživatel mohl co nejrychleji najít požadovanou funkci.

3.2.3 ObjectListView

ObjectListView je komponenta třetí strany, kterou tato aplikace používá místo *listView*. Autorem této komponenty je Phillip Piper. Hlavní cílem při vývoji této komponenty bylo poskytnutí *listView*, který by byl jak pro uživatele, tak i pro programátora přívětivým. Jedna z nejvíce užitečných funkcí sestává z možnosti vytvoření *listView* z kolekce objektů. Po definování sloupců, které budou zobrazeny, *ObjectListView* sám dokáže vyplnit tyto sloupce z kolekce objektů a každý objekt umístit na nový řádek. Toto vše se dá provést pouze zavoláním jediné metody *SetObjects*.

Další vylepšení oproti klasickému *ListView* sestává z možnosti rozčlenění řádků do skupiny. Pro tyto skupiny lze definovat vlastnosti jako například vzhled řádku či název dané skupiny. Skupiny mohou mít také přiřazeny své vlastní ikony, podnázvy a další.

ObjectListView nativně podporuje také automatické řazení. Pokud uživatel klikne na hlavičku sloupce, dojde k automatickému seřazení. Toto řazení probíhá podle datového typu, který daný sloupec obsahuje. Například tedy sloupec zobrazující čas a datum umožňuje seřazení od nejmladšího záznamu po nejstarší a naopak.

ObjectListView je dostupný v několika verzích: [9]

- *ObjectListView* – čistá verze, která pracuje s kolekcí objektů a s nimi vyplňuje *ListView*.
- *DataListView* – představuje verzi, jejíž hlavní výhodou je možnost propojení zdroje dat a komponenty na takové úrovni, že v případě změny ve zdroji dat se komponenta sama aktualizuje.
- *FastObjectListView* – jedná se o rychlou verzi *ObjectListView*, dokáže např. vytvořit seznam 10 000 objektů za méně než 0.1 sekundy.
- *VirtualObjectListView* – nepracuje s kolekcí objektů, ale s modelovými objekty. Jednodušeji řečeno pracuje s tzv. *RowGetter* delegátem, který je volán v případě, že list potřebuje zobrazit konkrétní modelový objekt.
- *TreeListView* – kombinuje stromovou strukturu *TreeView* se zobrazování více sloupců.

3.3 Jmenný prostor IO

Tento jmenný prostor poskytuje třídy pro práci se vstupy a výstupy, což zahrnuje schopnost číst a zapisovat do datových proudů synchronně či asynchronně. Také to umožňuje komprimovat data v datovém proudu, vytvářet a používat izolované úložiště a mapovat soubory do logického adresního prostoru aplikace. Nebo také ukládat více datových objektů do jediného místa, komunikovat pomocí anonymních anebo pojmenovaných kanálů a zpracovávat tok dat do nebo ze sériových portů. [10]

3.3.1 Třída File

Jedná se o třídu obsaženou ve jmenném prostoru *System.IO*. Tato třída poskytuje statické metody pro vytváření, upravování, mazání, kopírování a otevírání souboru. Převážně pak slouží k vytvoření *FileStream* objektů. Tyto objekty poskytují *stream* dat pro soubor, čímž podporují jak synchronní, tak asynchronní operace typu čtení a zápis. *Stream* je abstraktní třídou, která poskytuje náhled na data jako posloupnost bajtů.

3.3.2 Třída StreamReader

Tato třída implementuje abstraktní třídu *TextReader*, která dokáže číst znaky z posloupnosti bajtů použitím určitého kódování. *StreamReader* tedy umožňuje pracovat pouze s pár znaky

bez nutnosti načíst celý soubor do paměti, což se stává zásadním aspektem při práci s opravdu velikými soubory.

3.4 Jmenný prostor Collections.Generic

Obsahuje rozhraní a třídy definující generické kolekce, které umožňují uživateli vytvářet silně typové kolekce poskytující větší typovou bezpečnost a výkonnost oproti obecným kolekcím.

3.4.1 Generické kolekce

Při vytváření kolekcí si programátor volí podle potřeby mezi obecnými a generickými kolekcemi. Obecná kolekce se od generické liší tím, že nemá určený datový typ. Tento datový typ se musí definovat u každého prvku, který se do dané kolekce vkládá (obr. 2).

```
ArrayList list = new ArrayList();
list.Add("položka");

string polozka = (string)list[0];
```

Obr. 2. Ukázka vytvoření obecné kolekce [11]

Generické kolekce odstraňují problém s definováním datového typu každého prvku při jeho vkládání. Jedná se o kolekce, u kterých se definuje jejich datový typ již při vytvoření (Obr. 3). Následné zpětné přetypování není již možné.

```
List<string> list = new List<string>();
list.Add("položka");

string polozka = list[0];
```

Obr. 3. Ukázka vytvoření generické kolekce [11]

3.5 XML Serialization

Serializace představuje konverzi objektu na proud bytů a poté jeho uložení někde v paměti, databázi či souboru. *Deserializace* je přímým opakem, kdy se vezme například soubor a převede se do určitého objektu. Hlavní účelem *XML serializace* v prostředí *.NET* je umožnění převodu *XML* dokumentů a proudu dat do objektů za běhu programu a naopak. Tento proces umožňuje převádět *XML* dokumenty do formy, v níž jsou jednodušeji zpracovatelné běžnými programovacími jazyky. Na druhou stranu pomocí *deserializace* se může například databáze uživatelů, uchovávat či převádět mnohem snadněji a bezpečněji.

XML serializace v prostředí *.NET* podporuje například *serializaci* objektů podle specifikovaného schématu *XSD*. Během *serializace* jsou pouze veřejné vlastnosti objektů *serializovány.* [12]

3.5.1 XML Schéma (XSD)

Jedná se o doporučení od *W3C*, jak správně popisovat položky v *XML* dokumentu. Tento popis může být použit pro ověření, zda každá položka obsažena v dokumentu souhlasí s popisem objektu, k němuž má být přiřazena. Schéma obecně představuje abstraktní reprezentaci vlastností a vztahů jednoho objektu ke druhému. *XSD* slouží pro popis vztahů mezi atributy a prvky objektu v rámci *XML* dokumentu. [13]

3.6 Garbage Collection

Garbage collector je odpovědí *.NET* na správu paměti a také na otázku, jak přistupovat k obnovení paměti v případě zažádání běžící aplikací. Před uvedením *garbage collectoru* se k uvolňování paměti přistupovalo jednou ze dvou následujících technik:

- O správu paměti se bylo nutno starat manuálně v kódu aplikace.
- Uchovával se počtu referencí na jednotlivé objekty.

Techniku, při které se stará o správu paměti samotná aplikace, používají nízkoúrovňové, vysoce výkonné jazyky jako *C++*. Jedná se o efektivní řešení. Hlavní výhoda spočívá v tom, že zdroje nejsou zadržovány na delší dobu, než je nezbytně nutné. Na druhou stranu, velkou nevýhodou jsou časté chyby, kterých se při psaní takovéto aplikace programátoři často dopouštějí a následně pak dochází k únikům paměti.

Běh v prostředí *.NET* závisí na *garbage collectoru*. Účel tohoto programu leží v úklidu paměti. Myšlenka spočívá v tom, že všechna dynamicky vyžádaná paměť se nachází na haldě. V případě pokud *.NET* detekuje, téměř zaplněnou haldu spravovanou procesem, dojde k zavolání *garbage collectoru*. *Garbage collector* projde všechny proměnné, které jsou použity v kódu a zkontroluje jejich reference, zda některá z nich neodkazuje na právě se zaplňující haldu. Ta část haldy bez referencí se vymaže.

Garbage collector využívá ke svému fungování následujícího principu. Tento princip říká, že v rámci *.NET* jakožto typově bezpečného jazyka se nelze odkazovat na existující objekt jinak než zkopírováním existujícího odkazu. Pokud tedy existuje nějaký odkaz na objekt, lze z dostupných informací zjistit jeho datový typ. Tento mechanismus správy paměti nelze využít například v jazyku *C++,* protože tento jazyk dovoluje měnit ukazatele volně mezi typy.

Garbage collector je nedeterministický. Není tedy možné určit, kdy bude *garbage collector* zavolán a kdy proběhne uvolnění paměti. I když tento aspekt lze obejít a přímo v kódu lze vynutit jeho zavolání. [14]

4 JAZYK C#

C# je v současnosti společně s *Javou* nejmodernějším a nejpoužívanějším jazykem. Vyvinula jej firma Microsoft spolu s vývojovým prostředím *.NET*. Jedná se o typově bezpečný a objektově orientovaný jazyk. V rámci objektového programování jazyk jako takový podléhá speciálnímu paradigmatu založenému na konceptu objektů. Tyto objekty mohou obsahovat data ve formě polí, často označovaných jako atributy, a kód ve formě procedur označovaných jako metody. Existuje spousta objektově orientovaných jazyků a každý může k tomuto rozdělení přistupovat svým vlastním způsobem. *C#* využívá tzv. tříd, což si lze představit tak, že objekty jsou vlastně instancemi dané třídy, tedy mohou využívat metody, které tato třída implementuje.

Syntaxe jazyka *C#* je velice podobná jazykům *C, C++* a *Java*. Zjednodušuje mnoho složitostí *C++* a poskytuje výkonné funkce jako jsou *typy nulových hodnot, výčty, delegáty, lambda výrazy a přímý přístup k paměti*, které se v jazyce *Java* nenachází.

Překladače jazyka *C#* jsou *case sensitive*, což znamená, že rozlišují velká a malá písmena. V rámci jazyka bylo také zavedeno několik konvencí. Jména balíků, tříd, rozhraní a většina dalších položek začínají velkými písmeny. Malými písmeny začínají privátní a chráněné (*protected*) atributy, lokální proměnné a parametry. Podobně jako *Java* obsahuje pouze jednoduchou dědičnost s možností násobné implementace rozhraní. O uvolňování zdrojů aplikace se stará *garbage collector*. Správa paměti je tedy automatická. [15]

4.1 funkce

Níže jsou uvedeny některé funkce, které přináší jazyk *C#* a které byly použity při tvorbě této práce.

4.1.1 Typové odvození

Jedná se o funkci, díky které odpadá nutnost definovat datový typ při deklaraci proměnné. Toho lze docílit jednoduše použitím klíčového slova *var*. Například

var číslo = 0;

Proměnná *číslo* bude definována jako *integer* i přesto, že nikde v kódu tak deklarována není. Toto je možné díky překladači. Sám překladač totiž vyhodnotí, jakého datového typu by měla proměnná být na základě hodnoty, která je jí přiřazena.

4.1.2 Výčty

Výčet je uživatelem definovaný *integerový* typ. Při deklaraci výčtu se specifikuje rozsah takzvaných dovolených hodnot, kterých může daný výčet nabývat. Těmto hodnotám lze také přiřadit nějaký název, který se lépe pamatuje než pouhé číslo. Pokud se následně někde v kódu tomuto *výčtu* přiřadí jiná hodnota než povolená, daný řádek se označí jako chybný. *Výčty* v *C#* jsou v jádru vytvářeny jako struktury odvozené od základní třídy *System.Enum*.

4.1.3 Delegáti

Jedná se o možnost vytvoření objektů reprezentujících reference na metody. *Delegáti* mohou odkazovat jak na statické, tak instanční metody. *Delegáti* v *C#* berou jeden parametr do svého konstruktoru. Parametr vždy představuje metodu, na kterou daný *delegát* odkazuje. *Delegáti* musí mít stejný počet vstupních i návratových hodnot jako metody, na které odkazují. Každý *delegát* má možnost volat více než jednu metodu. Takový to *delegát* se nazývá *multicast delegate*. Pokaždé, když dochází k volání takovéhoto *delegáta*, jsou jednotlivé metody, na které odkazuje, volány popořadě. Aby bylo možné získat výsledky z obou metod, na které *delegát* odkazuje, musí být *delegát* typu *void*, jinak by došlo k navrácení pouze hodnoty z poslední volané metody. [16]

4.1.4 Lambda výrazy

Jsou to anonymní funkce, metody bez deklarací, jména bez deklarace návratového typu a bez modifikátoru přístupu. Umožňují napsat zkrácenou metodu, která bude použita na místě jejího vytvoření. Jedná se o efektivní způsob v případě, že je metoda použita pouze jednou a její definice je krátká.

Lambda výrazy by měli být krátké. Složitá definice činí následný kód špatně čitelným.

Nyní následuje krátký příklad použití *lambda výrazu*:

- *x => x % 2 == 1*
- *x je vstupním parametrem*
- *x % 2 == 1 je výrazem*

Tento příklad lze interpretovat následovně:

Vstupní parametr pojmenovaný *x* vstupuje do anonymní funkce, která navrací *true,* pokud je vstupním parametrem liché číslo. [17]

4.2 Asynchronní programování

V případě synchronního programování je aplikace jako celek napsána pro jedno vlákno. Tedy veškeré výpočetní operace a zároveň obsluha uživatelského rozhraní jsou jako operace prováděný jedním vláknem. Hlavní nevýhoda takovéto aplikace se projevuje ve chvíli, kdy se dostává kód aplikace do výpočetně náročně části. V tuto chvíli uživatelské rozhraní aplikace zamrzá, jelikož je hlavní vlákno, na kterém celý program běží, zaměstnáno. Uživatel pak může reagovat na toto zamrznutí tak, že se pokusí vrátit změny zpět předtím, než aplikace přestala odpovídat. Aplikace by měla okamžitě reagovat na uživatelovi požadavky. V minulosti byly všechny aplikace programovány s použitím jednoho vlákna a uživatel byl v této chvíli bezmocným, jelikož aplikace přestala odpovídat. Jako řešení se objevilo asynchronní programování.

Princip asynchronního programování spočívá v tom, že o časově náročný kód se postará vedlejší vlákno. Díky tomu je hlavní vlákno schopno ovládat uživatelské rozhraní a reagovat tak na uživatelovi příkazy, zatím co se na pozadí zpracovává časově náročná část. Po dokončení časově náročného výpočtu oznámí vedlejší vlákno hlavnímu vláknu, že skončilo a následně se ukončí. Hlavní vlákno poté pracuje s výsledky, které získalo od vedlejšího vlákna.

4.2.1 Zastavení vlákna

V případě provádění časově náročných úloh na pozadí se jeví jako užitečné, mít možnost tuto operaci kdykoliv ukončit. Vlákno se během provádění dá zastavit několika způsoby. Klasický způsob je založen na spolupráci, kdy se nejedná o násilné ukončení. Dlouho trvající operace neustále kontroluje, zda nebylo požádáno o její ukončení. Pokud ano, pak operace sama zodpovídá za své řádné ukončení a následném uklizení prostředků. Takovéto ukončení je založeno na třídě *CancellationTokenSource*, která slouží k odeslání žádosti o zrušení. [18]

4.2.2 Paralelní programování

Paralelní programování by se mělo používat vždy, když dochází k provádění složitých a časově náročných operací, které mohou být rozděleny do více menších úkolů. Paralelní programování zvyšuje dočasně využití *CPU* za účelem vylepšení propustnosti. Toto lze provést na straně klienta, kde je *CPU* většinu času nečinné. Na druhou stranu tento efekt není moc žádán na straně serveru.

Existují dvě formy paralelismu, a to paralelismus datový a paralelismus úloh. Datový paralelismus nastává, když je zapotřebí zpracovat spoustu dat, která jsou na sobě navzájem nezávislá. Paralelismus úloh se používá v případě nutnosti zpracovat velkou úlohu či více úloh, kdy většinu částí úlohy lze zpracovat nezávisle na zbytku. [19]

Paralelismus lze provést několika způsoby. *C#* poskytuje tyto metody *Parallel.ForEach* a *Parallel.For*. Tyto metody se v jádru nijak neliší od obyčejných metod *ForEach* a *For*. Hlavním rozdílem metod třídy *Parallel* od normálních metod je v jejich zpracování. Normální metoda *ForEach* prochází na jednom vláknu pěkně popořadě danou kolekci, se kterou pracuje. Metoda *Parallel.ForEach* spustí několik vláken, z nichž každé prochází smyčkou s jiným prvkem z kolekce. Jednotlivé operace probíhají stejně rychle jako u obyčejného *ForEach*. Nicméně z pohledu zpracování celé kolekce je zajisté rychlejší případ, kdy se na stavbě domu podílí deset zedníků místo jednoho.

5 METODY PRO PRÁCI S LOGOVACÍMI SOUBORY

Způsobů, jak pracovat s *logovacími* soubory, je mnoho a každý z těchto způsobů má své pro i proti. Při práci s *logovacími* soubory rozumné velikosti lze tyto soubory nahrát celé do paměti. Takovýmito soubory jsou myšleny soubory, které dokáže počítač zpracovat přibližně do 60 sekund. Načtením celého souboru najednou narůstá vytížení paměti. Operační systém poskytuje aplikacím pouze *2GB* paměti *RAM* k dispozici. Pokud aplikace překročí toto omezení, program skončí s chybovou hláškou *out of memory*. Z tohoto tedy vyplývá, že rozsáhlé *logovací* soubory nelze načíst celé najednou do paměti a lze s nimi tedy pracovat pouze jako s proudem dat.

V případě, že aplikace poběží na počítači s klasickým *HDD*, je rychlost práce se souborem značně omezena rychlostí disku. Pokud ovšem aplikace běží na počítači s diskem *SSD*, který poskytuje mnohem rychlejší čtení i zápis, se rychlost práce se souborem značně odvíjí od některé z následujících metod.

5.1 Paralelní metody zpracování

Při paralelních metodách se nabízí využití cyklů *Parallel.For* a *Parallel.Foreach*. Tyto cykly fungují tak, že nad kolekcí, kterou prochází, vytváří nová vlákna. Každé vlákno poté projde cyklem a provede veškeré operace nacházející se uvnitř. Průchod každého vlákna snižuje počet zbývajících iteračních kroků. Správná volba procházení cyklů se odvíjí od informací, které známe a od výhod, které dané metody přináší. Použití paralelního cyklu *for* je použití nejrychlejšího řešení. I klasická forma cyklu *for*, která běží na jednom vláknu, má časové provedení více než dvakrát rychlejší než cyklus *foreach* nad kolekcí *List*. [20] Hlavní nevýhoda cyklu *for* spočívá v nutnosti znalosti velikosti kolekce, kterou tento cyklus prochází. Z toho vyplývá hlavní výhoda cyklu *foreach* a tedy, že není potřeba vědět, kolik prvků se nachází v kolekci.

Použití paralelních metody při práci s rozsáhlými *logovacími* soubory lze provést například s využitím vzoru producent-konzument. Jedno vlákno se chová jako producent a načítá jednotlivé řádky souboru do zásobníku. Mezitím více vláken (konzumenti) zpracovává načtené řádky v zásobníku. Celý průběh je samozřejmě mnohem rychlejší než práce pouze na jednom vláknu, nicméně nevýhodou je, že dochází ke zpracování řádků náhodně. Tedy jednotlivé řádky nejsou zpracovávány popořadě. Tento fakt se stává problémem při práci s jistými *logovacími* soubory, které zapisují jednotlivé události na více než jeden řádek do souboru a pouze na prvním řádku se nachází identifikační časová známka. Další řádky poté obsahují

pouze zbytek zprávy o události. Pokud každý řádek neobsahuje svůj vlastní identifikátor, který slouží k jednoznačné identifikace, není možné výsledné zpracované údaje seřadit postupně tak, jak patří.

5.2 Zpracování jedním vláknem

V případě zpracování souboru jedním vláknem se naskýtá možnost využití metod *StreamReaderu*. Tato metoda čte soubor jako proud bajtů. Nedochází tedy k načtení celého souboru do paměti najednou, a nemůže tedy dojít k nedostatku paměti. Metoda *StreamReader* také poskytuje funkci *ReadLine*, která čte ze souboru přesně jeden řádek. S využitím cyklu *while* lze pročíst celý soubor od začátku až do konec, řádek po řádku. Každý řádek se následně zpracovává zvlášť a výsledek se ukládá do kolekce, například do listu. V tomto listu se nachází všechny řádky souboru popořadě.

Díky této metodě odpadají problémy s pořadím, ale celá operace se stává mnohem pomalejší. Takovéto čtení rozsáhlých logovacích souborů je velice časově náročné a při čtení opravdu rozsáhlých souborů i nemožné z důvodu nedostatku paměti. Načítání lze také vyřešit následovně:

Nedochází k načtení celého souboru najednou, ale načte se pouze pár tisíc řádků. Další řádky jsou načítány ve chvíli, kdy se uživatel nachází pod 75% načtených řádků.

6 GRAFICKÉ UŽIVATELSKÉ ROZHRANÍ

Takovéto rozhraní se specifikuje tím, že jeho jednotlivé části (objekty) mají grafickou reprezentaci a uživatel může s těmito objekty manipulovat skrze různá zařízení (např. myš, dotyková obrazovka atd.).

Opravdu dobré uživatelské rozhraní splňuje tyto podmínky: [21]

- Složitost – Návrh by měl být co nejméně složitý. Při tvorbě návrhu by se mělo předejít tzv. chytrým návrhům.
- Údržba – Údržba tohoto návrhu by měla být snadná.
- Spojení – Návaznost mezi ostatními částmi programu by měla být udržena na minimu.
- Rozšiřitelnost – Provedení změn by mělo být možné bez větších zásahů.
- Opakovatelnost – Systému by měl být navržen tak, že jeho jednotlivé části lze použít i v jiných systémech.
- Přenositelnost – Návrh by mělo být možné přenést do jiného vývojového prostředí.
- Hubenost – Návrh by neměl obsahovat žádné přebytečné části a měl by se skládat pouze z nutných částí.
- Běžné techniky – Celý systém by měl používat co nejméně speciálních technik. Návrh se tak stává lépe „čitelným“.

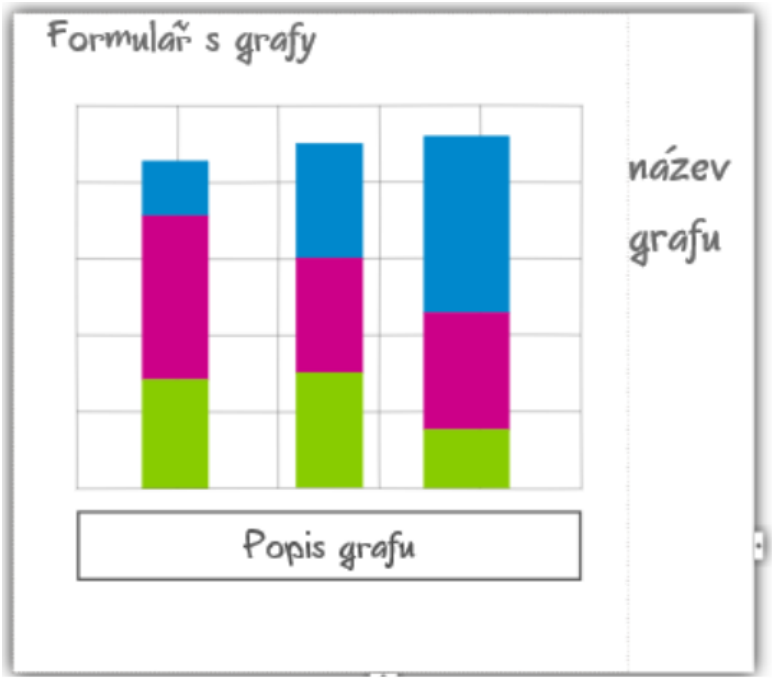
6.1 Specifikace uživatelského rozhraní

V této podkapitole bude uveden návrh uživatelského rozhraní pomocí tzv. *wireframe*. Tyto návrhy byly vytvořeny pomocí webové aplikace *ninjamock*. Jako první bude uveden hlavní formulář, který se zobrazí po startu aplikace.

Obr. 4. Specifikace uživatelského rozhraní – hlavní formulář

- Obsahem hlavního výběrového menu budou všechny dostupné funkce aplikace. Menu bude rozděleno na záložky, do kterých budou spadat patřičné funkce.
- Tabulka s obsahem událostí se objeví až po zvolení souboru k přečtení.
- Kliknutím na jednotlivé funkce se provede patřičná metoda. Funkce budou mít stručný popis.

Dále bude aplikace zobrazovat formulář s grafy po kliknutí na funkci, která tuto akci vykoná.



Obr. 5. Specifikace uživatelského rozhraní – formulář s grafy

- Grafy budou rozděleny do záložek. Na každé záložce se bude nacházet jiný graf.
- Název grafu říká, co daný graf představuje
- V popisu grafu by se měl nacházet podrobný popis grafu.

Součástí bude také pár formulářů, které se zobrazí například při vytváření grafu či profilu. Účel těchto formulářů spočívá ve výběru, ze které právě otevřené záložky má být daný graf či profil vytvořen. Pro představu následuje ukázka jednoho takového formuláře (Obr. 6).

Obr. 6. Specifikace uživatelského rozhraní – formulář výběru grafu

- Seznam *radiobuttonů* by měl být vytvořen z momentálně otevřených souborů. *Radiobuttony* umožňují volbu pouze jednoho ze všech dostupných. Z toho vyplívá, že lze vytvořit pouze graf z jednoho souboru.
- Výběr se potvrdí potvrzovacím tlačítkem, tím dojde k zobrazení formuláře s grafy.

6.2 Specifikace funkcí aplikace

Obr. 7. Specifikace funkcí aplikace – první záložka

- *Nový* – Stiskem tohoto tlačítka dojde k otevření formuláře pro výběr souboru, který chceme otevřít. Po výběru souboru a potvrzení výběru se soubor načte do tabulky s obsahem událostí.
- *Zavřít vše* – Toto tlačítko zavře všechny otevřené záložky (soubory).
- *Systémové události* – Tato funkce slouží k načtení systémových událostí z operačního systému Windows.
- *Události aplikací* – Načte události aplikací z operačního systému Windows.

Obr. 8. Specifikace funkcí aplikace – druhá záložka

- *Sledovat vše* – Tímto tlačítkem se zavolá funkce, která zapne sledování změn ve všech momentálně otevřených souborech. Pokud dojde k zápisu nového záznamu do některého ze souborů, aktualizuje se tabulka s obsahem událostí.

- *Vytvořit profil* – Slouží k vytvoření profilu, který bude obsahovat cesty k vybraným souborům. Vybírat soubory lze z otevřených souborů. Po vytvoření profilu se dostupné profily objeví vedle tohoto tlačítka. Dvojitým kliknutím na daný profil dojde k otevření všech souborů v profilu.
- *Smazat* – Smaže vybraná profil z databáze.
- *Spojit stránky* – Spojí vybrané stránky v tabulce do jedné nové.
- *Grafy* – Slouží k otevření formuláře výběru grafu (Obr. 6).

7 MĚŘENÍ VÝKONU

Měření výkonu představuje zaznamenávání údajů a jejich následnou analýzu. To vše za účelem zjištění jak dobře si daný systém vede v plnění úkolů.

Nikdy nelze s jistotou říci, kde se nachází problémy s výkonem, pokud není vše podrobně změřeno. Programátor může mít pocit, kde tyto problémy nastávají, ale stejně by se mělo provést měření a to z jednoho ze dvou následujících důvodů: [22]

- Programátor ví, kde nastává problém, nicméně neví jak velký je.
- Programátor se může vždy mýlit.

Měření výkonu v informatice se nazývá profilování. Jedná se o proces, při němž dochází ke sledování jistých aspektů programu, za účelem optimalizace. Profilování lze provést například pomocí Visual Studia. Visual Studio dokáže analyzovat využití CPU, paměťovou alokaci atd. [22]

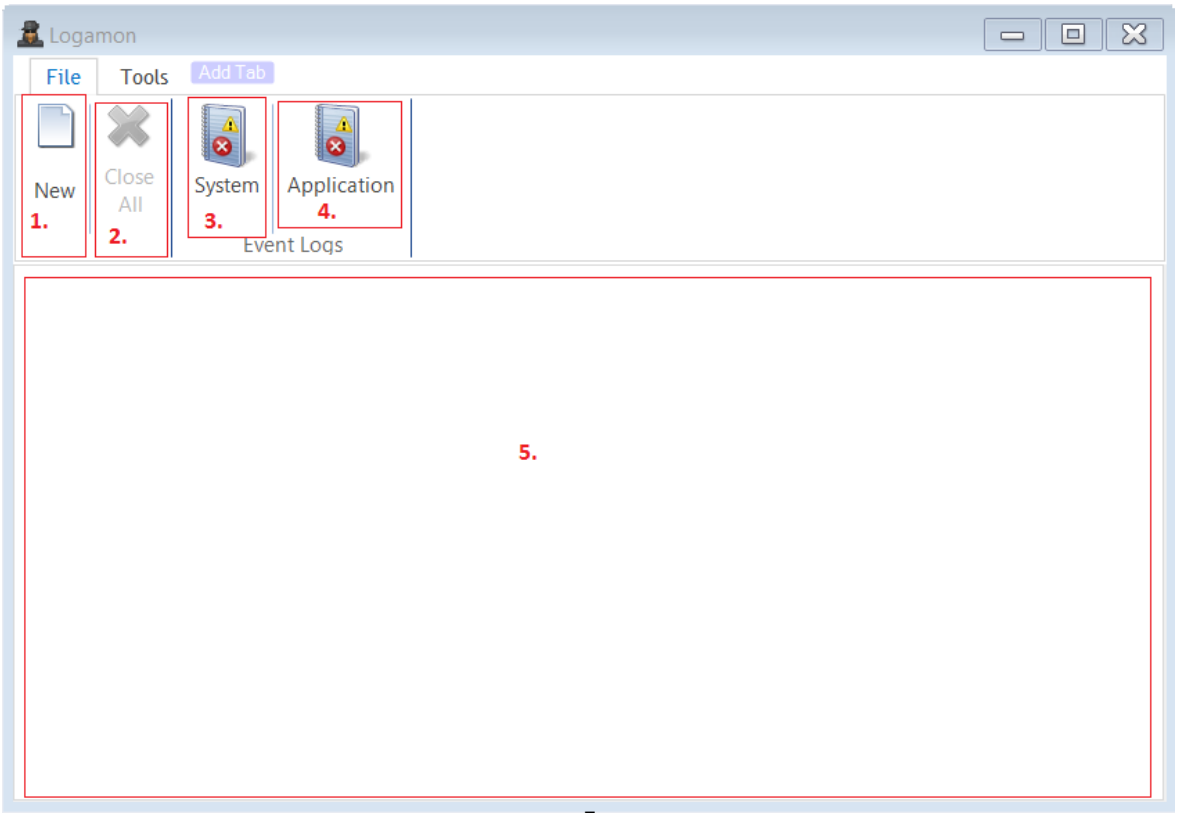
II. PRAKTICKÁ ČÁST

8 IMPLEMENTACE UŽIVATELSKÉHO ROZHRANÍ

Tato kapitola popisuje výslednou aplikaci, která byla vytvořena podle specifikace uživatelského rozhraní a jejích funkcí.

8.1 Hlavní formulář

Úvodní formulář, který se zobrazuje při spuštění programu.



Obr. 9. Výsledný design – hlavní formulář

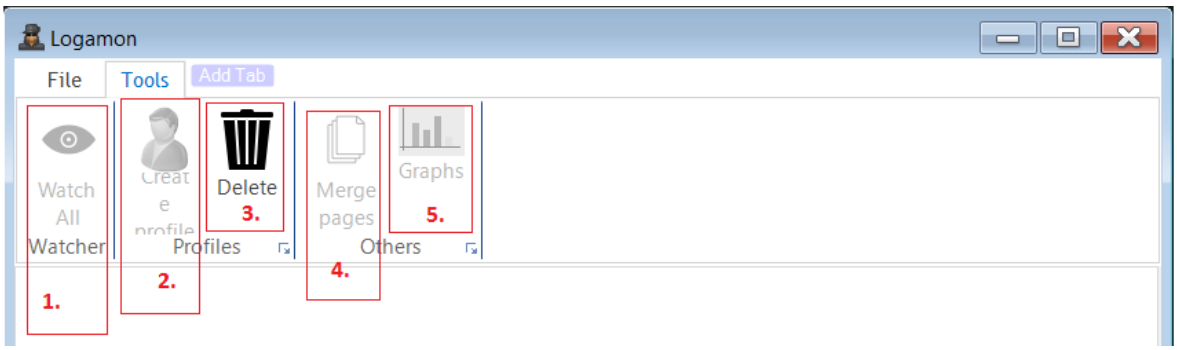
První komponentou je tlačítko *New*. Po stisknutí tohoto tlačítka program vykoná tyto operace. Nejprve dojde k zobrazení dialogového okna pro výběr souboru, který se má otevřít. Kontroluje se, zda bylo dialogové okno ukončeno stisknutím tlačítka *ok* a také zda daný soubor není již otevřen. Pokud jsou tyto kontroly úspěšné, dochází k vytvoření nové instance ovládacího prvku *UserControl* a vytvoření nové záložky v ovládacím prvku *TabControl* (na Obr. 9 znázorněn číslem 5.). Ovládací prvek *UserControl* je přidán na novou stránku komponenty *TabControl* a dochází k vytvoření instance ovládacího prvku *BackgroundWorker*. Tento prvek provádí načtení souboru metodou *Read*, která je součástí třídy *ReadFile*.

Druhá komponenta zavírá všechny stránky (*TabPages*), které jsou otevřeny na ovládacím prvku *TabControl* (na Obr. 9. znázorněn číslem 5).

Třetí komponenta nejprve kontroluje, zda již nedošlo k načtení systémových událostí. Pokud ne, opět se vytváří nová instance ovládacího prvku *UserControl*. Tato instance nyní spadá pod třídu *UserControlEventLog*. Nyní b*ackgroundWorker* volá metodu *ReadEventLog*.

Průběh u čtvrté komponenty je stejný jako u třetí až na to, že dochází k načtení událostí aplikací.

Pátá komponenta představuje ovládací prvek *TabControl*. U tohoto prvku lze vytvářet mnoho stránek, které se nachází pod sebou a lze mezi nimi přepínat formou záložek. Při otevření nového souboru a jeho načtení se na této komponentě vytvoří nová stránka. Na této stránce je umístěn ovládací prvek *UserControl*.



Obr. 10. Výsledný design – druhá záložka menu hlavního formuláře

První ovládací prvek zapíná sledování změn ve všech momentálně otevřených souborech. Kód pracuje následovně. U každé stránky komponenty *TabControl* dochází ke kontrole, zda neobsahuje ovládací prvek typu *UserControlTabControl*, což značí, že se na dané stránce nachází obsah souboru. Pokud ano, pak se nastavuje sledování souboru a kontroluje se, zda nedošlo ke změnám v souboru.

Druhý ovládací prvek umožňuje vytvoření profilu. Při kliknutí na tento ovládací prvek se zobrazí profilový formulář, který obsahuje pole pro zadání názvu profilu a potvrzovací tlačítko. Po zadání názvu a stisku potvrzovacího tlačítka dojde ke kontrole, zda zadaný profil se stejným jménem již neexistuje. Pokud existuje, jsou do něj přidány nové soubory, které se v něm zatím nenachází. Naopak v případě že neexistuje, je otevřena databáze profilů, do které se přiřadí nový profil.

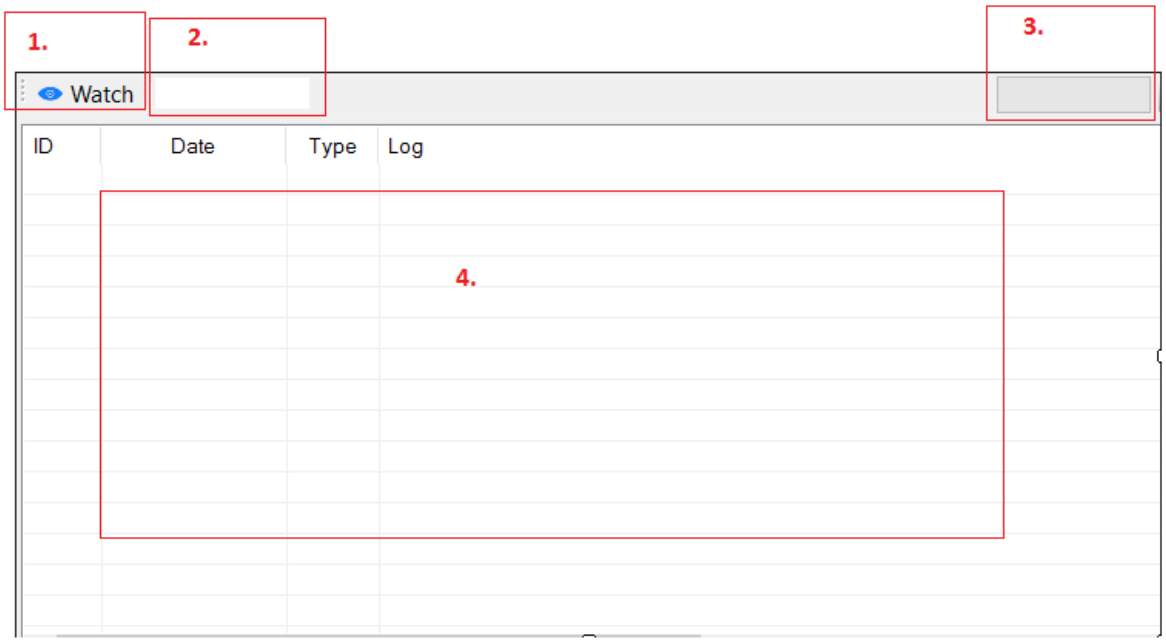
Třetí ovládací prvek slouží k vymazání zvoleného profilu. Ten je odstraněn z databáze profilů.

Čtvrtý ovládací prvek funguje tak, že zobrazí formulář *MergePages*, na kterém se nachází všechny stránky ovládacího prvku *TabControl* a před jejich jmény leží také *checkbox*. Uživatel poté zaškrtnutím *checkboxu* vybere stránky, které chce spojit. Stisknutím tlačítka *Merge* potvrdí svůj výběr a vzniká nová stránka, která obsahuje všechny načtené události ze všech vybraných stránek.

Pátý ovládací prvek otevírá dialogové okno formuláře, který zobrazuje všechny otevřené stránky spolu s *radiobuttonem* vedle jejich jmen. Výběrem jedné stránky a potvrzením výběru stiskem tlačítka *Create* se otevře formulář s grafy.

8.2 Komponenta UserControl

Tato komponenta je umístěna na stránce *TabControl* při otevření souboru.



Obr. 11. Výsledný design – vzhled komponenty *UserControlTabControl*

Tato komponenta se téměř nijak neliší od komponenty *UserControlEventLog*, která slouží k zobrazování událostí systému Windows a komponenty *UserControlMerge*, která zobrazuje spojené stránky.

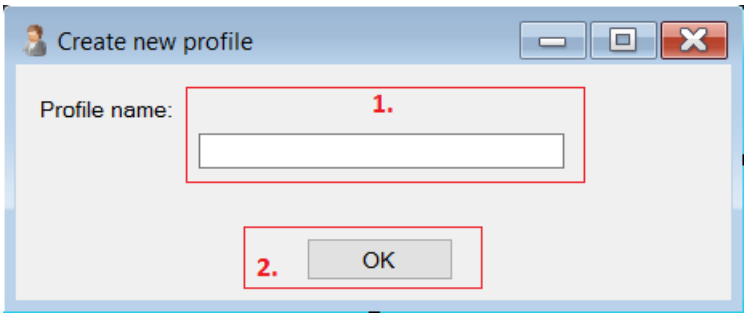
První tlačítko *Watch* slouží k zapnutí sledování změn pro daný soubor, ke kterému tato komponenta náleží. V případě, že dojde ke změně obsahu tohoto souboru, dochází k zavolání opětovného čtení souboru a aktualizaci načteného obsahu.

Druhým ovládacím prvkem je *Textbox*. Ten slouží pro zadávání hledaného výrazu. Tento výraz se následně vyhledává ve všech informacích, které jsou součástí *DataListView* (ovládací prvek č. 4 na Obr. 11).

Třetí ovládací prvek představuje *ProgressBar*, který se aktualizuje při načítání souboru. Jeho hodnota se mění s počtem zpracovaných řádků souboru při jeho načítání.

Posledním ovládacím prvkem je *DataListView*. Ten je rozdělen do čtyř sloupců. Tyto sloupce jsou vyplněny z kolekce, do které se načítají získané údaje ze souboru.

8.3 Formulář ProfileForm

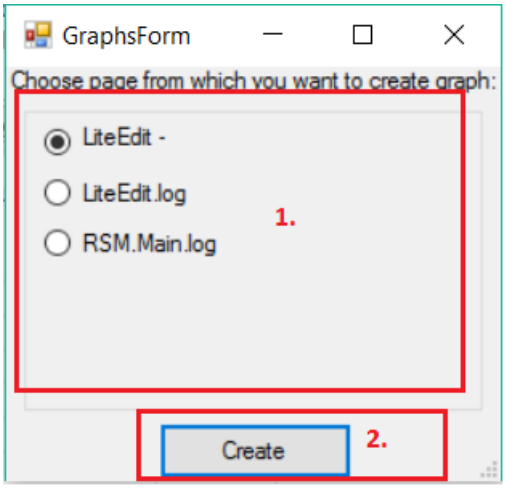


Obr. 12. Výsledný design – formulář pro vytvoření profilu

První ovládací prvek na tomto formuláři slouží k zadávání jména profilu. Po stisknutí druhého ovládacího prvku, potvrzovacího tlačítka *OK,* dochází ke kontrole zadaného textu. Text, který zadal uživatel do *textboxu*, se kontroluje na zakázané znaky či zda není prázdný. Pokud dojde k úspěšnému vyhodnocení kontroly, formulář se uzavře. Text, který zadal uživatel, je předán jako proměnná představující název profilu. Program poté zkontroluje zadaný profil, zda již neexistuje anebo se jedná o profil nový. Pokud profil již existuje, zkontroluje se jeho obsah s obsahem momentálně otevřených stránek. V případě, že je otevřena stránka, která se ve starém profilu nenachází, dochází k aktualizaci profilu. Pokud je obsah totožný, nic se neděje. Pokud profil neexistuje, vytvoří se nový.

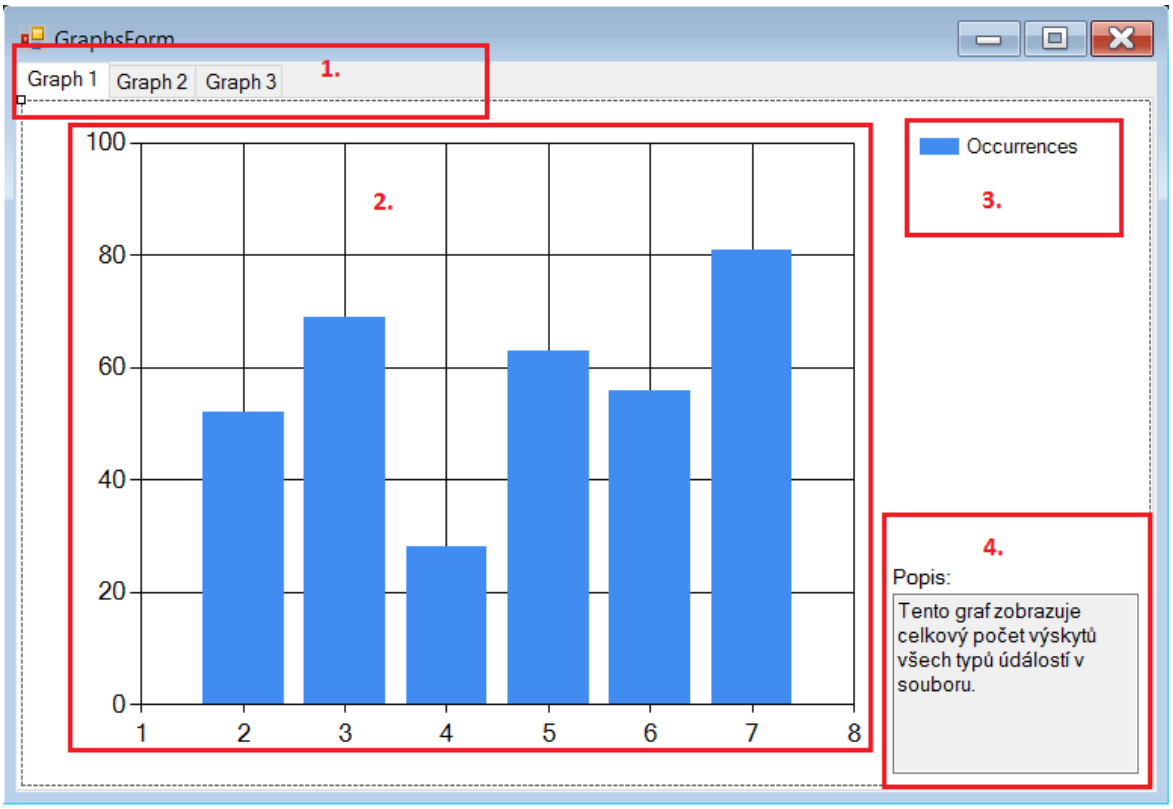
8.4 Formulář zobrazující grafy

Stisknutím tlačítka *Graphs* (pátá komponenta na Obr. 10), dojde k vyvolání okna (Obr. 13) pro výběr stránky, z které se mají vytvořit grafy.



Obr. 13. Výsledný design – formulář pro výběr grafu

První komponenta zobrazuje všechny momentálně otevřené stránky. Vedle jejich jména se nachází *radioButton*, kterým se daná stránka vybírá. Druhou komponentou je potvrzovací tlačítko, které slouží k potvrzení zvoleného výběru. Po potvrzení dojde k zobrazení formuláře, na kterém se nachází všechny grafy (Obr. 14).



Obr. 14. Výsledný design – formulář grafů

První komponentu tohoto formuláře tvoří záložky, na nichž se nachází jednotlivé grafy. Mezi těmito záložkami lze přepínat pouhým stisknutím. Na každé záložce se nachází stejné rozložení komponent.

Druhou komponentu představuje samotný graf. Graf slouží k získání lepší představy načtených dat a k zobrazení závislostí mezi nimi. Třetí komponenta je součástí grafu a definuje název *série*, která je právě zobrazena. Čtvrtou komponentou je vyjádřen popis grafu, tedy přiblížení toho co graf zobrazuje.

9 POPIS TŘÍD APLIKACE

V této kapitole se nachází popis několika důležitých tříd v rámci aplikace.

9.1 Třída Content

Jedná se o veřejnou třídu, do které se získávají údaje každého řádku při načtení logovacího souboru. Třída má tyto datové proměnné:

```
public long Id { get; set; }
public string FileName { get; set; }
public string Log { get; private set; }
public DateTime Date { get; private set; }
public Type Type { get; private set; }
```

Proměnná *Id* představuje číslo řádku. Význam proměnné *FileName* spočívá v definici souboru, ze kterého záznam události pochází. *Log* je proměnná typu *string* a ukládá se do ní celý řádek události. Do proměnné *Date* se ukládá datum a čas, kdy proběhla daná událost. Nakonec proměnná *Type*, která je i stejného datového typu *Type*. Tento datový typ tvoří výčet možných datových typů (*Information, Debug, Warning, Error, Succes audit, Failure audit*).

Tato třída má dva konstruktory. První konstruktor vypadá takto:

```
Id = id;
Log = log;
Date = date;
Type = type;
```

Tento konstruktor se využívá při prvním načtení souboru a jednotlivém *parsování* řádků. Druhý konstruktor využívá i proměnnou *FileName*. Tohoto konstruktoru se využívá při spojení více stránek do jedné. To proto, aby mohl uživatel rozlišit, ze kterého souboru pochází jednotlivé řádky událostí.

9.2 Třída ReadFile

Tato třída poskytuje metodu *Read*, která tvoří základní kámen celé aplikace. Také metodu *ReadEventLog*.

9.2.1 Metoda Read

Metoda má šest vstupních parametrů a tři výstupní. Vstupní parametry:

- `string path` – Představuje cestu k souboru.
- `BackgroundWorker bw` – Instance *backgroundworkeru*.

- `long begin` – Definuje řádek, od kterého se soubor začne načítat.
- `long end` – Určuje, po kolikátý řádek se má soubor načíst.
- `int direction` – Představuje směr načítání řádků. 1 značí směr dolů a 2 směr nahoru.
- `Occurrences Occur` – Slouží k předání odkazu na *list*. Tento *list* udržuje záznamy o tom, kolik se načetlo událostí daného typu.

Výstupními parametry jsou:

- `out long newBegin` – Do této proměnné se ukládá číslo, které představuje, od kterého řádku se bude při příštím zavolání této funkce soubor načítat.
- `out long newEnd` – Slouží k definování, do kterého řádku se má soubor načítat při novém volání metody *Read*.
- `out bool can` – Tato proměnná nabývá hodnoty *true* v případě, že kolekce obsahující načtené prvky překročila požadovanou velikost a první řádky se z ní mohou vymazat.

Nejprve se vytvoří *FileStream* s pravomocemi ke čtení souboru. Také je nutno definovat pravidla pro sdílení, a to jak pro čtení tak zápis. To proto, kdyby došlo k zápisu do souboru při jeho načítání. Z tohoto *FileStreamu* se udělá *StreamReader* pomocí kterého se načítají řádku ze souboru. Nejprve se zjistí počet řádků v souboru. Následně podle velikosti souboru se rozhodne, zda se z časového hlediska vyplácí načíst soubor celý anebo pouze pár tisíc řádků. V obou případech probíhá načítání řádků stejně.

Načte se řádek ze souboru. Celý řádek je prohledán s využitím regulárního výrazu na znaky, které by mohly představovat časovou známku událost.

```
var regex1 = new Regex(@"\d{4}\-\d{2}-\d{2}.\d{2}:\d{2}:\d{2}");
```

Následně jsou tyto znaky převedeny do jednoho ze dvou formátů data.

```
string[] formats = { "yyyy-MM-dd HH:mm:ss", "yyyy-MM-ddTHH:mm:ss" };

DateTime.TryParseExact(m.Value, formats, null, DateTimeStyles.None, out parsedDate);
```

Jako další se provádí tento cyklus:

```
foreach (var item in appconfig.InfoKeys)
         if (line.Contains(item))
         {
            _type = Type.Info;
            Occur.Info++;
         }
```

Cyklus *foreach* prochází všechny prvky, které jsou obsaženy v konfiguračním souboru *service.config*. Tento soubor se nachází ve složce projektu, přesněji v podsložce *debug*. V souboru se nachází veškerá klíčová slova definující typ události. Tento soubor existuje z důvodů možnosti úpravy těchto klíčových slov bez nutnosti zásahu do kódu aplikace. *Appconfig* je instance třídy, která poskytuje obsah souboru *service.config*.

Tento konkrétní cyklus hledá klíčová slova ze sekce *InfoKeys*. Pokud jsou obsaženy někde v řádku, dochází k definování typu události. Také se navýší počet výskytů informačních událostí u proměnné *Occur*.

Takto dochází ke kontrole všech typů událostí. Následně se ukládá nově zpracovaný řádek souboru do *listu*, jehož datový typ je definován třídou *Content*.

```
listFile.Add(new Content(id, parsedDate, _type, line));
```

Také zde dochází ke hlášení, kolik procent práce bylo zpracováno. Procenta jsou vypočítána následujícím vztahem.

```
long percentage = (y + 1) * 100 / length;
```

Kde proměnná *y* představuje počet zpracovaných řádků. Poté zbývá ještě uvědomit ovládací prvek *backgroundWorker* o postupu zpracování souboru.

```
bw.ReportProgress((int)percentage);
```

BackgroundWorker následně aktualizuje ovládací komponentu *ProgressBar*, která se nachází na komponentě *UserControl*.

9.2.2 Metoda ReadEventLog

Tato metoda slouží k čtení záznamu událostí z operačního systému Windows. Metoda má tyto tři vstupní parametry:

- `string type` – Definuje typ událostí, které chceme načíst. Lze načítat systémové události nebo události aplikací.
- `Occurrences Occur` – Předává instanci na kolekci, do které se ukládá počet výskytů jednotlivých typů událostí.
- `BackgroundWorker bw` – Instance *backgroundWorkeru*, na které došlo k volání této metody.

V této metodě se nejprve vytváří nová instance typu *EventLog*. *EventLog* představuje třídu, kterou poskytuje rozhraní *.NET* pro práci s událostmi operačního systému Windows.

```
EventLog myLog = new EventLog(); //new instance of EventLog
```

Dále je nutno definovat typ událostí, které se mají načíst.

```
myLog.Log = type; //Name of the event log file
myLog.MachineName = "."; //Name of computer from which I want to take log
```

Zde se musí definovat také typ kolekce která počítá, kolik událostí a jakého typu bylo načteno. To proto, že události systému Windows obsahují další typy, které v externím souboru nejsou, a při vykreslování grafů by pak mohlo dojít k chybám.

```
Occur.FileName = type;
long y = 0; //variable used for counting number of processed entries
```

Nyní již následuje cyklus *foreach*, kterým se prochází všechny události v dané sekci. Opět stejně jako u metody *Read*, je nutno získat datum a typ události. Datum se dá získat takto:

```
parsedDate = entry.TimeGenerated;
```

Typ události se získá například následujícím způsobem.

```
if (entry.EntryType == EventLogEntryType.Information)
{
  _type = Type.Info;
  Occur.Info++;
}
```

Poté se nově zpracovaná událost vloží do kolekce událostí a cyklus se prochází znovu. I v tomto případě dochází ke hlášení, kolik procent událostí již bylo načteno.

9.3 Třídy Serialization a Deserialization

Tyto třídy slouží k manipulaci se souborem *XML*, který obsahuje profily. Třída *Serialization* poskytuje metodu *XMLSerializer* a Třída *Deserialization* metodu *XMLDeserializer*.

9.3.1 Metoda XMLSerializer

Tato metoda slouží k vytvoření souboru *.xml*, který obsahuje profily. Metoda má dva vstupní parametry. První parametr představuje lokaci, ve které se má soubor vytvořit. Druhým parametrem je databáze profilů, ze které má být soubor vytvořen. Při vytváření souboru aplikace nejprve prochází tři lokace na počítači a kontroluje, zda se v nich nenachází soubor s profily. Těmito lokacemi jsou:

- *C:\Users\All users\Logamon\profiles.xml*
- *C:\Users\"Aktuální uživatel"\AppData\Roaming\Logamon\profiles.xml*
- *„Cesta do složky s projektem"\Logamon\bin\Debug\profiles.xml*

Kód metody vypadá takto:

```
XmlSerializer serializer = null;
FileStream stream = null;
serializer = new XmlSerializer(typeof(db));
stream = new FileStream(filename, FileMode.Create, FileAccess.Write);
serializer.Serialize(stream, profile);
if (stream != null)
    stream.Close();
```

Vytvoří se nová instance *XMLSerializeru*, jejímž typem je *db*. Tento typ představuje databázi obsahující všechny vytvořené profily. Dále se vytvoří instance *FileStreamu* s právy zápisu do souboru, který se nachází na lokaci, kterou definuje proměnná *filename*. Pokud se na této lokaci soubor nenachází, dojde k jeho vytvoření. Využitím metody *Serialize,* kterou poskytuje třída *XMLSerializer*, se zapíše do souboru databáze profilů předána jako parametr profile. *Stream* se poté ukončí a soubor se tak zavře.

9.3.2 Metoda XMLDeserializer

Tato metoda je typu *db*, jelikož jejím výsledkem je databáze profilů. Má jeden vstupní parametr a to cestu k souboru s profily. Pokud existuje soubor s profily tak je pomocí této metody načten. Načtení probíhá ihned při spuštění aplikace jako jedna z prvních věcí.

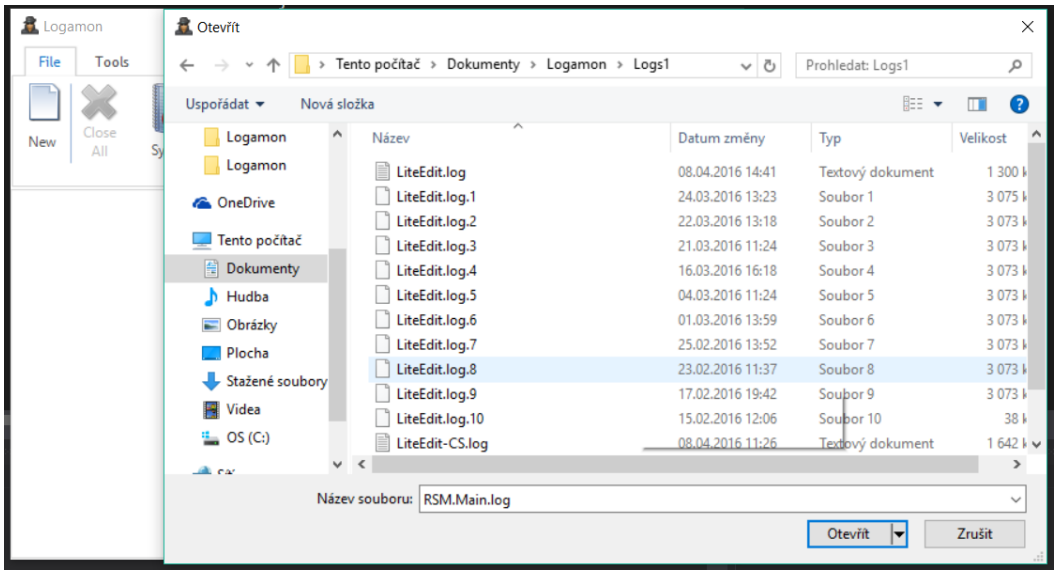
Kód této metody je následující:

```
XmlSerializer serializer = null;
FileStream stream = null;
db database = new db();
serializer = new XmlSerializer(typeof(db));
stream = new FileStream(filename, FileMode.Open);
database = (db)serializer.Deserialize(stream);
if (stream != null)
   stream.Close();
return database;
```

Jako první se vytváří instance nové databáze. Následně je vytvořena instance *XMLSerializer* typu databáze. Vytvoří se nový *FileStream* k souboru s profily s právy čtení. Instance databáze je naplněna metodou *Deserialize*, která převádí soubor z formátu *XML* do objektu daného datového typu. Soubor s profily se zavře a výsledná načtená databáze je navrácena.

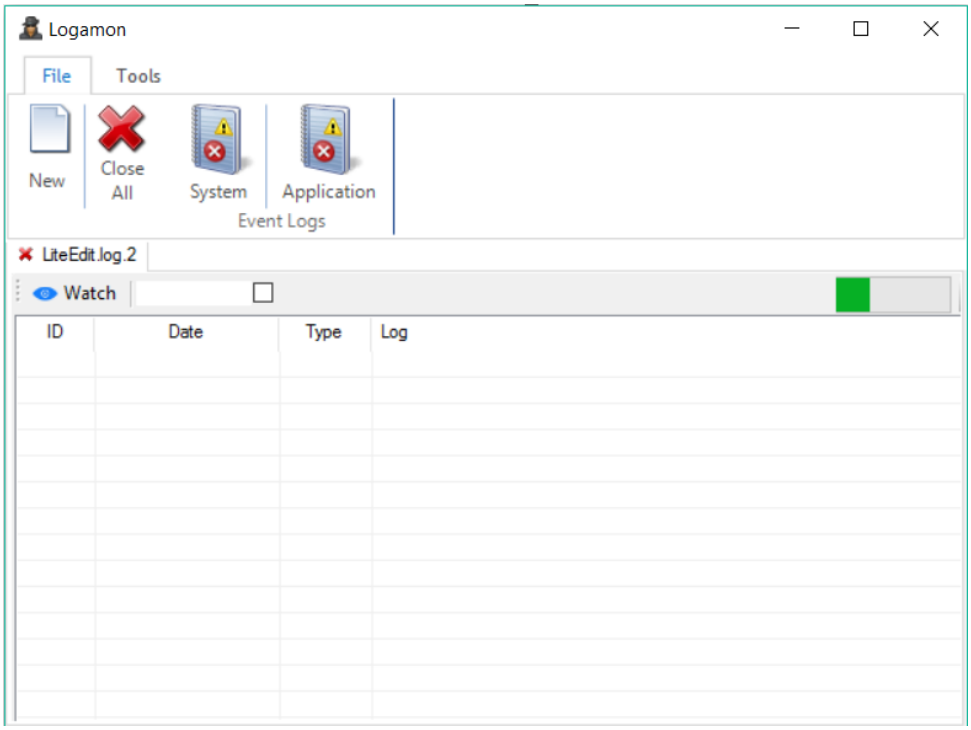
10 POPIS OVLÁDÁNÍ APLIKACE

Zde se nachází ukázka, jak aplikace vypadá v provozu. Po zapnutí a stisknutí tlačítka *New* se zobrazí dialogové okno pro výběr souboru (Obr. 15).



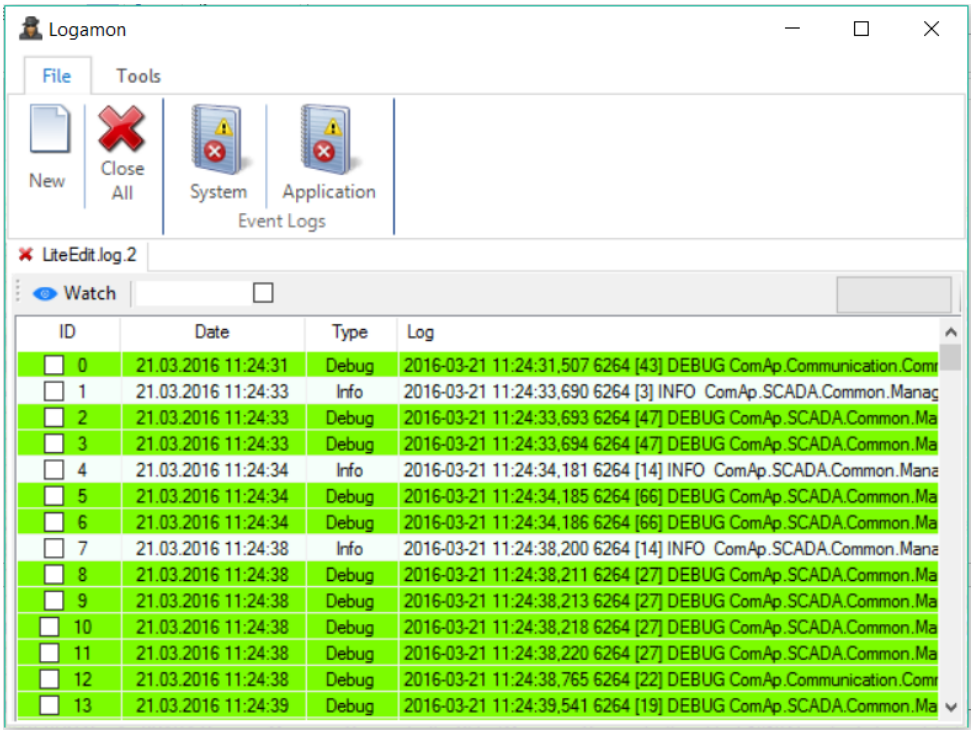
Obr. 15. Test aplikace – výběr logovacího souboru

Po výběru souboru a stisknutí tlačítka *otevřít* se dialogové okno uzavře a v oknu aplikace lze sledovat průběh načítání souboru na *progressbaru* jak je ukázáno v následujícím obrázku (Obr. 16).



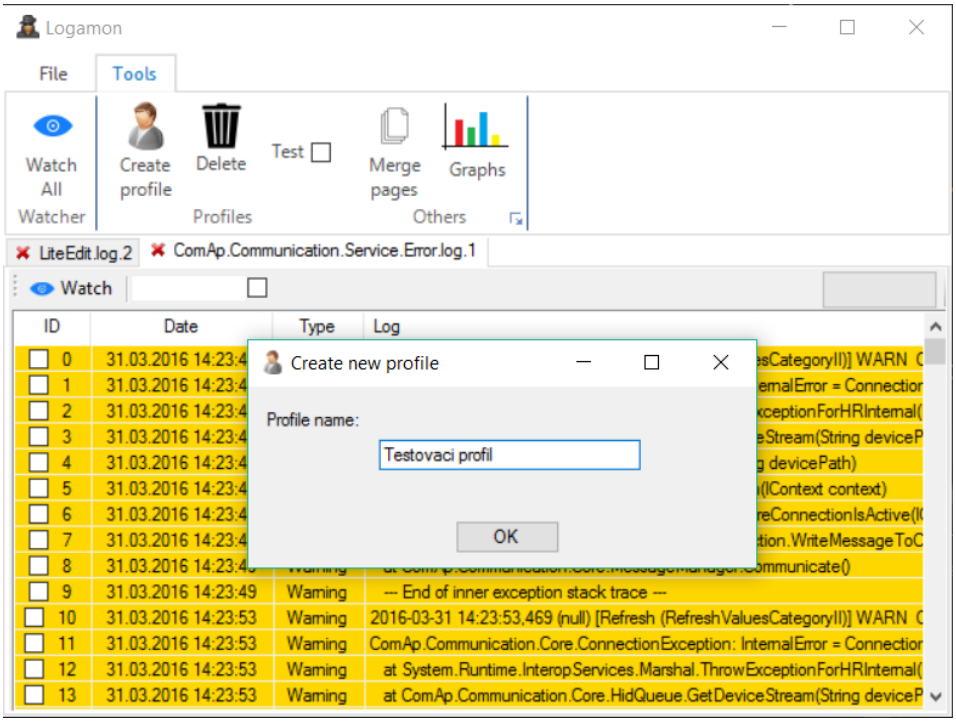
Obr. 16. Test aplikace – načítání logovacího souboru

Po dokončení načítání souboru se zobrazí načtené údaje, které jsou podbarveny podle typu události (Obr. 17). Podbarvení usnadňuje rozlišení událostí mezi sebou.



Obr. 17. Test aplikace – vzhled načtených dat

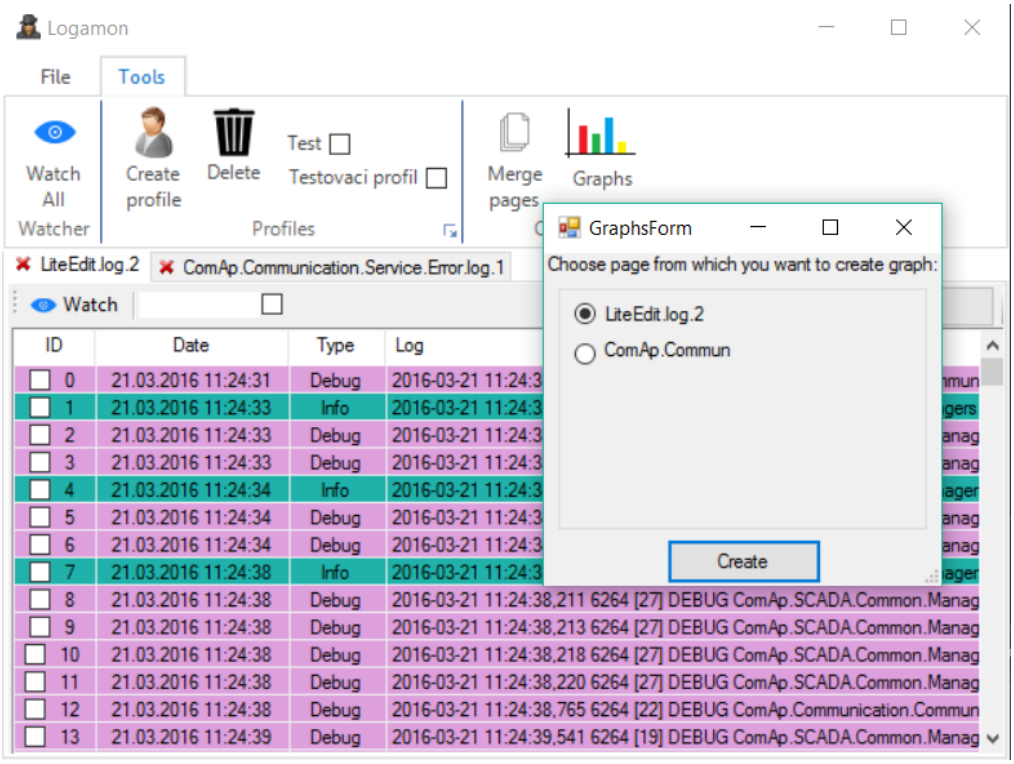
Po načtení ještě jednoho souboru a následném stisknutí tlačítka *Create Profile* se objeví dialogové okno pro zadání názvu profilu (Obr. 18).



Obr. 18. Test aplikace – vytvoření profilu

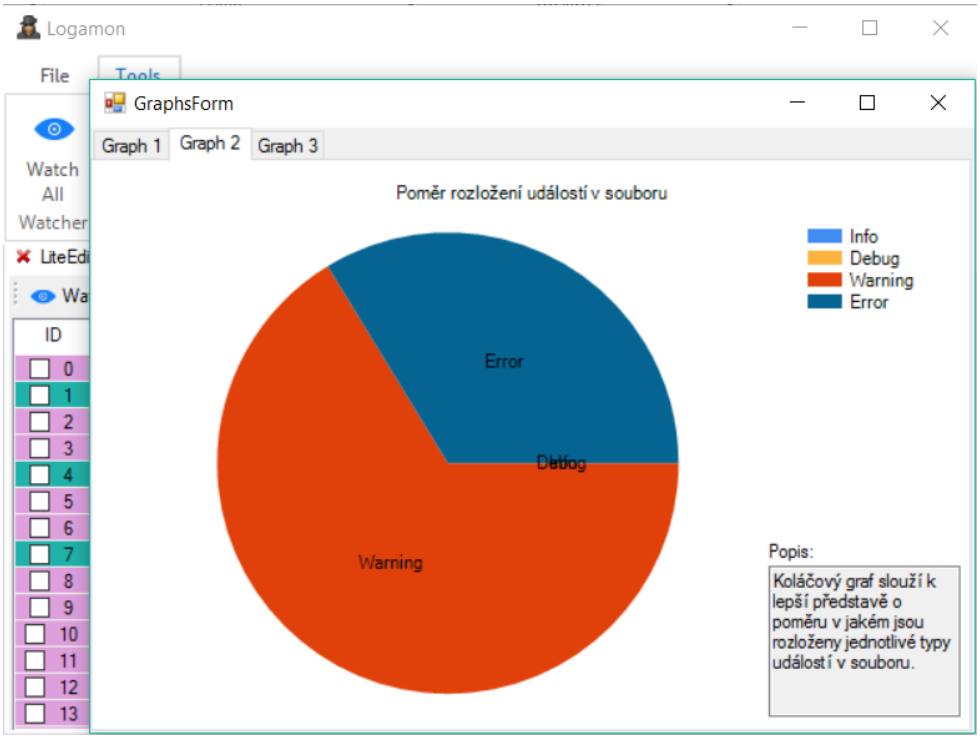
Po ukončení aplikace lze otevřít soubor s profily a manuálně nastavit barvu jakou se mají řádky daného typu události podbarvit. Profil se dá po opětovném zapnutí aplikace otevřít dvojitým kliknutím na jeho jméno. Profil se nyní nachází v sekci *Profiles* na záložce *Tools*.

Stisknutím tlačítka *Graphs* dojde k zobrazení dialogového okna (Obr. 19). Toto okno obsahuje seznam všech právě otevřených stránek. U jména každé stránky se nachází *radiobutton*. Označením *radiobuttonu* dochází k výběru stránky, z níž se mají vytvořit grafy.



Obr. 19. Test aplikace – dialogové okno pro výběr grafu

Následným potvrzením výběru stránky stiskem tlačítka *Create* se dialogové okno uzavře a zobrazí se formulář s grafy (Obr. 20).

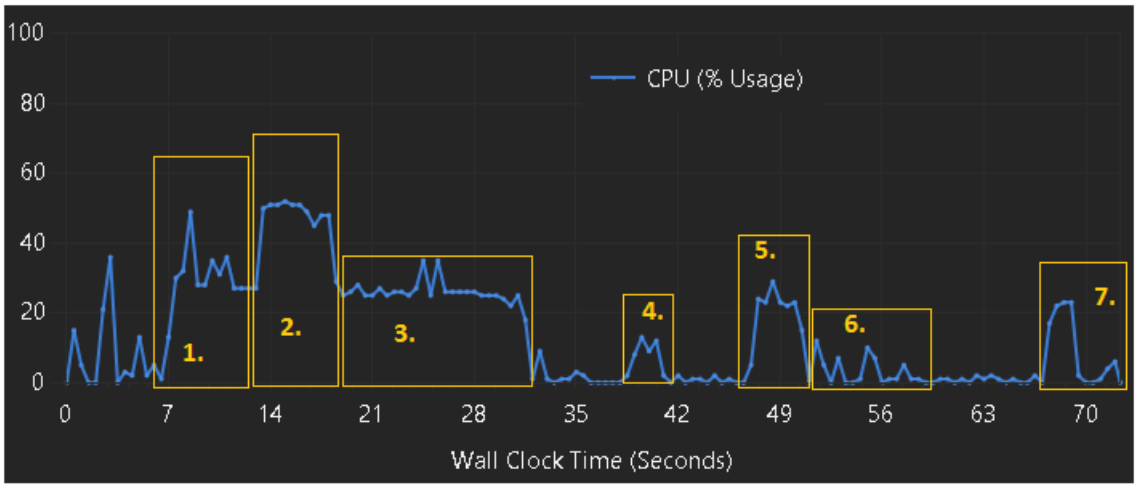


Obr. 20. Test aplikace – formulář obsahující grafy

11 PROFILOVÁNÍ APLIKACE

Poslední kapitola praktické části slouží k popisu chování aplikace z hlediska využití zdrojů počítače (např. CPU a paměť). Měření bylo provedeno až na pár výjimek na stejných operacích jako v kapitole 10.

11.1 Využití CPU



Obr. 21. Profilování – využití CPU

1. Došlo k otevření prvního souboru. Jak lze vidět tak průměrné využití CPU se pohybovalo okolo 30%.
2. Otevřel se druhý soubor a aplikace tak načítala dva soubory v jednu chvíli. Využití CPU vzrostlo tedy nad 50%.
3. Se již dokončilo načítání prvního souboru a proto se opět využití CPU snížilo.
4. Došlo k vytvoření profilu.
5. Spustila se pro spojení více stránek v jednu.
6. Zobrazení formuláře s grafy.
7. Ukončení aplikace.

Jak můžeme z obr. 21. vidět, tak největší využití CPU dosahovalo při načítání souborů. Tedy metoda *Read* vytěžuje procesor nejvíce. Podrobnějším průzkumem kódu lze zjistit, že operace, která nejvíce zatěžuje CPU v této metodě je:

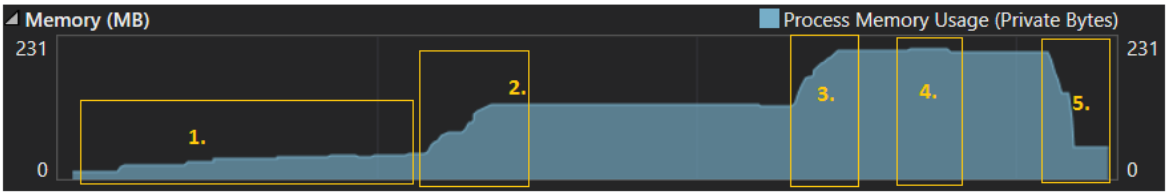
```
var matches = listFile.FirstOrDefault(p => p.Id == id);
```

Zde dochází ke kontrole, zda *list* se všemi načtenými řádky neobsahuje nově načítaný řádek, aby se zabránilo duplikátům. Tato operace představuje až 90% využití CPU z celkového

využití CPU metodou *Read*. Rychlost načítání souboru by se tedy dala vylepšit zvolením jiného způsobu kontroly načítaných řádků.

11.2 Využití paměti

Při testování využití paměti, byly opět provedeny téměř stejné operace jako v kapitole 10.



Obr. 22. Profilování – využití paměti

1. Dochází k načítání prvního souboru.
2. Načítá se druhý soubor.
3. Vytváří se *mergePage* skládající se z obsahu obou načtených souborů.
4. Nepatrně narůstá paměť po zobrazení formuláře s grafy, následně klesá zpět po jeho ukončení.
5. Uvolnění paměti po vymazání všech stránek s načtenými soubory.

Jak lze vidět, využití paměti je opravdu velké při načítání souborů. Nicméně po vymazání stránky, obsahující načtený soubor dojde k uvolnění paměti. Bohužel toto uvolnění není kompletní a v aplikaci dochází tedy k paměťovým únikům. Jinými slovy řečeno nedochází k uvolnění všech prostředků, které dané operace využily.

ZÁVĚR

Výsledkem této bakalářské práce je aplikace usnadňující práci s rozsáhlými logovacími soubory, která byla vytvořena splněním všech funkčních a nefunkčních vstupních požadavků.

Aplikace byla po ukončení stáže plně funkční a připravená pro práci s logovacími soubory, jejichž velikost se pohybuje kolem padesáti tisíc řádků. S novým požadavkem na aplikaci, kterým bylo umožnění práce s rozsáhlými logovacími soubory, vyvstaly nové problémy. Řešení těchto problémů zabralo nejvíce času a energie.

Největší nevýhodou aplikace je její rychlost zpracovávání souborů. První pokusy načítání souborů byly postaveny na paralelním programování. Takovéto řešení by bylo nejefektivnější. Nicméně s tím že při paralelním programování nelze zaručit pořadí, v němž jsou údaje zpracovány, bylo nutno od tohoto řešení upustit. Za předpokladu, že by byl dostatek energie pro řešení tohoto problému, by jej jistě bylo možné vyřešit. Například úpravou logovacího souboru tak, aby každý řádek obsahoval časovou známku. Díky tomu by šlo soubor zpracovat paralelně.

Aplikace by také potřebovala doladit po stránce správy paměti. V aplikaci totiž nedochází zcela k uvolňování prostředků a malá část paměti zůstává zaplněna. Toto zaplňování vede k nutnosti vypnutí aplikace po pár hodinách provozu, aby došlo k uvolnění všech prostředků, jinak může dojít k pádu aplikace.

SEZNAM POUŽITÉ LITERATURY

[1] ŠTORC, Ondřej. Historie .NETu. In: *Itnetwork* [online]. [cit. 2017-05-07]. Dostupné z: https://www.itnetwork.cz/csharp/historie-dotnetu

[2] NET Framework Guide. In: Docs.microsoft [online]. [cit. 2017-05-07]. Dostupné z: https://docs.microsoft.com/cs-cz/dotnet/articles/framework/

[3] JIRAVA, Jarda . *DataBinding a DataTemplate* [online]. [cit. 2017-05-12]. Dostupné z: http://xaml.cz/wpf/databinding-a-datatemplate/

[4] 10 reasons to switch to WPF. *Dzone* [online]. [cit. 2017-05-12]. Dostupné z: https://dzone.com/articles/10-reasons-switch-wpf

[5] Windows Presentation Foundation vs WinForms. *Infragistics* [online]. [cit. 2017-05-12]. Dostupné z: https://www.infragistics.com/community/blogs/devtoolsguy/archive/2015/04/17/windows-presentation-foundation-vs-winforms.aspx

[6] Introduction to Windows Forms. *Msdn.microsoft* [online]. [cit. 2017-05-12]. Dostupné z: https://msdn.microsoft.com/en-us/library/aa983655(v=vs.71).aspx

[7] Custom Controls in Visual C# .NET. *Acadia* [online]. [cit. 2017-05-12]. Dostupné z: http://www.akadia.com/services/dotnet_user_controls.html#User%20controls

[8] MACDONALD, Matthew. User interfaces in C♯: Windows forms and custom controls. Berkeley, Calif: Apress, 2002. ISBN 1590590457.

[9] Features of an ObjectListView. *Objectlistview.sourceforge* [online]. [cit. 2017-05-12]. Dostupné z: http://objectlistview.sourceforge.net/cs/features.html

[10] System.IO Namespaces. *Msdn.microsoft* [online]. [cit. 2017-05-12]. Dostupné z: https://msdn.microsoft.com/en-us/library/mt481548(v=vs.110).aspx

[11] Úvod do kolekcí a genericita. *Itnetwork* [online]. [cit. 2017-05-12]. Dostupné z: https://www.itnetwork.cz/csharp/kolekce-a-linq/c-sharp-tutorial-uvod-do-kolekci-a-genericita

[12] XML Serialization in the .NET Framework. *Msdn.microsoft* [online]. Microsoft Corporation, 2003 [cit. 2017-05-12]. Dostupné z: https://msdn.microsoft.com/en-us/library/ms950721.aspx

[13] *XSD (XML Schema Definition)* [online]. [cit. 2017-05-12]. Dostupné z: http://searchmicroservices.techtarget.com/definition/XSD-XML-Schema-Definition

[14] BĚHÁLEK, Marek. *Programovací jazyk C#* [online]. , 16 [cit. 2017-05-07]. Dostupné z: http://www.cs.vsb.cz/behalek/vyuka/pcsharp/text.pdf

[15] *Professional C# 5.0 and .NET 4.5.1*. 1. Indianapolis: John Wiley, 2014, s. 219-230. ISBN 978-1-118-83294-3.

[16] Delegates. Docs.microsoft [online]. 2016 [cit. 2017-05-16]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/articles/csharp/programming-guide/delegates/

[17] Understand Lambda Expressions in 3 Minutes. In: *Codeproject* [online]. 2013 [cit. 2017-05-13]. Dostupné z: https://www.codeproject.com/tips/298963/understand-lambda-expressions-in-minutes

[18] *Professional C# 5.0 and .NET 4.5.1*. 1. Indianapolis: John Wiley, 2014, s. 378. ISBN 978-1-118-83294-3.

[19] Introduction to Parallel Programming. CLEARY, Stephen. *Concurrency in C# cookbook*. 1. Sebastopol: O'Reilly Media, 2014, s. 7. ISBN 1449367569.

[20] An easy and efficient way to improve .NET code performances. *Codebetter* [online]. 2008 [cit. 2017-05-13]. Dostupné z: http://codebetter.com/patricksmacchia/2008/11/19/an-easy-and-efficient-way-to-improve-net-code-performances/

[21] Design in Construction. *Code complete: umění programování a techniky tvorby software*. 2nd ed. Redmond, Wash.: Microsoft Press, c2004, s. 80-81. ISBN 0-7356-1967-0.

[22] Performance Measurement and Tools. WATSON, Ben. *Writing High-Performance .NET Code*. 1. 2014, s. 15-17. ISBN 978-0-9905834-3-1.

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

API	Application Programming Interface, rozhraní pro programování aplikací.
CLR	Common Language Runtime, společné běhové prostředí platformy .NET.
CPU	Central Processing Unit, základní elektronická součástka v počítači.
GB	Gigabyte, označení jednotky množství informace používané v informatice.
GUI	Graphical user interface, grafické uživatelské rozhraní.
HDD	Hard Disk Drive, zařízení sloužící k ukládání dat.
RAM	Random-Access Memory, elektronická polovodičová paměť.
SSD	Solid-State Drive, pevný disk využívající ne-volatilní flash paměti.
W3C	World Wide Web Consortium, mezinárodní konsorcium.
WPF	Windows Presentation Foundation, knihovna tříd pro tvorbu grafického rozhraní.
XAML	Extensible Application Markup Language, značkovací jazyk pro popis grafického rozhraní.
XML	Extensible Markup Language, strukturovaný značkovací jazyk.
XSD	XML Schema, představuje XML schéma.

SEZNAM OBRÁZKŮ

SEZNAM PŘÍLOH