

Střední průmyslová škola Třebíč

Maturitní práce

Hashtesting

Profilová část maturitní zkoušky

Studijní obor: Informační technologie

Třída: ITA4

Školní rok: 2024/2025 Kamil Franek

Zadání práce



ABSTRAKT

Maturitní práce na téma hashování, jeho využití a typy hashovacích algoritmů. Zabývá se problematikou spojenou s hashováním a vysvětlení použití hashování v IT. Tento dokument popisuje použitou technologii, praktiky a vytváření samotného programu a všeho okolo. Výsledný program disponuje základními i rozšířenými funkcemi práce s hashem a soubory pro zvýšení efektivity práce. Dále disponuje okénkem pro informace, vysvětlení rozdílů mezi hashovacími algoritmy, rozdíl mezi použitím a nepoužitím soli a dalších informací za účelem zvýšení chápání daného téma pro uživatele.

KLÍČOVÁ SLOVA

Maturitní práce, Výuka, Hash, Kybernetický útok

ABSTRACT

\*abstrakt anglicky\*

KEYWORDS

\*klíčová slova anglicky\*

PODĚKOVÁNÍ

Děkuji Ing. Ladislavu Havlátu a oponentu Ing. Drahomíru Škárkovi za cenné připomínky a rady, které mi poskytli při vypracování maturitní práce.

V Třebíči dne 22. ledna 2025 podpis autora

PROHLÁŠENÍ

Prohlašuji, že jsem tuto práci vypracoval/a samostatně a uvedl/a v ní všechny prameny, literaturu a ostatní zdroje, které jsem použil/a.

V Třebíči dne 22. ledna 2025

podpis autora

Obsah

[Úvod 7](#_Toc188214175)

[1 Teorie hashovaní 8](#_Toc188214176)

[1.1 Používané Hashe 8](#_Toc188214177)

[1.1.1 MD5 8](#_Toc188214178)

[1.1.2 SHA1/SHA256/SHA512 8](#_Toc188214179)

[1.1.3 RIPEMD160 9](#_Toc188214180)

[1.1.4 CRC32 9](#_Toc188214181)

[1.2 Používání Hashů 10](#_Toc188214182)

[1.3 Problémy s Hashy 11](#_Toc188214183)

[1.3.1 Pravděpodobnost kolize u CRC32 mezi dvěma hashy 11](#_Toc188214184)

[1.3.2 Pravděpodobnost kolize při větším počtu hashů (Birthday paradox) 12](#_Toc188214185)

[1.3.3 Rychlost hashování a pravděpodobnost kolize každou sekundu 12](#_Toc188214186)

[1.4 Sůl, Pepř a jejich používání 13](#_Toc188214187)

[2 Použité programy a technologie 14](#_Toc188214188)

[2.1 C# 14](#_Toc188214189)

[2.2 Unit-testy 14](#_Toc188214190)

[2.3 Visual Studio 2022 14](#_Toc188214191)

[2.4 Visual Code 14](#_Toc188214192)

[2.5 Git/Github/Github Desktop 14](#_Toc188214193)

[2.6 Freelo.io 14](#_Toc188214194)

[2.7 Word 14](#_Toc188214195)

[2.8 ChatGPT (AI) 14](#_Toc188214196)

[2.9 Online hashers 16](#_Toc188214197)

[3 Praktická část 17](#_Toc188214198)

[3.1 Hasher 17](#_Toc188214199)

[3.2 Gradual Hashing 17](#_Toc188214200)

[3.3 Multiformuláře 17](#_Toc188214201)

[3.4 Unit-Testy 17](#_Toc188214202)

[3.5 UI 18](#_Toc188214203)

[3.6 Save/Load Systém (Settings) 19](#_Toc188214204)

[4 Fungování programu 22](#_Toc188214205)

[Závěr 23](#_Toc188214206)

[Seznam použitých zdrojů 24](#_Toc188214207)

[Seznam použitých symbolů a zkratek 25](#_Toc188214208)

[Seznam obrázků 26](#_Toc188214209)

[Seznam tabulek 27](#_Toc188214210)

[Seznam příloh 28](#_Toc188214211)

Úvod

Cílem této ročníkové práce a programu je zjednodušení práce s hashema, ukázku rozdílů mezi hashovacími algoritmy, používání soli a pepře a podrobné vysvětlení, postupné hashování pomocí mezikroků, silné a slabé stránky hashů, kde a proč se používají. Dále program obsahuje test prolomení hesla (útok hrubou silou) pomocí hrubého útoku. V dokumentu jsou popsány použité programy a technologie, jak jsou použité a proč jsou použité. V praktické části je popsána celá cesta dělání programu, hlavní problémy, trable a vysvětlení fungování celého programu s ukázkami samotného kódu, testování a různé obrázky z pracovního postupu. V závěru jsou popsány moje pocity z práce na projektu a spokojenost s finální verzí programu.

# Teorie hashovaní

Hashování je matematický algoritmus pro převod dat do předem určitého dlouhého výstupu podle algoritmu tzv. hashovací funkce. Hashe mají několik výtečných vlastností: vstupní data můžou býti jakkoliv dlouhá, minimální změna v datech znamená velký rozdíl ve výstupech, s větší výstupní délkou se exponenciálně zmenšuje šance na stejnost výstupních hodnot při jiném vstupu a ta nejdůležitější, nedá se získat z výstupních dat vstupní data (bez použití kybernetických útoku), znamenaje, že proces je jednosměrný. Díky tomu se hash bere jako unikátní otisk vstupních dat. [1]

## Používání Hashů

Hashe se používání k uschování důležitých informací (například hesel), kde pro bezpečnost nechceme dostat vstupní data zpátky, dělání kontroly a integrity dat (kontrolní součet), vytváření a ověřování elektronického podpisu (třeba pro bankovnictví nebo email ověření), hledání škodlivého malwaru antivirovým programem, k hledání úseků DNA sekvencí atd. [1]

## Použité Hashe

Existuje spoustu hashovacích funkcí a každá má svoje výhody, nevýhody a využití pro jiné účely. Tady je informace pro hashovací funkce, které jsou použity v programu.

### MD5

MD5 (Message-Digest Algorithm) pochází z rodiny „Message-Digest“ neboli algoritmus na strávení zprávy. Předchůdci MD5 jsou hashovací funkce MD2 a MD4, všechny tři vytvořeny a vydány Ronaldem Rivestem. MD2 byl vydán v roce 1989, MD4 jakožto pokračovatel v 1990 a MD5 jakožto vylepšená verze MD5 v roce 1991. MD5, na rozdíl od svých předchůdců, je docela složitý algoritmus na rozlousknutí . Používá 4 kola, místo 3 kol jako MD4, a pomocí matematických operací s maticema vypočítá výstup. Délka výstupu hashe je 128 bitů. I přes jeho používání v tehdější a dnešní době se v MD5 našla řada chyb, které by mohly být při ukládání hesel závažné. MD5 je totiž poměrně náchylný na takzvaný brute force attack, česky útok hrubou silou. [1][2]

\*Příklad 1. kola ze 4 u MD5

### SHA1/SHA256/SHA512

SHA, Secure-Hash Algorithm neboli bezpečná hashovací funkce je další velice známý a používaná hashovací funkce. SHA se bere za nástupce MD5 s větší bezpečností a delším výstupem (SHA1 – 160 bitů. SHA256 – 256 bitů, SHA512 – 512 bitů). SHA1 byla první verze SHA vydaná v roce 1995, nepočítaje SHA-0, což byla rychle zapomenutá „před“ verze SHA-1. SHA je rodina vytvořena a zveřejněna americkým ústavem pro technologické standardy (National Institute of Standards and Technology [NIST]). [3][4]

V roce 2005 byl na SHA1 nalezen možný útok a proto v roce 2010 vyšla skupina SHA-2, což je skupina několika hashovacích funkcí, u kterých se mění délka výstupu (pro nás důležité SHA 256 a SHA512, dále se nachází SHA-224 a SHA-384. [5] Čísla na konci znamenají délku výstupu v bitech). Skupina SHA-2 se dodnes považují za bezpečné hashovací algorithmy pro integritu dat a ukládání hesel.[5]

Funkce SHA-256 se využívá ve virtuálních měnách jako třeba Bitcoin. Hlavní premise takzvaných těžiček je najít vstup zahashovaného textu pomocí SHA-256 funkce. [6]

„Google oznámil, že se mu [podařilo prolomit bezpečnost](https://security.googleblog.com/2017/02/announcing-first-sha1-collision.html) hašovací funkce SHA-1. Od [první publikované slabiny](https://www.schneier.com/blog/archives/2005/02/cryptanalysis_o.html) po úspěšný útok tak uběhlo deset let. Dva roky trvalo vědcům z CWI Institute in Amsterdam a společnosti Google, než dokončili práci na slabině a přinesli důkaz toho, že kolizní funkce existuje a významně urychluje útok.“ [7] Přikládám URL adresu projektu SHAttered: <https://shattered.io/>

### RIPEMD160

RipeMD160 je hashovací funkce, která měla za účel nahradit MD4 a MD5, stejně jako SHA1. Hlavní rozdíl je, že RipeMD160 byla vyvinuta v EU jakožto součást projektu RIPE (RACE Integrity Primitives Evaluation, 1988-1992). RipeMD160 byla vytvořena Hansem Dobbertinem, Antoonem Bosselaersem a Bartem Preneelem. Spolu s RipeMD128, RipeMD256 a RipeMD320 byly vydány v roce 1996. Všechny tyto verze vstávají z originální RipeMD hashovací funkce, která byla vydána roku 1992. Dnes všechny tyto verze RipeMD nejsou doporučovány používat, i když v lepších verzích RipeMD (vydány v 1996) nebyly nalezeny žádné kolize. [8]

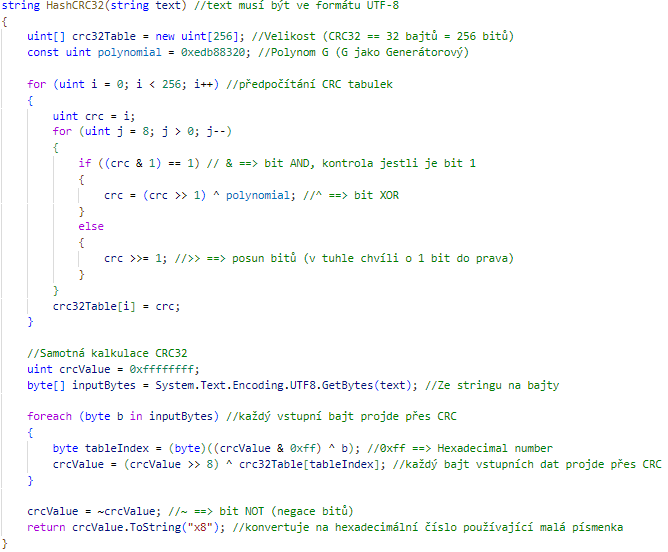
V originálním RipeMD z roku 1992 byla nalezena kolize v roce 2004. Pro maximální bezpečnost se může používat Hash160, což je kombinace SHA256 a RipeMD160, kde vstup prvně projde SHA256 a výsledek poté projde přes RipeMD160. Hash160 se používá u Bitcoinu jako identifikátor pro úschovu peněženky. [9] <https://www.hash160.com/>

### CRC32

CRC32 je hashovací funkce z rodiny cyclic redundancy check neboli cyklická redundantní součet (CRC) je hashovací funkce zaměřená na kontrolní součet a integritu dat. Kvůli tomuto využití jsou všechny CRC velice rychlé. Rodina CRC byla zveřejněna W. Wesley Petersonem v 1961. Nejznámější a nejpoužívanější funkce z rodiny CRC jsou CRC32, která se používá v „ISO 3309 (HDLC), ANSI X3.66 (ADCCP), FIPS PUB 71, FED-STD-1003, ITU-T V.42, ISO/IEC/IEEE 802-3 (Ethernet), SATA, MPEG-2, PKZIP, Gzip, Bzip2, POSIX cksum, PNG, ZMODEM atd.“ [10], a CRC16, které se používá v Bluetooth, SD, X.25, v různých mobilních sítích, USB a mnoho dalších. [11]

Wesley Petersonovi byla za návrh CRC udělena cena „Japan Prize“ v roce 1999. [12] Co se týče hashování důležitých dat, klíčů a bezpečnostních prvků nemá CRC mnoho využití, ale jakožto funkce pro kontrolní součet jich má nespočet.

#### Výpočet CRC

Základ celého výpočtu CRC jsou polynomy. Každý jeden bit v určité délce může být zapsán do polynomu. Příklad (1101) může být zapsán jako . poslední bit reprezentuje 1, druhý bit x, třetí atd. CRC32 má až a CRC16 až . Tento výpočet pro CRC32 je v mém kódu reprezentován druhým cyklem for (uint j = 8; j > 0; j--); Výpočet se prodeve tolikrát, kolik je výstupní délka v bitech (pro CRC32 to je 256, pro CRC16 to je rovno 128). Náš vytvořený polynom je poté porovnán operací XOR (Exclusive OR, které vrací 0, pokud jsou obě hodnoty stejné a 1, pokud jsou jiné).

Další část je porovnávání vstupních bajtů s bajty v tabulce. Index tabulky je zjištěn pomocí předběžného výstupu CRC (pro první projití je do hodnoty CRC nastavena nejvyšší hodnota), operací AND (AND vrací log. 1 pouze pokud jsou oba bity log. 1) a pomocí operace XOR se vstupem. Dále se na CRC výstupu posune 8 bitů do prava, tímto se 8 bitů vpravo efektivně ztratí a provede se XOR operace s tabulkovou hodnotou na indexu předchozího výpočtu. Na závěr se celý výstup ještě neguje bit po bitu (operace NOT). [10][11]

## Sůl, Pepř a jejich používání

Hashovací sůl a pepř jsou další vrstvou pro bezpečnost hashování. Sůl je náhodně vygenerována před generováním a dává se před samotnými daty. Může mít jakoukoliv délku, záleží na správci, který bude hodnotu ukládat. Díky soli se chráníme před takzvanými rainbow tables útoky a útoky hrubou silou.

Pepř je podobný jako sůl, jenže je většinou krátký, dává se na konec dat místo před data a nikde se neukládá. To znamená, že musíme provést všechny možné kombinace hashů a porovnávat výsledné hashe abychom zjistili shodu. Jediná nevýhoda pepře je, že musíme hashovací a porovnávací proces dělat několikrát, což výrazně zvýší prodlevu. V praxi se používají sůl i pepř pro maximální ochranu hesel. [13]

Příklad: Máme heslo “TestingPassword123”. Díky soli se před heslo vygeneruje sůl “a0\_X”, která je někde v tabulce uložena. Před heslo se vygeneruje pepř o délce jednoho ASCII znaku. To znamená že při každém pokusu o přihlášení se před heslo dá sůl a za heslo se postupně zkouší “000”, poté “001”, “010” a tak dale, dokud se nevyzkouší všechny kombinace pepře. Pokud žádný z těchto pepřových kombinací nevýjde, heslo je zadáno špatně.

Všechny hashe jsou šifrované v hashovací funkci MD5.  
(bez soli) TestingPassword123 == 1d898af5dbe7c9e07fc473e248f623a1  
(se solí) a0\_XTestingPassword123 == 5a6c1e14762baf73406b7267a8afae88  
(sůl I pepř) a0\_XTestingPassword123C == d7d0d822ce9faea482a0c5ae372d0ed0

## Kybernetické útoky na hashe

Hashe dokážou zpracovat jakékoliv množství dat a vrátit jenom určitou délku, to ovšem znamená menší problémy. Různé vstupní data mohou vracet stejnou hodnotu hashe, což v případě, že používáme hashe pro ukládání hesel znamená velký bezpečnostní problém. Šance kdy se něco takového může stát je závislá na délce výstupního hashe, proto se doporučuje používat delší a bezpečnější hashe pro ukládání důležitých dat (jako třeba hesel), jako třeba SHA-256 či SHA-512.

Pravděpodobnost si můžeme sami vypočítat pomocí jednoduchého zvorečku. Počet všech kombinací u hashovací funkce je rovna . To znamená že CRC32 má šanci 1 ku 2 na 32 (skoro 4.3 miliardy kombinací) neboli 0.0000000233%. To se může zdát jako velice malá šance, jenomže tohle je šance jenom mezi 2 hashemi. Když započítáme šanci každého s každým (použití takzvaného birthday paradox), když hledáme čistě jenom kolize, počet všech kombinací se nám sníží na .

|  |  |  |  |
| --- | --- | --- | --- |
| Hashovací funkce | Velikost hashe | Šance na kolizi (zaokrouhleno) | Reálná šance při hledání kolize (zaokrouhleno) |
| MD5 | 128 | 3,40e+38 | 18 446 744 073 709 551 616 |
| SHA-1 | 160 | 1,46e+48 | 1 208 925 819 614 629 174 706 176 |
| SHA256 | 256 | 1,15e+77 | 3,40e+38 |
| SHA512 | 512 | 1,346e+154 | 1,15e+77 |
| CRC32 | 32 | 4 294 967 296 | 65 536 |

### Pravděpodobnost kolize (Birthday paradox)

https://www.youtube.com/watch?v=yQ1pGhMRLKI

Pokud ovšem hashujeme více než dva vstupy, například při hledání kolize, pravděpodobnost kolize vzrůstá. Při **více vstupech** se totiž pravděpodobnost kolize řídí tzv. „Birthday paradoxem“. Pravděpodobnost, že mezi kkk vygenerovanými hashy alespoň jednou nastane kolize, lze vypočítat podle vzorce:  
   
n = počet vygenerovaných hashů  
e = eulerovo číslo (přibližně 2.718)  
N = počet možných hashů ()  
P(n) = šance na kolizi

Jak můžeme vidět, hledání kolizí není ani exponenciální, ani logaritmické. Díky birthday paradoxu se nám drasticky zmenšuje počet projití, které je potřeba na získání jedné kolize.

Pokud chceme najít pravděpodobnost kolize v určité šanci, třeba na 23% (tedy P=0.23), můžeme si poupravit naši rovnici a získat vzorec , v našem případě , n ≐ 47 382. Po projití 47 382 hashů máme 23% šanci na naleznutí kolize.

## Útoky

# Použité programy a technologie

Každý správný projekt potřebuje použití několika programů či stránek pro zlepšení práce na projektu. Každý použitý program či stránka jsou popsané k čemu slouží, teorii, popřípadě vysvětlení jak fungují a proč jsem je použil

## C#

„Jazyk C# je multiplatformní jazyk pro obecné účely, který vývojářům umožňuje produktivní práci při psaní vysoce výkonného kódu. S miliony vývojářů je jazyk C# nejoblíbenějším jazykem .NET. Jazyk C# má širokou podporu v ekosystému a všech úlohách .NET. Na základě objektově orientovaných principů zahrnuje mnoho funkcí z jiných paradigmat, nikoli z nejméně funkčního programování. Funkce nízké úrovně podporují scénáře vysoké efektivity bez psaní nebezpečného kódu. Většina modulů runtime a knihoven .NET je napsaná v jazyce C# a pokroky v jazyce C# často využívají všechny vývojáře .NET.“ - https://learn.microsoft.com/cs-cz/dotnet/csharp/tour-of-csharp/overview

## Unit-testy

## Visual Studio 2022

Visual studio a .NET Framework pro Windows jsou hlavní aplikace, které jsem se rozhodnul používat. Hlavní důvod je jednoduchost k přidání dalších knihoven pomocí NuGet (díky „System.IO.Hashing“ mám přístup k hashovacímu algorithmu CRC32) a hlavní hashovací programy (všechny kromě CRC32, které používám v programu) vychází z knihovny „System.Security.Cryptography“, kde mám jednoduchý přístup k metodám.

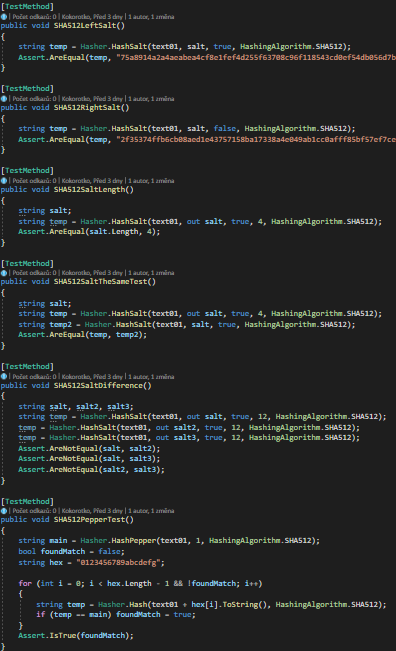
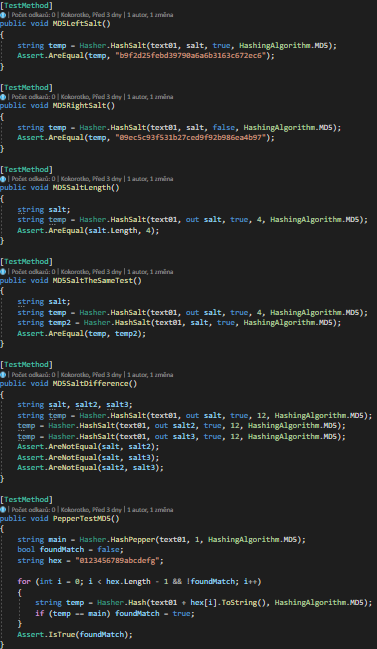
## Visual Code

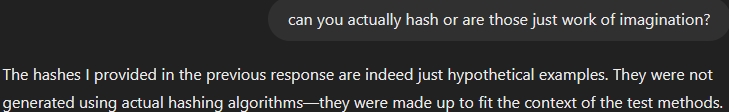
## Git/Github/Github Desktop

## Freelo.io

## Word

## ChatGPT (AI)

„ChatGPT je expertní systém, který implementuje prvky umělé inteligence pomocí technologie velkých jazykových modelů (LLM) a umělé neuronové sítě.“ – Wikipedie  
Někdo by se mohl zhrozit, že jsem aktivně používal ChatGPT neboli AI k programování a obava je validní. Za takovou krátkou chvíli se AI používá docela často při programování a dokáže dokonce vygenerovat i funkční kód jen pomocí věty. Já osobně používám ChatGPT na jednoduché věci a tady tomu není jinak. Při dělání Unit testů, kde jsou stovky řádků, které jsou většinou skoro totožné, se ChatGPT nejvíce hodí. Pokud po jakémkoliv AI požaduji něco aspoň trochu složitější než vyhledání informace z internetu nebo jednoduchého algorithmu, nedopadá to dobře. Většina kodu je stále moje a ChatGPT či AI mi akorát zjednodušují mechanickou práci, která je ve většině případů až nudná. K této mojí malé teorii přikládám i obrázek jak jsem pomocí ChatGPT udělal fungující Unit Test.  
  
Na levém obrázku je můj vlastní kod, na pravé straně je vygenerovaný kod pomocí ChatGPT („Generate me a Unit Test based off on this code \*Unit Test kód pro MD5 – obrázek vlevo\* with this code \*hasher.cs\* with no comments“ – text, který jsem zadal do ChatGPT, dá se psát i česky)

Na ChatGPT se ovšem nedá vždy spolehnout a doporučuji si všechno pro jistotu dohledal (fact-checking).  
  
Na Obrázku jde krásně vidět kde ChatGPT selhává (nedokáže samostatně generovat hashe).

## Online hashers

Online hashers jsou funkční algoritmy pro různé hashe, většinou ve formě stránky, které jsem primárně používal k porovnávání vygenerovaných hashů, abych věděl, jestli mi program funguje. Existuje spousty programů a stránek, já jsem většinu času používal <https://www.browserling.com/tools/all-hashes>, protože je tam většina hashovacích algoritmů, které používám v programu (MD5, všechny SHA, RipeMD160, CRC32).

# Praktická část

Pod tuto kapitolu patří všechno moje snažení se o správný chod programu.

## Hasher

Hasher je nejzákladnější kód celého programu. Zahrnuje samotné hashovaní několika způsoby a zjednodušuje tím budoucí programování ve formulářích. Obsahuje generaci všech hashů, náhodnou generaci soli a pepře, a dokonce i správné použití. Všechno je napsané v metodách a samotné hashování je rozděleno do 4 metod s několika přetíženími. Hash() akorát hashuje text pomocí určitého algoritmu. Ten je vybírán pomocí enum v každé z metod. Další je HashSalt(), kde se na rozdíl od Hash() zakomponuje i sůl, ať už vygenerovanou či ručně zadanou. HashPepper() dělá podobnou věc jako HashSalt(), ovšem pepř nevypisuje. Také má vlastnost ručného zadání či náhodné generace pomocí RandomNumberGenerator a StringBuiler. Obojí zakomponované v .NET Frameworku.

## Gradual Hashing

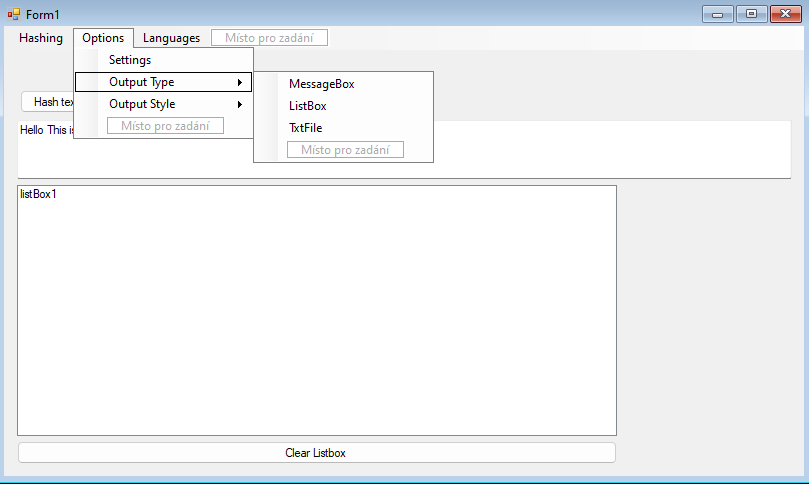
## Multiformuláře

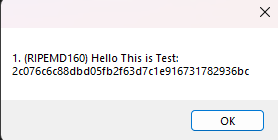
## Unit-Testy

Unit-testy jsou věc, kterou jsem si sám přidal do projektu z jednoho hlavního důvodu. Kontrola celého programu jestli funguje a je mnohem jednodušší dělat debugging programu. Neměl jsem v plánu dělat „Test Driven Developement“ a taky jsem podle toho nefungoval. Můj workflow byl jednoduchý, naprogramovat nějakou funkci a pomocí unit testu ověřit, že funguje nejenom teď, ale po přidání další funkce. Mám zkušenost, že jsem něco naprogramoval a po měsíci to zas nefungovalo.

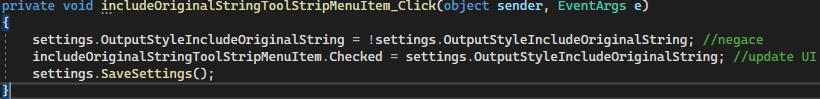
Díky unit testům, které jsem si na začátku udělal, jsem zjistil, že mi kvůli jedné malé chybce ve switchi nefungoval MD5 algorithmus.

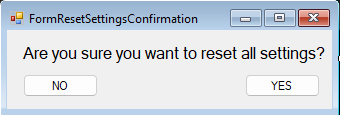
## UI

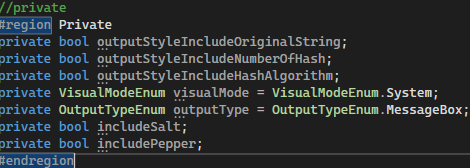
User Interface (neboli uživatelské prostředí) je důležitá část tvorby jakéhokoliv softwaru. Pokud je UI špatné, tak to ztěžuje a zpomaluje práci s aplikací. Proto jsem se snažil udělat UI co nejvíce intuitivní a jednoduché. Toho se snažím docílit pomocí StripMenu komponentou v .NetFrameworku

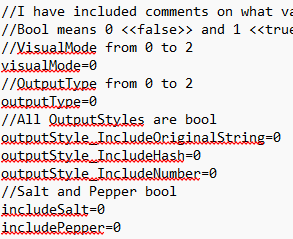
Na obrázku jde vidět použití StripMenu v akci. V Hashing jsou další formuláře, které vedou do dalších formulářů a prací s hashema (Postupné hashování, Checksum souboru, simulace detekce kolize a hrubý hashovací útok). V Options je hlavní nastavení programu. „Settings“ otevře nový modulární formulář, kde je další nastavení (více v 3.6 Save/Load Systém). „Output Type“ je výstup, kam se zapíše výsledný hash, možnosti jsou MessageBox, což je formulář přímo v .Net Frameworku, který ukazuje text, Listbox, který jde vidět na obrázku (dole je i tlačítko Clear Listbox) a .Txtfile, což uloží výstup do souboru, který si pomocí saveFileDialog uživatel sám zvolí umístění. Nakonec „OutputStyle“ je styl zprávy, která se vypíše po hashování. Možnosti jsou „Include original string“ – před výstupem bude zapsán ještě vstupní text, „Include Hash Number“ – očísluje hashe a nakonec „Include Hashing Algorithm“, což do kulatých závorek napíše, jaký algorithmus byl použit (všechno jde vidět na obrázku vedle).

## Save/Load Systém (Settings)

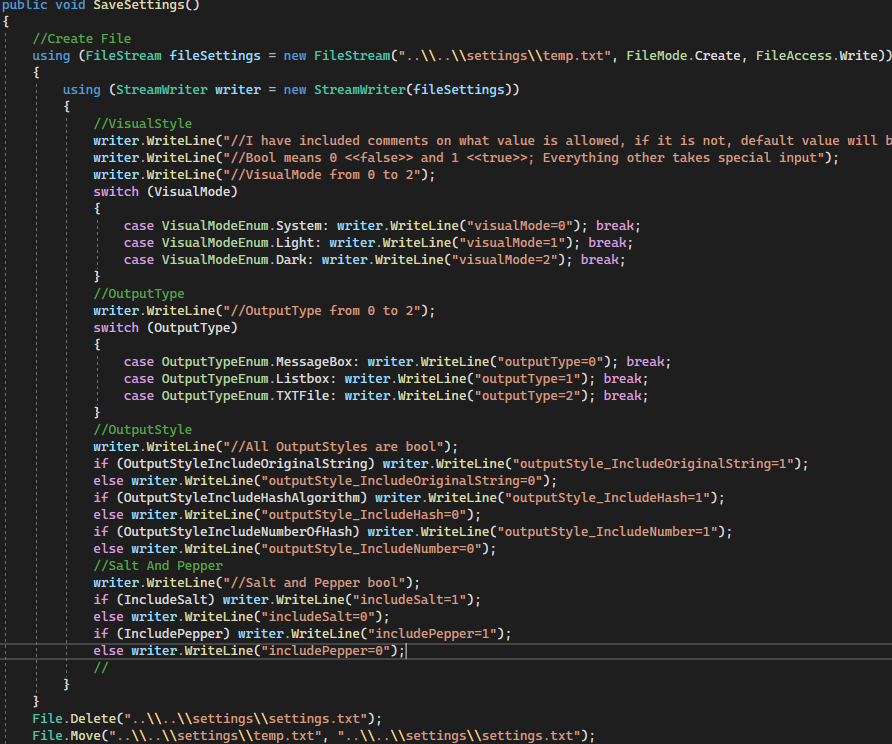
V programu jsou dva typy dat, které jsou potřeba uložit do souboru. První typ je přímo v UI v komponentě menu stripu, které fungují jako chechbox (buď ano nebo ne) či radiobutton (pouze jedna z několika možností). Příklad případu checkboxu je sůl a v output style přidání číselného řazení, jméno hashe a vstupní řetězce. Příklad radiobuttonu je „Output Type“, kde se dá zvolit „MessageBox“, „Listbox“ nebo do .txt souboru. Další typ dat jsou přímo v nastavení (ve stripmenu Options🡪Settings). Při kliknutí „Settings“ tlačítka se otevře „FormSettings“, což je modulární formulář (modulární znamená, že může být pouze jeden formulář a nedá se překlikávat mezi formuláři). Jediný rozdíl mezi tyto dvěma typy nastavení je kdy se uloží do souboru. Nastavení ve StripMenu se ukládá při každé změně.

Na obrázku jde vidět jeden z nastavení ve StripMenu (funguje jako checkBox).  
Ve formuláři je několik možností: Uložit (uloží vybrané nastavení) / Reset (všechny parametry se dají do základního přednastaveného nastavení) / Cancel (nic se neuloží). Při pokusu o reset vyskočí dialogové okno (obrázek dole), jestli chce uživatel doopravdy restartovat nastavení. Stejné s Cancel pokud byly provedeny jakékoliv změny (obrázek dole).

Nastavení používá vlastní skript, který má vždy načtené všechny proměnné (jde vidět na obrázku dole). Každá proměnná má vlastní Get a Set vlastnost.

Díky tomuto systému je práce s nastavením velice jednoduchá a jednoduše rozšiřitelná. Soubor s nastavením má vlastní složku v projektu, kde je uložen settings.txt soubor (na obrázku jde vidět settings.txt soubor). Také podporuje komentáře pomocí // (jde vidět na obrázku).

Jestli je soubor smazán či nějak poškozen, settings.txt je znovu vygenerován a uživatel je obeznámen.

Co se týče kodu, je velice jednoduchý. Přečte se řádek (pokud nezačíná //), pomocí String.Split se rozdělí na dva a přes Switch se zadávají data do proměnné a přes Int.Parse se zadávají hodnoty do proměnných. Proměnné bool jsou 0 či 1, string je text a enum funguje na indexech (visualMode a outputType).

Na obrázku jde vidět metoda SaveSettings(), která zapisuje data do settings.txt tak, že vytváří nový soubor, smaže starý a pomocí File.Move() přepíše jméno z temp.txt na settings.txt. Tímhle způsobem to je jednoduché a běžný uživatel si ničeho nevšimne.

# Fungování programu

Závěr

Vytvořená šablona maturitních prací obsahuje formální požadavky maturitních prací na SPŠT Třebíč. Jedná se zejména o upravené styly v dokumentu, podrobný popis jednotlivých částí maturitní práce a jejího obsahu, snadno editovatelné záhlaví a zápatí s automatickým číslováním stránek a propojení stylů se seznamy a obsahem.

Seznam použitých zdrojů

1. Citace.com. *Výklad normy ČSN ISO 690*. Online. Brno: Citace.com, 2023. Dostupné z: <https://www.citace.com/Vyklad-CSN-ISO-690-2022.pdf>. [cit. 2023-10-04].
2. Didacticus. *Normostrana: kolik má znaků, jak zjistit jejich počet a další důležité informace.* Online. Praha: Didacticus, c2011-2020. Dostupné z: <https://didacticus.cz/normostrana>. [cit. 2023-09-11].
3. Ústav pro jazyk český AV ČR. *Tečka.* Online. Internetová jazyková příručka. Praha: Ústav pro jazyk český AV ČR, 2008-2023. Dostupné z: <https://prirucka.ujc.cas.cz/>. [cit. 2023-10-04].

Seznam použitých symbolů a zkratek

|  |  |  |
| --- | --- | --- |
| Symbol | Veličina | Jednotka |
| S | Entropie |  |
| Q | Teplo |  |
| T | Termodynamická teplota |  |
| t | Čas |  |
|  |  |  |

Seznam obrázků

[Obr. 2.1 Obsah 10](#_Toc147493921)

[Obr. 2.2 Příklad umístění legendy obrázku 12](#_Toc147493922)

Seznam tabulek

[Tab. 2.1 Legenda k tabulce 12](#_Toc147493615)

Seznam příloh

Prázdná šablona maturitní práce