Ekonomická fakulta
Faculty of Economics

Jihočeská univerzita v Českých Budějovicích
University of South Bohemia in České Budějovice

Jihočeská univerzita v Českých Budějovicích

Ekonomická fakulta

Katedra aplikované matematiky a informatiky

Bakalářská práce

Vývoj desktopové aplikace pro výpočet objemu kulatiny

Vypracoval: David Smetana

Vedoucí práce: Mgr. Radim Remeš

České Budějovice 2021

JIHOČESKÁ UNIVERZITA V ČESKÝCH BUDĚJOVICÍCH

Ekonomická fakulta

Akademický rok: 2018/2019

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**David SMETANA**
Osobní číslo:	**E17452**
Studijní program:	**B6209 Systémové inženýrství a informatika**
Studijní obor:	**Ekonomická informatika**
Téma práce:	**Vývoj desktopové aplikace pro výpočet objemu kulatiny**
Zadávající katedra:	**Katedra aplikované matematiky a informatiky**

Zásady pro vypracování

Cílem bakalářské je vytvořit aplikaci pro výpočet objemu kulatiny na základě zadaných hodnot. Výpočty objemu budou provedeny na základě způsobu měření a typu dřeviny. Data budou zaznamenávána do databáze, ze které je bude možné exportovat do základních běžných formátů tabulkových procesorů (např. MS Excel, OpenOffice, LibreOffice, apod.).

Metodický postup:

1. Studium odborné literatury.
2. Návrh a popis vývoje a implementace výsledné aplikace.
3. Zhodnocení použitelnosti aplikace pro nasazení v reálném prostředí.
4. Závěry a doporučení.

Rozsah pracovní zprávy:	**40 – 50 stran**
Rozsah grafických prací:	**dle potřeby**
Forma zpracování bakalářské práce:	**tištěná**

Seznam doporučené literatury:

1. Albahari, J. & Albahari, B. (2017). *C# 7.0 in a Nutshell.* Sebastopol, California (USA): O'Reilly Media.
2. Harrington, J. L. (2016). *Relational Database Design and Implementation.* Cambridge, MA, USA: Morgan Kaufmann.
3. Nagel, C. (2016). *Professional C# 6 and .NET Core 1.0.* Indianapolis, Indiana (USA): John Wiley & Sons.
4. Posadas, M. (2016). *Mastering C# and .NET Framework.* Birmingham, UK: Packt.
5. Price, M. J. (2017). *C# 7.1 and .NET Core 2.0: Modern Cross-Platform Development.* Birmingham, UK: Packt Publishing.

Vedoucí bakalářské práce: **Mgr. Radim Remeš**
Katedra aplikované matematiky a informatiky

Datum zadání bakalářské práce: **15. ledna 2019**
Termín odevzdání bakalářské práce: **12. dubna 2020**

V Českých Budějovicích dne 4. března 2019

L.S.

JIHOČESKÁ UNIVERZITA
V ČESKÝCH BUDĚJOVICÍCH
EKONOMICKÁ FAKULTA
Studentská 13 (26)
370 05 České Budějovice

doc. Dr. Ing. Dagmar Škodová Parmová
děkanka

doc. RNDr. Jana Klicnarová, Ph.D.
vedoucí katedry

Prohlášení

Prohlašuji, že svoji bakalářskou práci na téma „Vývoj desktopové aplikace pro výpočet objemu kulatiny“ jsem vypracoval samostatně pouze s použitím pramenů a literatury uvedených v seznamu citované literatury. Prohlašuji, že v souladu s § 47b zákona č. 111/1998 Sb. v platném znění souhlasím se zveřejněním své bakalářské práce, a to – v nezkrácené podobě/v úpravě vzniklé vypuštěním vyznačených částí archivovaných Ekonomickou fakultou – elektronickou cestou ve veřejně přístupné části databáze STAG provozované Jihočeskou univerzitou v Českých Budějovicích na jejích internetových stránkách, a to se zachováním mého autorského práva k odevzdanému textu této kvalifikační práce. Souhlasím dále s tím, aby toutéž elektronickou cestou byly v souladu s uvedeným ustanovením zákona č. 111/1998 Sb. zveřejněny posudky školitele a oponentů práce i záznam o průběhu a výsledku obhajoby kvalifikační práce. Rovněž souhlasím s porovnáním textu mé kvalifikační práce s databází kvalifikačních prací Theses.cz provozovanou Národním registrem vysokoškolských kvalifikačních prací a systémem na odhalování plagiátů.

---------- ----------

Datum Podpis studenta

Poděkování

Rád bych zde poděkoval Mgr. Radimu Remešovi, vedoucímu této bakalářské práce, za trpělivost, poskytnutí cenných rad a podnětných připomínek při vypracování díla. Dále bych chtěl poděkovat mé rodině a přátelům, kteří mě podporovali po celou dobu studia.

Obsah

1 Úvod

Vývoj desktopových aplikací zažívá v posledních letech značné změny. Vzestup mobilních zařízení a dominance cloudových aplikací oprávněně zastiňuje vývoj desktopových aplikací. Příkladem mohou být aplikace pro výpočet objemu kulatiny, které jsou z drtivé většiny na webovém prostředí. Nastává tedy otázka, zdali je stále relevantní vyvíjet aplikace na desktopové systémy. Technologické firmy reagují tvorbou platforem použitelných na široké škále zařízení. Díky těmto technologiím lze publikovat desktopové aplikace s téměř nulovými úpravami logické vrstvy na mobilní zařízení či webové rozhraní.

Záměrem této práce je seznámit čtenáře s vývojem aplikací určených pro desktopové počítače, specificky pro operační systém Windows a s pomocí vývojové platformy .NET. Funkcionality aplikace jsou zaměřeny na výpočet objemu kulatiny na základě zadaných dat a zvolených parametrů. Aplikace dále umožní export dat do tabulkových procesorů. Data zadána uživatelem jsou ukládána do databáze, která umožní uživateli zobrazovat záznamy dle požadovaných atributů.

1.1 Cíl práce

Cílem této práce je seznámit čtenáře s platformou .NET a přidruženými technologiemi vhodnými pro vývoj desktopové aplikace. Práce se též zabývá popisem databázových systémů, kde jsou vysvětleny rozdíly mezi datovými modely a je nastíněna problematika stále populárnějšího databázového designu NoSQL. Dále je definována aplikace a jsou popsány základní typy aplikací objevující se na trhu. Praktická část práce je zaměřena na návrh aplikace, který je doplněn o diagramy, a také následný vývoj, jenž je vyobrazen formou popisu uživatelského prostředí a částmi zdrojového kódu.

2 Teoretická část

2.1 Aplikace

Aplikace je zkrácený výraz pro aplikační software, který označuje programy či soubory programů poskytující uživateli služby dle svého zaměření. Aplikační software je navržen pro interakci s uživatelem a na základě zadaných požadavků provádí dané úkony. Službou se rozumí vykonání specifické činnosti dle pokynu uživatele a zaměření aplikace. Aplikační software lze rozdělit dle zaměření na základní skupiny: textové, prezentační a grafické editory, podnikový software, webové prohlížeče, databázové aplikace atd. Aplikační software můžeme rozdělit dle typu zařízení, pro které jsou určeny. Základními kategoriemi jsou desktopové, mobilní a cloudové aplikace.

2.1.1 Desktopová aplikace

Desktopová aplikace je software, který je možno nainstalovat a spustit na stolním počítači nebo notebooku a vykonávat specifické úlohy. K tomu využívá výkon hardwaru počítače a operační systém, včetně souborů a dat uživatele („Definition of Desktop Application", b.r.). Při vývoji aplikace zaměřené na desktop je nutné zvážit možnost volby operačního systému. Nejrozšířenějšími operačními systémy pro počítače a notebooky jsou: Microsoft Windows, macOS a Linux. Každý z těchto systémů používá jiné API[1]. Pokud je požadavkem podpora více operačních systémů, vývojáři jsou nuceni provést portaci aplikace, ba dokonce přeprogramovat části kódu.

2.1.2 Mobilní aplikace

Mobilní aplikace je software určený pro mobilní zařízení, který je specifický nízkými pořizovacími náklady, snadnou instalací, intuitivním ovládáním a zaměřením na širokou škálu mobilních zařízení. Trh s mobilními aplikacemi neustále roste díky stále rostoucí popularitě vývojářů mobilních aplikací. Mobilní aplikace jsou vyvíjeny a provozovány na různých platformách, jako jsou Android, iOS, KaiOS, Windows. Dle NetMarketShare tržní podíl zmíněných platforem přesahuje 58 %.

Mobilní aplikace jsou zpravidla distribuovány na zařízení pomocí platforem, jenž poskytují snadný a bezpečný způsob pořízení a spravování aplikací. Vlastníci těchto platforem nabízí zdarma sady vývojových nástrojů, případně i vývojové prostředí.

[1] Aplikační rozhraní

2.1.3 Cloudové aplikace

Cloudová aplikace je software založený na spolupráci koncového zařízení uživatele a cloudových zařízení. Veškeré logické operace jsou zpracovávány na vzdálených serverech, ke kterým je uživatel neustále připojen pomocí internetu přes webový prohlížeč. Cloudové servery jsou umístěny v datových centrech. Tato centra jsou obvykle spravována třetími stranami a poskytují cloudové služby dle distribučního modelu. Distribuční modely se od sebe odlišují nabízenou službou. Rozlišují se tři základní služby: pronájem infrastruktury, platformy, či softwaru.

Při pronájmu infrastruktury, často pod zkratkou IaaS, nabízí pronajímatel kompletní hardwarové vybavení – servery a úložiště. Nejpodstatnější předností této služby je okamžitá dostupnost pronajímaných služeb. Zákazníci nemají náklady spojené s pořízením a provozem serverů, nemusejí se zabývat síťovým zabezpečením a je zajištěna škálovatelnost. Pronajatá infrastruktura může sloužit prakticky k čemukoliv, např. provozování platforem, webové aplikace, práce s velkými daty, zálohování dat apod. Nejznámějšími poskytovateli infrastrukturních služeb jsou: Microsoft Azure, Amazon Web Service, Google Cloud Platform, IBM Cloud a Orancle Cloud Infrastructure.

Platforma poskytovaná formou služby (PaaS) je další forma poskytování cloudových služeb. Nabízí veškeré prostředky předchozího distribučního modelu a přidává vývojářské nástroje pro aplikace a databáze v cloudu. Často se jedná o aplikace vytvořené dle specifických požadavků zákazníka. Platforma nabízí nejen vývoj, ale i následný provoz aplikací v délce celého životního cyklu.

Software jako služba (SaaS) je poslední formou poskytování cloudových služeb. Jedná se o provoz a správu aplikace, kdy zákazník platí za používání aplikace a poskytovatel danou aplikaci společně s potřebnými hardwarovými nároky spravuje. Služba tedy poskytuje kompletní hardwarové i softwarové řešení. Uživatelé se pouze připojují přes internet.

Cloudové aplikace jsou nejčastěji emailové služby, ukládání a sdílení dat, firemní systémy, kancelářské balíky apod. Hlavními výhodami cloudových aplikací jsou dostupnost, škálovatelnost, spolehlivost a nízké nároky na koncového uživatele. Nevýhodou je nutnost neustálého připojení k internetu či pořizovací náklady.

2.2 Operační systém

Operační systém lze definovat jako základní softwarové vybavení počítače, které je nainstalované do paměti zařízení a zůstává v chodu od zapnutí zařízení až po jeho vypnutí či odpojení od zdroje napájení. Jeho účelem je umožnit uživateli ovládání počítače, vytváření a přidělování systémových zdrojů pro aplikační prostředí (Tanenbaum, 2015).

Operační systém se typicky tvořen jádrem (kernel) a pomocnými systémovými nástroji. Hlavními činnostmi jádra je správa paměti, procesů, zařízení a systémová volání. Jinými slovy má přístup k veškerému fyzickému vybavení počítače a vykonává jakékoliv operace, jichž je dané zařízení schopné. Na operační systém navazuje aplikační vrstva v podobě GUI[2] nebo shell[3], která umožňuje uživatelům instalaci a spuštění ostatních programů.

V dnešní době existuje pestré množství operačních systémů se zaměřením na specifická zařízení. Mezi tyto zařízení lze zařadit: mobilní telefony, notebooky, servery, televize, herní konzole, tablety atd. Nejrozšířenějšími operačními systémy jsou: Windows, macOS, Linux, Android a iOS, z nichž za desktopové lze považovat první tři zmíněné. V tabulce 1 jsou vyobrazeny tržní podíly zmíněných operačních systémů.

Tabulka 1 Tržní podíl operačních systémů za období 2020

Operační systém	**Podíl na trhu v %**
Android	42,12
Windows	35,79
iOS	16,71
macOS	3,90
Linux	0,99
Ostatní	0,49

Zdroj: („NetMarketShare", 2020), upraveno autorem

2.2.1 Microsoft Windows

Windows je souhrnné označení pro operační systémy od firmy Microsoft. Jedná se o operační systém pro stolní počítače a notebooky, ale můžeme ho nalézt i na mobilech, tabletech, konzolích a serverech, pro které jsou vyvíjeny speciální verze. Označení Windows se poprvé objevuje roku 1983, kdy Microsoft oznámil vývoj

[2] Grafické uživatelské prostředí
[3] Interpret příkazů

operačního systému s grafickým rozhraním a v roce 1985 již byl v prodeji pod označením Windows 1.0. Nejnovější dostupná verze je Windows 10 (Bellis, 2019).

Operační systém Windows je jedním z nejznámějších operačních systémů na světě a nejběžnějším desktopovým operačním systémem s tržním podílem přes 80 % („NetMarketShare", 2020). Svůj tržní podíl si udržuje díky rozmanitosti podporovaných zařízení a prodeji OEM licencí[4]. Microsoft vydal již mnoho verzí Windows. Kromě aktuální verze jsou stále hojně používány i verze předchozí. Graf 1Tržní podíl desktopových operačních systémů vyobrazuje tržní podíl nejpoužívanějších verzí Windows z prosince 2020. Za zmínku stojí verze Windows 7 z roku 2009, která stále tvoří signifikantní část celkového podílu Windows i přes jeho stáří. Lze s jistotou konstatovat, že daná hodnota bude klesat. Microsoft totiž ukončil dne 14.1.2020 rozšířenou podporu zajišťující bezpečnostní aktualizace („Produkty ukončující podporu", 2020).

Vývoj desktopových aplikací pro Windows usnadňuje několik platforem, které jsou poskytovány Microsoftem a jejich hlavním záměrem je usnadnění tvorby aplikací pro klasickou plochu Windows a současně poskytnutí specifické funkce. Níže uvedené platformy často sdílejí určité rysy, ale každá je zaměřena na konkrétní typ aplikace.

Universal Windows Platform (UWP)

UWP je jedna z nejmladších vývojových platforem pro Windows. Uživatelské prostředí je tvořeno XAML značkovacím jazykem. Dále platforma podporuje systém *Fluent Design*, který umožní snadné použití vstupních zařízení (myši, klávesnice, dotykové plochy, herní ovladače apod.), dodá aplikaci responzivitu a podporu široké škály zařízení (Schofield, Toliver, Parente, & Hickey, 2020). Zásadní výhodou je jednotný kód pro všechna zařízení podporující tuto platformu.

Windows Presentation Foundation (WPF)

Vývojová platforma pro tvorbu formulářových aplikací s využitím XAML pro oddělení grafického prostředí od aplikační logiky. WPF je platforma navržena pro vývoj desktopových aplikací s důrazem na rozmanité grafické prostředí. WPF je součástí .NET platformy a lze ji označit za nástupce Windows Forms.

[4] Softwarová licence vázaná k danému zařízení

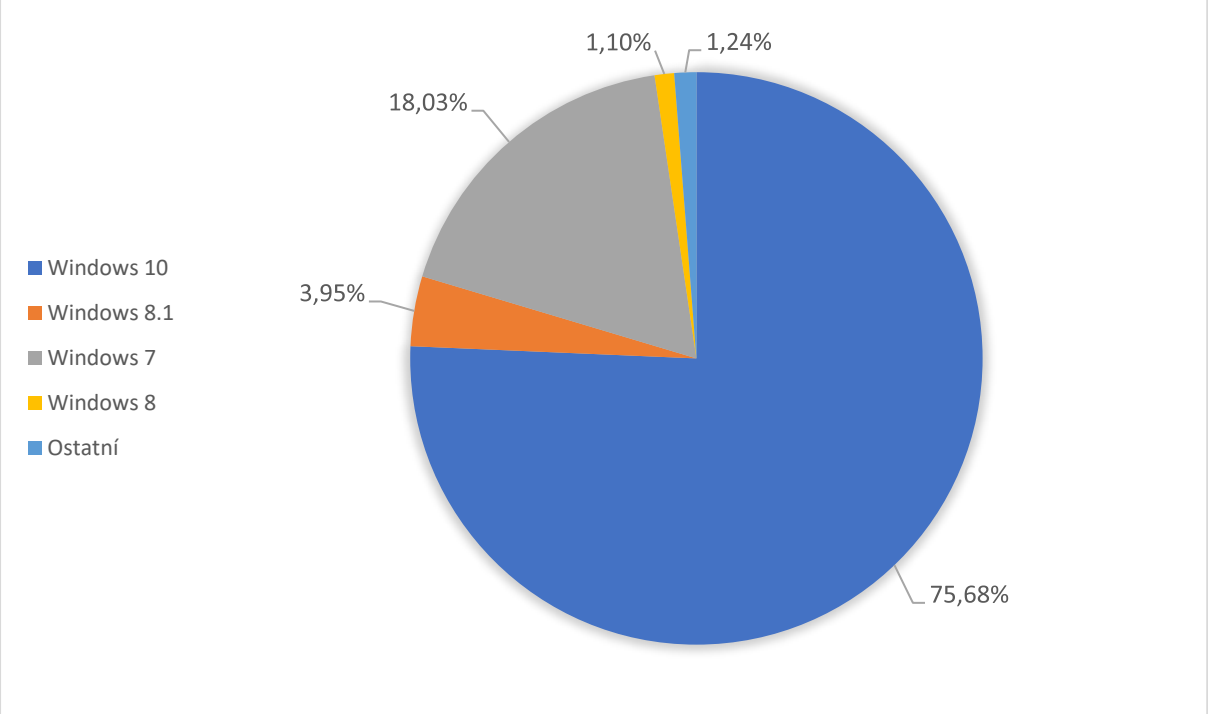
Windows Forms

Původní platforma pro tvorbu aplikací s jednoduchým modelováním uživatelského rozhraní. Hlavní výhodou je vestavěná sbírka vizuálních ovládacích prvků. Na rozdíl od WPF nepoužívá XAML, tudíž při převádění aplikace na jinou platformu je nutné kompletně předělat uživatelské rozhraní. Obdobně jako předchozí platforma je Windows Forms součástí .NET.

Win32

Rozhraní pro vývoj aplikací určených k vývoji C/C++ nativních aplikací pro Windows, které vyžadují přímý přístup k systému a hardwaru (Blevins, Wojciakowski, & Graham, 2021).

Graf 1Tržní podíl desktopových operačních systémů



Zdroj: („Desktop Windows Version Market Share Worldwide", 2020), upraveno autorem

2.2.2 macOS

macOS je souhrnné označení operačních systémů od firmy Apple. Tyto operační systémy jsou výhradně určeny pro počítače a notebooky, které vyrábí již zmíněná firma. Počátky systému sahají do roku 1999, kdy se objevil na trhu pod názvem Mac OS 9 (později Mac OS X) a nahradil svého předchůdce s označením Mac OS (Tanenbaum, 2015). Aktuální verze je macOS 10.15 s kódovým označením Catalina.

Na rozdíl od Windows je macOS striktně dodáván pouze se zařízeními Apple. Firma neposkytuje licence OEM výrobcům, a tedy díky omezenému počtu možných hardwarových specifikací je operační systém na daných zařízeních velmi stabilní a s minimálním výskytem chyb.

Apple nabízí integrované vývojové prostředí Xcode obsahující veškeré potřebné nástroje pro vývoj aplikací specificky zaměřených na operační systémy macOS, iOS, watchOS a tvOS. Xcode podporuje značnou rozmanitost programovacích jazyků, jako jsou Swift, C, C++, JavaScript, Ruby, AppleScript, Java a Objective-C (Varma, 2015). Díky doplňkům třetích stran je ovšem možné přidat podporu pro ostatní jazyky. Aktuální verze Xcode je označena číslem 12 a obsahuje vývojové sady pro aktuální verze již zmíněných operačních systémů.

2.2.3 Linux

Linux je open-source operační systém založený na stejnojmenném jádře, které slouží jako základ pro mnohé operační systémy. Systém Linux se poprvé objevuje roku 1991, kdy byla vydána první verze systému, jehož autorem je Linus Torvalds (Vaughan-Nichols, 2011). V dnešní době se Linux nejčastěji objevuje na serverech a výkonných výpočetních sestavách. Tržní podíl desktopové verze Linuxu se odhaduje dle NetMarketShare (2020) na necelé procento.

Uživatelé mají na výběr z mnoha distribucí Linuxu, které jsou nejčastěji poskytovány zdarma. Mezi nejznámější patří Debian, Ubuntu a Fedora. Licence open-source umožňuje distribuci pro komerční i nekomerční účely. Na vývoji Linuxu se podílí mnoho programátorů a firem. Často je využíván společnostmi a institucemi jako levnější alternativa vůči Windows.

2.3 .NET

.NET je open-source vývojové prostředí od firmy Microsoft, které slouží k vytváření nejrůznějších druhů aplikací a služeb. Obsahuje programovací jazyky C#, Visual Basic, F# a platformy podporující .NET Standard a Mono.

2.3.1 .NET Framework

.NET Framework je vývojová platforma obsahující modul CLR, který zajišťuje spuštění kódu, a BCL (Base Class Library) se značným množstvím knihoven tříd pro vývoj aplikací. Microsoft původně zamýšlel využít .NET napříč platformami, ale

nakonec, se snahou maximalizovat funkčnost, soustředil své síly pouze na operační systém Windows (Price, 2019).

Od verze 4.5.2 je .NET Framework součástí operačního systému Windows a je nainstalován na více než jedné miliardě zařízení. S tímto obrovským počtem instalací je nezbytné provádět co nejmenší změny a předejít nechtěným problémům. I opravy chyb mohou způsobit vážné problémy, tudíž je aktualizován nepravidelně. Veškeré aplikace vyvinuté na platformě .NET Framework sdílí stejnou verzi CLR a knihoven tříd, která je uložená v GAC[5], což může vést k problémům, pokud aplikace vyžaduje specifickou verzi kvůli kompatibilitě (Price, 2019).

2.3.2 .NET Core

Vzestup mobilních zařízení, důraz na využití aplikací napříč platformami a příchod cloudového vývoje má za následek úpadek významu operačního systému Windows. Microsoft reagoval vytvořením nové open source platformy založené na .NET Frameworku se snahou oddělit vazby Windows a zároveň odstranit části, které již nejsou považované za klíčové. Nově vytvořená platforma byla pojmenována .NET Core.

Hlavní předností je multiplatformní implementace CLR modulu zvaná CoreCLR a knihovny tříd označované CoreFX (Price, 2019). Signifikantním rozdílem oproti svému předchůdci je vývoj a systém aktualizací. Platformu .NET Core lze použít ve více verzích na stejném zařízení. Aplikace tak mohou využívat starší verze a Microsoft může častěji vydávat aktualizace a záplaty bez obav o kompatibilitu aplikací.

.NET Core byl vydán v září 2019 ve verzi 3.0. Následující verze je přejmenována na .NET s číselným označením 5, aby nedošlo k záměně s .NET Framework 4.x (Hunter, 2018). Verze .NET 5.0 vyšla v listopadu 2020 společně s novou verzí programovacího jazyka C#. Microsoft dále plánuje každý rok vydávat novou verzi ve stejném období. Verze frameworků dělí dle podpory na LTS nebo aktuální.

LTS (Long-Term Support)

Verze s dlouhodobou podporou jsou považovány za stabilní a během jejich životního cyklu obdrží pouze několik aktualizací. Tyto verze jsou vhodné pro aplikace, které vývojáři nechtějí aktualizovat periodicky. U .NET Core je LTS v délce tří let. Aktuální LTS verze je .NET Core 3.1.

[5] Globální mezipaměť sestavení

Aktuální verze

Aktuální verze obsahují nejaktuálnější funkce a vylepšení, které mohou být na základě zpětné vazby změněny, tudíž je tato verze vhodná pro aktivně vyvíjející se aplikace. Pokud budou určité funkcionality změněny či odstraněny, vývojáři musí adekvátně zareagovat, aby se předešlo nechtěnému chování aplikace. Tyto „menší“ verze mají tříměsíční životnost po vydání nové aktuální verze. Toto období je určeno vývojářům k přizpůsobení kódu pro novou verzi.

2.3.3 Xamarin/Mono

Mono je projekt třetích stran, který se soustředil na vývoj nástrojů kompatibilních s prostředím .NET. Jednoduše lze říct, že se jedná o implementaci .NET Framework napříč platformami. Mono si našel uplatnění při vzniku platformy Xamarin zaměřené na mobilní zařízení, kdy byl využit jako její základ.

Xamarin vznikl v roce 2013 kdy byl založen vývojáři, kteří pracovali na projektu Mono. V roce 2016 byl odkoupen firmou Microsoft. Jedná se o open source platformu pro vývoj aplikací na operační systémy Windows, iOS a Android, mezi kterými lze sdílet až *90 %* kódu díky propracované podpoře a aplikačnímu rozhraní *Xamarin.Essentials* poskytujícímu jednotný přístup ke zdrojům (Johnson, 2020).

Xamarin poskytuje framework Xamarin.Forms k tvorbě jednotného uživatelského prostředí pro Windows, iOS a Android s jednotným zdrojovým kódem, tvořeným značkovacím jazykem XAML doplněným o programovací jazyk C# pro logiku.

Xamarin je nyní poskytován formou bezplatného rozšíření pro Visual Studio 2019.

2.3.4 .NET Standard

Vývojové prostředí .NET, spravované firmou Microsoft, nabízí v době tvorby této bakalářské práce sedm diferencovaných platforem. Tyto platformy jsou:

- .NET Framework
- .NET Core/.NET 5
- Xamarin
- Mono
- Universal Windows Apps
- Unity

Každá z těchto platforem byla tvořena pro jiné účely. Aby se předešlo nutnosti naučit se pravidla a omezení každé platformy, Microsoft definoval .NET Standard: formální specifikace aplikačního rozhraní .NET, které je možno implementovat na veškerých platformách .NET. Záměrem bylo vytvořit větší jednotnost v .NET ekosystému. Různé .NET implementace jsou zaměřeny na specifické verze .NET Standard, díky čemu je možné jednoznačně stanovit úroveň kompatibility.

Nejnovější verze je 2.1, která je implementována platformami .NET Core 3.0 a Xamarin a Mono. Platformy Unity a Universal Windows Platform zatím neobdržely podporu verze 2.1 a jsou tedy označeny zkratkou To Be Decided.

Microsoft nadále nebude vyvíjet nové verze .NET Standard, neboť postupem času bude nahrazen .NET 5. Důvodem jsou problémy spojené s vývojem cíleným na více platforem. Mezi hlavní problémy patří pomalé přidávání nových API a komplexní správa verzí. Tyto nedostatky jsou odstraněny v .NET 5, který zpětně podporuje .NET Standard 2.1 (Warren, 2020).

2.3.5 NuGet

NuGet je open source správce pro sdílení kódu skrz platformu .NET, jenž stanovuje pravidla a poskytuje potřebné nástroje pro tvorbu, sdílení a používání balíčků. Balíčky jsou sdíleny v souborech typu ZIP s příponou *.nupkg* a obsahují zkompilovaný kód *(DDLs*) společně s dalšími soubory, jenž zahrnují informace o daném balíčku (Douglas, 2019). Balíčky mohou být vytvořeny specificky pro určený framework, případně mohou podporovat .NET Standard pro zajištění maximální kompatibility. Kompatibilita zajišťuje plnou podporu daného frameworku v aktuální verzi.

Balíčky jsou distribuovány prostřednictvím webové stránky Nuget.org pro veřejnost, případně mohou být využity privátní NuGet servery a lokální úložiště pro sdílení balíčku v rámci týmu či organizace. Microsoft Visual Studio od verze 2017 nativně obsahuje správce balíčků NuGet. Uživatelé tedy mohou snadno vyhledávat, instalovat a spravovat balíčky, které lze získat z různých zdrojů.

2.3.6 Entity Framework Core

Entity Framework Core je multiplatformní technologie poskytující nástroje pro snadný přístup k datům v databázi díky objektově relačnímu mapování (ORM). EF Core je nástupcem Entity Framework, který měl značná omezení při multiplatformním využití. Poslední verze Entity Framework je 6.3 a je podporována .NET Core 3.0, ovšem již není nadále vyvíjen a lze ho považovat za *legacy* (Price, 2019).

EF Core podporuje jak tradiční Relational Database Management System (RDBMS), tak moderní cloudové NoSQL databáze jako je Microsoft Azure Cosmos DB a MongoDB, které Entity Framework nepodporuje (Price, 2019).

ORM využívá mapování k přičlenění sloupců a tabulek databáze vlastnostem a třídám, čímž vznikne model. Model je tvořen entitami tříd a kontextovým objektem poskytující přístup k databázi, a tedy umožňuje načítání a ukládání dat. Vývojáři tudíž mohou snadno pracovat s různými objekty jim známým způsobem a nemusí znát pravidla pro ukládání dat do tabulek relačních databází nebo do struktur typu NoSQL.

EF Core umožňuje dvěma způsoby vytvořit model:

- Database first
  Při přístupu Database first je model generován na základě existující databáze.
- Code first
  Při využití Code first přístupu je databáze vytvořena na základě již existujícího modelu.

Pro zajištění synchronizace mezi modelem a databází slouží migrace. Migrace reagují na veškeré změny provedené na modelu a inkrementálně aktualizují schéma databáze při zachování veškerých dat v databázi. K využití migrací jsou nutné zprostředkovávající nástroje *.NET Core CLI* nebo *EF Core Package Manager Console* (Lambson, 2018).

Entity Framework Core je distribuována skrz balíčky NuGet a podporuje širokou škálu poskytovatelů databází.

2.3.7 Programovací jazyk C#

C# je objektově orientovaný programovací jazyk vytvořený společností Microsoft pro účely vývoje na platformě .NET. Jeho základy vychází z dvou nejúspěšnějších programovacích jazyků z rodiny C: C a C++, a dále je velmi úzce spjat s programovacím jazykem Java (Schildt, 2006). Syntax C# je tedy značně podobný již zmíněným jazykům.

Programovací jazyk C# implementuje pravidla objektově orientovaného programování a nativně podporuje dědičnost, polymorfismus a zapouzdření. Charakteristickými rysy C# z objektově orientovaného hlediska jsou: unifikovaný typový systém, třídy a rozhraní, vlastnosti, metody a události, delegáty (Albahari & Johannsen, 2019). C# obsahuje unifikovaný typový systém *Common Type System*, ve kterém veškeré třídy implicitně dědí ze *System.Object*. Veškeré typy v .NET jsou buďto referenční

či hodnotové (De George, 2017). Hodnotové typy zastupují skutečnou hodnotu uloženou v zásobníku aplikace a při vytvoření instance proměnné je jí přirazena kopie aktuální hodnoty. Opakem jsou referenční typy, které pouze odkazují na skutečnou hodnotu do paměti, a proto při vytvoření instance proměnné odkazuje na skutečnou hodnotu a nevytváří kopii. *Common Type System* je sdílen skrze celou vývojovou platformu .NET a rozlišuje základních 5 kategorií typů: třídy, struktury, rozhraní, delegáty a výčty (De George, 2017). Jakýkoli definovaný typ obsahuje název, modifikátor dostupnosti, definované členy, případně implementovaná rozhraní a *base* typ. Typy mohou obsahovat členy specifikující chování a stav daného typu (De George, 2017). Mezi tyto členy patří: pole, vlastnosti, konstruktory, události, metody a vnořené typy.

Programovací jazyk C# se spoléhá na automatickou správu paměti. CLR obsahuje *garbage collector* (GC), který se automaticky spouští společně s aplikací a slouží k alokaci či uvolňování paměti (Warren, 2019). GC získává již nepotřebné objekty, uvolní jejich paměť a použije ji pro další alokace. Vývojáři tedy nemusí manuálně uvolňovat paměť a zároveň je zaručena vyšší bezpečnost paměti díky nemožnosti využití paměti jednoho objektu druhým objektem.

C# byl vydán v roce 2002 ve verzi 1.0. Aktuální verze je označená 9.0 a je podporována .NET 5.0 (Wagner, Kulikov, Yoshioka, & McMahon, 2020).

2.4 Databáze

Databáze je označení pro systém ukládání speciálně strukturovaných dat umožňující rychlé vyhledání, přístup a správu. Databáze jsou řízeny systémem pro správu a přístup zvaným *Database Management System* (DBMS), který překládá požadavky mezi uživateli a fyzickým úložištěm (Harrington, 2016). Uživatelé tak nepotřebují znát způsob ukládání dat na fyzické úložiště, ale pouze pracují s relacemi dat. DBMS poskytuje prostředky pro vytvoření struktury databáze, dle které jsou následně data ukládána. Dále musí umožnit správu databáze – uložení, úpravu či mazání dat a jejich získání. Tyto úpravy mohou být prováděny pomocí rozhraní založeném např. na formulářových sestavách, kde se každému uživateli zobrazí formulář a data jsou po vyplnění uložena. K získání dat pomocí relací je obecně používán jazyk *Structured Query Language* (SQL), jenž byl převzat za standard pro manipulaci s daty v relačních databázích (Harrington, 2016). Při dotazování jsou klíčové logické Booleovské operátory umožňující základní operace s daty. Logické operátory mají návratovou hodnotu true nebo false a lze je dále kombinovat. Speciální operátory následně dovolují tvorbu

komplexních dotazů, avšak návratová hodnota již není typu boolean, ale liší se dle použitého operátoru. Obecně se však jedná o seznam či sloupec hodnot.

2.4.1 Datové modely

Datový model je jedno z hlavních kritérií výběru vhodného systému pro správu databáze. Stanovuje logickou strukturu databáze a určuje, jak budou data spravována. Dle webu DB_Engines Ranking (2021) je drtivá většina komerčně využívaných databází založena na relačním modelu a jím příbuzných modelech, jako je např. objektově-relační. Relační model prezentuje databázi jako soubor tabulek, kde každou tabulku je možno uložit jako samostatný soubor. V posledních letech stoupá popularita NoSQL a *key-value* databází díky vhodným vlastnostem pro práci s velkými daty (anglicky Big data) (Elmasri & Navathe, 2016). Je také nezbytné zmínit hierarchické a síťové modely, jež jsou stále využívané některými staršími aplikacemi. Nutno poznamenat, že značná část systémů podporuje multi-model, tedy lze využít více databázových modelů při zachování jednotné datové vrstvy.

2.4.2 Entity a atributy

Entitu lze označit za základní prvek databázového modelu. Jedná se o určitý subjekt, o kterém se uchovávají informace. Tyto informace se nazývají atributy a jejich úkolem je charakterizovat danou entitu. Systémy pro správu databází obsahují entity ve formě tabulek s daným počtem atributů. Atribut lze chápat jako sloupec hodnot. Při uložení dat se vytváří nová *instance* entity a ukládají se pouze atributy (Harrington, 2016). Instanci lze tedy chápat jako řádek s počtem hodnot odpovídajícímu počtu sloupců. Aby nedošlo k záměně entit, je nutné použít unikátní identifikátor, jenž zajistí rozlišitelnost entit. Při výběru identifikátoru je nezbytné zvolit neměnnou hodnotu, neboť změny identifikátoru mohou způsobit nežádoucí problémy.

2.4.3 Primární klíč

Primární klíče jsou unikátní identifikátory v relačních databázích, které představují sloupec či kombinaci sloupců s jedinečnými identifikačními hodnotami (Harrington, 2016). Každá hodnota tedy zastupuje řádek s daty atributů. Primární klíče mohou být jedinečné pro každý řádek, případně se mohou opakovat. Při dotazování tak nemusíme vždy získat konkrétní hodnotu. Je důležité zvolit primární klíč tak, aby byla zajištěna jeho jedinečnost a hlavně neměnnost. Změna by mohla vyvolat nekonzistentnost mezi daty. Z těchto důvodů nemohou nabývat hodnoty null. Tato restrikce se nazývá integrita entity a je kontrolována DBMS při každém vložení či změně dat.

2.4.4 NoSQL

Označení NoSQL se vžilo pro systémy pracující s velkými daty. Nejedná se o datový model, nýbrž o postup při vytváření datového modelu. Hlavními přednostmi je rychlé načítaní dat, škálovatelnost a konzistentnost dat. Oproti relačním databázím, kde dochází k načtení jedné či více tabulek, je možné pomocí klíčů či indexů snadno a rychle vyhledat a získat přímý přístup k datům. Existuje mnoho nerelačních modelů, nejčastěji se však používají modely typu klíč-hodnota, dokumentové, grafové a sloupcově-orientované.

- **Databáze typu klíč-hodnota**

  Tyto systémy jsou založeny na jednoduchém modelu umožňující rychlý přístup k datům prostřednictvím odpovídajícího klíče. Neexistuje limitace pro typ dat, lze tedy databázi využít pro ukládání záznamů, dokumentů nebo webových stránek, ke kterým je přistupováno jako k celku pomocí jedinečného klíče. Využití pro tento databázový systém se naskytne v případě potřeby ukládání, a tedy i získávání jednotlivých záznamů jako jednotný celek. Hlavní předností tohoto typu databáze je velmi rychlé vyhledávání dat. Model však neposkytuje nástroje pro vyhledávání či úpravy dat v daném celku.

- **Dokumentové databáze**

  Databáze tohoto typu je podobná modelu klíč-hodnota, nicméně místo hodnot ukládá dokumenty reprezentující textová data. Tato data mohou být indexována, a tedy i vyhledávána. Model tedy obsahuje primární klíč, což lze považovat za jedinečný index pro každý záznam, avšak databáze umožňuje vytvoření sekundárních indexů, které mohou být tvořeny a mazány dle potřeb. Využití pro tento typ databáze je nejčastěji uváděno u webových katalogů, kde primární index např. rozděluje jednotlivá čísla a sekundární index může sloužit při vyhledávání klíčových slov či vět (Harrington, 2016).

- Sloupcově-orientované databáze

  Sloupcově-orientované systémy jsou značně podobné relačním databázím, neboť ukládají data do řádků a sloupců a k manipulacím s daty slouží příkazy obdobné SQL syntaxi. Na rozdíl od relačních databází tento model nepodporuje spojovací podmínku *JOIN* využívanou v SQL syntaxi pro

propojení tabulek databáze. Důvodem je orientace modelu na sloupce, kdy základní jednotkou je sloupec, kdežto u relačních databází je základní jednotkou řádek (Harrington, 2016). Jednotlivým řádkům tedy mohou být dle požadavků přiděleny různé sloupce. Sloupce obsahující podobnou informaci jsou následně seskupovány do tzv. *column families* (Harrington, 2016). Tyto sloupce jsou ukládány na fyzických úložištích ve své blízkosti. Z této vlastnosti vyplývá hlavní výhoda tohoto modelu – agregace. Díky agregaci jsou sloupcově-orientované databáze velice efektivní při načítání souhrnu dat z jednoho sloupce.

- **Grafové databáze**

  Tyto typy databází prezentují data v podobě grafů skládajících se z uzlů, jež lze považovat za ekvivalent entit. Uzly mohou být popsány vlastnostmi, které specifikují daný uzel. Jednotlivé uzly jsou následně propojeny hranami, jenž nalézají relace a propojují jednotlivé části. Výsledný pohled připomíná graf. Výhodou grafového modelu je práce s daty, při které není nutné využít JOIN, neboť data jsou již z podstaty databáze propojena hranami, a tedy při dotazování využívá DBMS již existující relace.

2.5 Výpočet objemu kulatiny

Výpočet objemu, též nazýván jako kubírování kulatiny, je proces, při kterém se zjišťuje objem kulatiny bez kůry. Pro výpočet objemu jsou využívány kubírovací vzorce a tabulky.

2.5.1 Základní pojmy

V průběhu práce jsou využívány pojmy, které jsou definovány v tabulce 2. Značná část pojmů je definována dle normy ČSN EN 844 (2020).

Tabulka 2 Základní pojmy

Pojem	Definice	Norma
bez kůry	Výraz používaný v souvislosti s termínem pro měření, který označuje, že měření nezahrnuje kůru.	ČSN EN 844
čelová tloušťka	Tloušťka měřená na tlustém konci (čele).	
čepová tloušťka	Tloušťka měřená na tenkém konci (čepu).	ČSN EN 844
délka	„Nejkratší vzdálenost mezi čely výřezu.“	ČSN EN 844
kmen	Nadzemní část stromu bez větví.	ČSN EN 844
kubírování	Zjištění objemu kulatiny dle vzorců či tabulek.	
kulatina	Pokácený strom s odděleným vrškem a větvemi, který může nebo nemusí být dále krácen, kromě palivového dříví.	ČSN EN 844
kůra	Vnější ochranná vrstva kmene a větví stromu.	ČSN EN 844
objem v m^3	Skutečný objem kulatiny stanovený z jejich rozměrů.	ČSN EN 844
středová tloušťka (průměr)	Tloušťka měřená v půli délky kulatiny.	
výřez	Vydruhovaná část kulatiny.	ČSN EN 844

Zdroj: (ČSN EN 844: Kulatina a řezivo – Terminologie, 2020), upraveno autorem

2.5.2 Kubírovací vzorce

Vzorce jsou odvozeny od stereometrických vzorců, kde výřez je nahrazen rotačním tělesem. Jednotlivé vzorce počítají s délkou výřezu a následně se liší v potřebě průměrů – čelový, středový a čepový. Výsledná hodnota objemu je závislá na přesnosti měření jednotlivých parametrů, ovšem u každého vzorce se objevuje chybovost v řádech procent.

Huberův vzorec

$$V = g_{\frac{1}{2}} \cdot L = \frac{\pi}{4} \cdot d_{\frac{1}{2}}^2 \cdot L \cdot 10^{-4} \qquad (1)$$

V objem výřezu v m^3

$g_{1/2}$ kruhová plocha ve středu kmene v m^2

$d_{1/2}$ je středová tloušťka měřená v centimetrech

L délka kmene v metrech

Smalianův vzorec

$$V = \frac{1}{2} \cdot (g_0 \cdot g_n) \cdot L = \frac{\pi}{4} \cdot \frac{d_0^2 + d_n^2}{2} \cdot L \cdot 10^{-4} \qquad (2)$$

V objem výřezu v m^3

g_0 kruhová plocha čela v m^2

g_n kruhová plocha čepu v m^2

d_0 čepová tloušťka měřená v centimetrech

d_n čelová tloušťka měřená v centimetrech

L délka kmene v metrech

Newtonův vzorec

$$V = \frac{1}{6} \cdot \left(g_0 \cdot 4g_{\frac{1}{2}} \cdot g_n\right) = \frac{\pi}{4} \cdot \frac{d_0^2 + 4 \cdot d_{\frac{1}{2}}^2 + d_n^2}{6} \cdot L \cdot 10^{-4} \qquad (3)$$

V objem výřezu v m^3

g_0 kruhová plocha čela v m^2

$g_{1/2}$ kruhová plocha ve středu kmene

g_n kruhová plocha čepu v m^2

d_0 čepová tloušťka

$d_{1/2}$ je středová tloušťka měřená v centimetrech

d_n čelová tloušťka

L délka kmene v metrech

2.5.3 Kubírovací tabulky

Tabulky objemu kulatiny jsou definovány českými technickými normami. V bakalářské práci jsem zahrnul normy ČSN 48 0007 A ČSN 48 0009.

ČSN 48 0007

Tato norma definuje tabulky pro výpočet objemu dle středové tloušťky bez kůry. Norma vychází z upraveného Huberova vzorce. Vstupními parametry je délka a středová tloušťka měřená bez kůry.

ČSN 48 0009

Norma definující tabulky objemu kulatiny dle středové tloušťky měřené v kůře. Obsahuje koeficienty pro přepočet tloušťky kůry. Tyto koeficienty jsou rozděleny do skupin dle typu dřevin (viz Tabulka 3). Samotný výpočet objemu vychází z Huberova vzorce.

$$V = \frac{\pi}{4} \cdot \left[d_{\frac{1}{2}} - \left(p_0 + p_1 \cdot d_{\frac{1}{2}}^{p_2} \right) \right]^2 \cdot L \cdot 10^{-4} \qquad (4)$$

V objem výřezu v m^3

$d_{1/2}$ je středová tloušťka měřená v centimetrech

L délka kmene v metrech

Tabulka 3 Parametry a skupiny dřevin

Skupina dřevin	Parametry		
	p_0	p_1	p_2
1. skupina	0,57723	0,006897	1,3123
2. skupina	0,24017	0,001915	1,7866
3. skupina	1,7015	0,008762	1,4568
4. skupina	-0,04088	0,16634	0,56076
5. skupina	1,2474	0,042323	1,0623

Zdroj: (ČSN 480009, 2020), upraveno autorem

3 Metodika

Pro vývoj desktopové aplikace byla zvolena platformu .NET Core a programovací jazyk C# vyvíjené firmou Microsoft, které jsou popsány v teoretické části práce. Konkrétně se jedná o verzi .NET Core 3.1. a C# 8.0. Aplikace využívá architekturu MVVM pro oddělení logické vrstvy od prezentační. K ukládání dat je využita relační databáze SQLite šířená pod licencí public domain. Databáze je nasazena lokálně – bude vytvořena při instalaci aplikace a následně bude sloužit po celou dobu životního cyklu aplikace. Pro přístup k datům byl zvolen softwarový rámec Entity Framework Core, který poskytuje objektově relační mapování. Aplikace je určena pro desktopový systém Windows. Ve snaze o zajištění podpory pro většinu verzí tohoto operačního systému jsem zvolil knihovny Windows Presentation Foundation (WPF). Technologie WPF poskytuje grafické rozhraní pro aplikace určené na operační systém Windows a jedná se o dlouholetou technologii využívanou jednoduchými aplikacemi i komplexními systémy. WPF využívá pro vytvoření grafického rozhraní značkovací jazyk XAML, díky kterému lze využít buďto zabudované ovládací prvky, nebo lze vytvořit vlastní. K vývoji aplikace je užito vývojové prostředí Microsoft Visual Studio 2019, jež podporuje výše zmíněné technologie, případně umožňuje jejich instalaci prostřednictvím správce balíčků NuGet. Při vývoji bylo využito několika rozšiřujících balíčků, jenž jsou popsány v následující kapitole.

4 Praktická část

4.1 Charakteristika aplikace

Vyvíjená aplikace je určena pro výpočet objemu kulatiny. Uživatel si může vybrat z několika vzorců, které se liší potřebnými parametry a způsobem výpočtu. Při výpočtu lze zahrnout i proměnné, jež přímo ovlivňují výslednou hodnotu. Dále lze k danému výpočtu uvést doplňující informace týkající se druhu dřeviny, jakosti a odhadované ceny dané kulatiny v závislosti na zadané ceně za metr krychlový. Přesnost výpočtu je závislá na přesnosti zadaných hodnot, ovšem se stále jedná o přibližné hodnoty, které se mohou lišit od skutečnosti. Veškerá data jsou ukládána do databáze a zároveň jsou prezentována v tabulce, kde se nabízí možnost editace či smazání jakékoliv výpočtu. Při exportu dat si lze zvolit data dle výpočtu. Exportování dat je možno do formátů XLSX a CSV, jež jsou podporovány nejčastěji využívanými tabulkovými editory a v případě druhého formátu i informačními systémy.

4.2 Tvorba databáze

Pro ukládání dat je použit relační databázový systém SQLite v kombinaci s Entity Framework Core, díky kterému je databáze vytvořena a spravována. Pro jeho využití je nezbytná instalace rozšiřujících balíčků. EF Core je distribuován pomocí NuGet balíčku s názvem Microsoft.EntityFrameworkCore. Dále pro práci s databází SQLite je potřeba naistalovat balíček Microsoft.EntityFrameworkCore.Sqlite, jenž obsahuje daný databázový systém přizpůsobený pro EF Core. Posledním využitým balíčkem pro správu databáze je Microsoft.EntityFrameworkCore.Tools, který umožňuje využít příkazy pro správu migrací v konzoli správce balíčků Nuget.

Databáze je vytvořena pomocí Code first přístupu, což znamená vygenerování databáze na základě modelů, jenž jsou reprezentovány třídami. Při tvorbě modelů je důležité dodržet stanovená pravidla pro jejich tvorbu, aby byla zaručena funkčnost vztahů mezi jednotlivými modely. Změny v modelu, při již vygenerované databázi umožňují migrace, ale při určitých operacích hrozí ztráta dat. Modely jsou vytvářeny dle ER diagramu v Příloha 1.

4.2.1 Tvorba modelů

Modely reprezentují tabulky databáze, a tedy obsahují atributy příslušné tabulky včetně relací. Pro zajištění funkčnosti relací je nezbytné definovat v modelech navigační vlastnosti. Veškeré modely jsou propojeny třídou Entity, která obsahuje atribut ID. Tento

atribut je formulován technologií EF Core jako primární klíč a je obsažen ve všech modelech.

Model Entity

Model Entity je základní třídou pro ostatní modely. Obsahuje pouze jeden atribut – integer Id (viz Příloha 2). Jedná se o primární klíč, díky kterému je přistupováno do ostatních tabulek. Existence základní třídy umožňuje vytvořit generickou třídu pro CRUD operace s daty databáze. Daná třída je popsána v následující kapitole.

Model Tree

Model Tree, vyobrazen v příloze 5, je potomkem modelu Entity a obsahuje druhy dřevin.

- string TypeOfTree – obsahuje typ stromu, který uživatel může vybrat ze seznamu
- int CalculationParametersId – cizí klíč tabulky CalculationParameters
- ICollection<Log> Logs – navigační vlastnost k modelu Log
- Calculationparameters CalculationParameters – navigační vlastnost

Model Log

Log (viz Příloha 3) obsahuje informace týkající se dané kulatiny, a tedy informace nutné pro výpočet. Stejně jako předchozí model je potomkem modelu Entity.

- double DiameterTop – průměr na tenkém konci kulatiny
- double DiameterMiddle – středový průměr kulatiny
- double DiameterBottom – průměr na tlustém konci kulatiny
- double Volume – objem kulatiny
- double Length – délka kulatiny
- double Value – cena kulatiny
- double Bark – tloušťka kůry
- string Tag – poznámka k danému výpočtu
- string Quality – informace o jakosti kulatiny
- Tree Tree – navigační vlastnost k modelu Tree
- Calculation Calculation – navigační vlastnost k modelu Calculation
- Quality Quiality – navigační vlastnost k modelu Quality

Model Calculation

Calculation (viz Příloha 6) obsahuje informace o zvoleném způsobu výpočtu.

- string TypeOfCalculation – metoda pro výpočet objemu
- string Description – základní popis dané metody
- ICollection<Log> Logs – navigační vlastnost
- ICollection<CalculationParameters> CalculationParameters – navigační vlastnost

Model CalculationParameters

Model CalculationParameters (viz Příloha 4) je určen pro koeficienty, jež určité metody pro výpočet objemu vyžadují.

- double E0, E1, E2 – koeficienty využité ve vzorci
- string Name – název skupin sjednocující typy dřevin.
- int CalculationId – cizí klíč
- ICollection<Tree> Trees – navigační vlastnost
- Calculation Calculations – navigační vlastnost

Model Quality

V tomto modelu (viz Příloha 7) se nachází jakostní třídy, které může uživatel přiřazovat k jednotlivým výřezům.

- string QualityClass – jakostní třídy
- ICollection<Log> Logs – navigační vlastnost

4.2.2 Komunikace mezi modely a databází

Pro komunikaci mezi jednotlivými modely a databází slouží model LogContext. Model je potomkem třídy DbContext, jež je součástí Entity Framework Core. Každá instance této třídy představuje propojení s databází, které je nezbytné pro dotazování a ukládání dat obsažených v instancích modelů. Model LogContext lze tedy chápat za jakýsi most mezi modely a databází.

Jak lze vidět v Příloha 8, model obsahuje kolekce typu DbSet<TEntity> pro každý model/entitu, které jsou mapovány ke stejnojmenným tabulkám v databázi. Dále se v modelu objevují dvě přepsané základní metody. První metodou je OnConfiguring, jež přepsáním umožňuje konfigurovat databázi. Metoda obsahuje parametr optionsBuilder, díky kterému lze vytvářet či upravovat nastavení. Pomocí zmíněného parametru je

nastavena cesta k samotné databázi. OnModelCreating je druhá metoda umožňující pokročilou konfiguraci modelů. V této metodě jsou nakonfigurovány vztahy mezi jednotlivými modely a zároveň je využita pro načtení dat do databáze při jejím vytvoření. Veškeré tato pravidla jsou aplikována pomocí migrací.

4.3 Práce s daty

Pro zajištění základních operací s daty jsou určeny dvě třídy: generická třída DataService<T>, kdy při instanci třídy je možno zvolit jakýkoliv typ dědící z modelu Entity, a třída LogDataService, jenž využívá předchozí třídu, a navíc přidává metody spojené s modelem Log.

4.3.1 Třída DataService<T>

Generická třída DataService<T> implementuje rozhraní IDataService<T>, které obsahuje metody umožňující CRUD operace. V každé metodě je inicializována nová instance třídy LogContext. Životnost instance třídy DbContext by měla být velmi krátká. Je totiž speciálně navrhnuta pro vykonání jediné operace. Po dokončení je nezbytné buďto zavolat interní metodu Entity Framework Core zvanou Dispose pro uvolnění paměti, nebo využít příkaz using, který po provedení operace automaticky uvolní paměť.

4.3.1.1 Metoda Add

Metoda Add (viz Ukázka kódu 1) slouží k přidání entity do databáze. Přidání entity probíhá pomocí metody TrackGraph, která je součástí Entity Framework Core. Metoda kontroluje, zda přidávaná entita již není sledována. Pokud tomu tak je, daná entita zůstane netknutá. V opačném případě začne být entita sledována, a tedy bude dostupná skrze její navigační vlastnosti v aktuální instanci proměnné context. Po přidání entity je zavolána metoda SaveChanges, která uloží veškeré provedené změny v aktuální instanci modelu LogContext. Metoda je návratového typu, kdy vratnou hodnotou je poslední přidaná entita získána dle posledního přidaného primárního klíče.

Ukázka kódu 1 Metoda Add

```
public T Add(T entity)
{
    T returnValue;
    using (var context = new LogContext())
    {
        context.ChangeTracker.TrackGraph(entity, note =>
        {
            note.Entry.State =
             !note.Entry.IsKeySet ? EntityState.Added : EntityState.Unchanged;
        });
        context.SaveChanges();
        returnValue = context.Set<T>().OrderBy(x => x.Id).LastOrDefault();
    }
    return returnValue;
}
```

Zdroj: Vlastní zpracování

4.3.1.2 Metoda Get

Ukázka kódu 2 znázorňuje metodu Get. Metoda slouží k získání specifické entity dle primárního klíče, jenž je při volání metody zadán formou parametru. Návratovou hodnotou je první nalezená entita s odpovídajícím primárním klíčem.

Ukázka kódu 2 Metoda Get

```
public T Get(int id)
{
    T returnValue;
    using (var context = new LogContext())
    {
        returnValue = context.Set<T>().FirstOrDefault(x => x.Id == id);
    }
    return returnValue;
}
```

Zdroj: Vlastní zpracování

4.3.1.3 Metoda GetAll

Metoda GetAll (viz Ukázka kódu 3) má obdobnou funkci jako metoda Get, kdy metoda GetAll navrací veškeré entity obsažené v tabulce. Návratová hodnota je typu IEnumerable<T>.

Ukázka kódu 3 Metoda GetAll

```
public IEnumerable<T> GetAll()
{
    IEnumerable<T> list;
    using (var context = new LogContext())
    {
        list = context.Set<T>().ToList();
    }
    return list;
}
```

Zdroj: Vlastní zpracování

4.3.1.4 Metoda Delete

Metoda Delete (viz Ukázka kódu 4) slouží k mazání specifické entity. Daná entita je nalezena dle primárního klíče získaného z parametru metody. Ke smazání je využita metoda Remove, která je součástí Entity Framework Core. Entita je smazána při zavolání.

Ukázka kódu 4 Metoda Delete

```
public void Delete(int id)
{
    using var context = new LogContext();
    T entity = context.Set<T>().FirstOrDefault(x => x.Id == id);
    context.Set<T>().Remove(entity);
    context.SaveChanges();
}
```

Zdroj: Vlastní zpracování

4.3.1.5 Metoda Update

Update (viz Ukázka kódu 5) je metoda sloužící ke změně již existující entity. Obsahuje dva parametry: id a entity. Parametr id obsahuje hodnotu primárního klíče entity, jenž má být změněna. Druhý parametr obsahuje aktualizovaná data. Samotná změna je provedena pomocí metody Update obsažené v Entity Framework Core. Změna je provedena při zavolání metody SaveChanges. Návratovou hodnotou metody je změněná entita.

Ukázka kódu 5 Metoda Update

```
public T Update(int id, T entity)
{
    using (var context = new LogContext())
    {
        entity.Id = id;
        context.Update(entity);
        context.SaveChanges();
    }
    return entity;
}
```

Zdroj: Vlastní zpracování

4.3.2 Třída LogDataService

Třída LogDataService implementuje rozhraní ILogDataService a slouží pro práci s daty spojené s modelem Log. Aby se předešlo duplicitnímu kódu, pro základní CRUD operace je využita generická třída DataService<T>, kdy DataService je typu Log. LogDataService ovšem navíc obsahuje metody specificky navržené pro daný model.

4.3.2.1 Metoda GetLogs

GetLogs (viz Ukázka kódu 6) slouží pro získání veškerých souvisejících dat modelu Log. Metoda Include slouží k dotazování relačních dat a funguje na obdobném principu jako spojovací podmínka JOIN, avšak metoda Include načte veškerá data bez možnosti filtrace. Návratová hodnota je typu IEnumerable<Log>.

Ukázka kódu 6 Metoda GetLogs

```
public IEnumerable<Log> GetLogs()
{
    using LogContext db = new LogContext();
    IEnumerable<Log> entities = db.Logs
        .Include(a => a.Calculation)
        .Include(a => a.Quality)
        .Include(a => a.Tree)
        .ToList();
    return entities;
}
```

Zdroj: Vlastní zpracování

4.3.2.2 Metoda GetLogsByCalculation

Metoda GetLogsByCalculation (viz Ukázka kódu 7) slouží k získání dat dle parametru reprezentující název metody výpočtu. Pro získání dat je využita Metoda GetLogs. Záznamy jsou následně vyfiltrovány dle zadaného parametru.

Ukázka kódu 7 Metoda GetLogsByCalculation

```
public IEnumerable<Log> GetLogsByCalculation(string typeOfCalculation)
{
    return GetLogs()
        .Where(x => x.Calculation.TypeOfCalculation == typeOfCalculation)
        .ToList();
}
```

Zdroj: Vlastní zpracování

4.3.2.3 Metoda DeleteAll

Účelem metody DeleteAll (viz Ukázka kódu 8) je smazání veškerých záznamů v tabulce Log. Pro tento účel je využita metoda RemoveRange, kde pro označení entit ke smazání je použita vlastnost Logs obsažená v třídě LogContext reprezentující stejnojmennou tabulku v databázi.

Ukázka kódu 8 Metoda DeleteAll

```
public void DeleteAll(IEnumerable<Log> deleteEntities)
{
    using LogContext context = new LogContext();
    context.RemoveRange(context.Logs);
    context.SaveChanges();
}
```

Zdroj: Vlastní zpracování

4.4 Výpočet objemu

Zásadní funkcionalitou aplikace je výpočet objemu kulatiny, který lze realizovat prostřednictvím vzorců a tabulek popsaných v kapitole 2.5. Vzhledem k opakujícím se proměnným v jednotlivých vzorcích a ve snaze zajištění škálovatelnosti byla zvolena pro veškeré typy výpočtů společná základní třída, do které jsou průběžně ukládána potřebná data. Potomci této třídy následně obsahují odlišné vzorce pro samotný výpočet. V případě potřeby lze snadno přidat další výpočetní třídy bez nutnosti zásadních změn. Diagram tříd, vyobrazen v příloze 10, znázorňuje tyto třídy a zároveň jsou podrobně popsány v následujících kapitolách.

4.4.1 NotifyPropertyChanged

Abstraktní třída NotifyPropertyChanged (viz Ukázka kódu 9) je základní třídou pro VolumeCalculation. Jedná se o speciální třídu implementující rozhraní INotifyPropertyChanged, jenž obsahuje událost PropertyChangedEventHandler. Podstatou tohoto rozhraní je informovat ovládací prvky uživatelského prostředí o změnách ve vlastnostech, na které jsou vázány. Účelem metody OnPropertyChanged je vyvolat změnu události, kdy jméno vlastnosti je použito formou parametru pro identifikaci. Je tedy zahrnuta u každé vlastnosti vázané na ovládací prvek uživatelského rozhraní.

Ukázka kódu 9 Třída NotifyPropertyChanged

```
public abstract class NotifyPropertyChanged : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;
    protected void OnPropertyChanged([CallerMemberName] string name = null)
    {
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(name));
    }
}
```

Zdroj: Vlastní zpracování

4.4.2 Třída VolumeCalculation

Abstraktní třída VolumeCalculation je základní třídou pro výpočetní třídy a zároveň obsahuje veškerá data zadaná uživatelem. Jak již bylo nastíněno, jednotlivé výpočty sdílí několik shodných parametrů, proto se samotné výpočty objemů odehrávají ve specializovaných třídách dědících z této třídy. Výhodou tohoto přístupu je snadné přidání nových výpočetních vzorců. Vlastnosti této třídy jsou popsány v tabulce 4. Třída VolumeCalculation obsahuje 5 metod, které jsou v následující části popsány, krom metody CalculateVolume, jenž je abstraktní, a tedy nemá implementované tělo.

Tabulka 4 Vlastnosti třídy VolumeCalculation

Typ	Název	Popis
double	**Bark**	Tloušťka kůry
	DiameterTop	Čepová tloušťka
	DiameterMiddle	Středová tloušťka
	DiameterBottom	Čelová tloušťka
	LogValue	Cena kulatiny
	Volume	Objem kulatiny
	TreeLength	Délka kulatiny
	Price	Cena za 1 m^3
int	**DecimalPlacesCount**	Počet desetinných míst ve výpočtu
	EditId	Primární klíč získaný při editaci výpočtu
	SelectedTreeClass	Vybraná skupina pro výpočet tloušťky kůry
String	**Quality**	Jakostní třída
	Tag	Poznámka k výpočtu
	TypeOfCalculation	Zvolený výpočet
	TypeOfTree	Druh dřeviny
List<string>	**TreeClasses**	Skupiny dřevin

Zdroj: Vlastní zpracování

4.4.2.1 Metoda CalculateLogValue

Metoda CalculateLogVolume (viz Ukázka kódu 10) slouží k výpočtu hodnoty kulatiny, kdy cena je stanovena v konstruktoru třídy. Metoda je volána při změně vlastnosti Volume. Výsledná hodnota je zaokrouhlena metodou Round z knihoven Math na celé číslo. Návratová hodnota je typu double.

Ukázka kódu 10 Metoda CalculateLogvalue

```
public double CalculateLogValue()
{
    return Math.Round(Price * Volume, 0);
}
```

Zdroj: Vlastní zpracování

4.4.2.2 Metoda CreateLog

Účelem metody CreateLog (viz Ukázka kódu 11) je vytvoření entity, kdy jsou veškerá data z vlastností nahrána do modelu Log. Hodnoty vlastností Calculation, Tree a

Quality jsou již obsaženy v databázi a pro jejich dohledání jsou využity Service třídy, ve kterých se nachází metoda GetEntity. Návratová hodnota je typu Log.

Ukázka kódu 11 Metoda CreateLog

```
public Log CreateLog()
{
    Log log = new Log()
    {
        DiameterBottom = diameterBottom,
        DiameterMiddle = diameterMiddle,
        DiameterTop = diameterTop,
        Length = TreeLength,
        Value = LogValue,
        Volume = volume,
        Tag = tag,
        Bark = bark,
        Calculation = calculationService
            .GetEntity(new Calculation
                { TypeOfCalculation = TypeOfCalculation }),
        Tree = treeService
            .GetEntity(new Tree
                { TypeOfTree = TypeOfTree }),
        Quality = qualityService
            .GetEntity(new Quality
                 QualityClass = quality }),
    };
    return log;
}
```

Zdroj: Vlastní zpracování

4.4.2.3 Metoda SaveLog

Pro uložení výpočtu je určena metoda SaveLog (viz Ukázka kódu 12). Při volání metody se vytvoří nová entita skrze metodu CreateLog, jenž je popsána v předchozí kapitole. Po vytvoření je následně entita buďto přidána do databáze nebo je aktualizována již existující entita. Vlastnost EditId obsahuje primární klíč entity Log v případě úpravy výpočtu. Pro přidání či úpravu jsou využity metody třídy LogDataService. Návratová hodnota je typu Log.

Ukázka kódu 12 Metoda SaveLog

```
public Log SaveLog()
{
    Log log = CreateLog();
    if (EditId == 0)
    {
        log = logService.Add(log);
    }
    else
    {
        log = logService.Update(EditId, log);
        EditId = 0;
    }
    return log;
}
```

Zdroj: Vlastní zpracování

4.4.2.4 Metoda EditLog

Úprava výpočtu probíhá prostřednictvím metody EditLog (viz Ukázka kódu 13), při které jsou načteny data předané v parametru typu Log. Vlastnosti Quality a Tree jsou nepovinné, proto je nezbytná kontrola, zdali nemají hodnotu null. Větvení jedné z podmínek obsahuje vlastnost SelectedTreeClass, jenž určuje skupinu dřevin při výpočtu tloušťky kůry. Při využití vzorce s výpočtem tloušťky kůry dle skupiny dřevin je druh dřeviny povinný údaj.

Ukázka kódu 13 Metoda EditLog

```
public void EditLog(Log log)
{
    EditId = log.Id;
    TreeLength = log.Length;
    DiameterTop = log.DiameterTop;
    DiameterMiddle = log.DiameterMiddle;
    DiameterBottom = log.DiameterBottom;
    Volume = log.Volume;
    Bark = log.Bark;
    Tag = log.Tag;
    if (log.Quality != null)
    {
        Quality = log.Quality.QualityClass;
    }
    if (log.Tree != null)
    {
        TypeOfTree = log.Tree.TypeOfTree;
        SelectedTreeClass = log.Tree.CalculationParametersId - 1;
    }
}
```

Zdroj: Vlastní zpracování

4.4.3 Výpočetní třídy

Výpočetními třídami se rozumí veškeré třídy, které mají společnou základní třídu VolumeCalculation. Účelem těchto tříd je výpočet objemu prostřednictvím přepsané metody CalculateVolume, jenž obsahuje implementované vzorce. Konstruktory tříd přiřazují vlastnosti TypeOfCalculation hodnotu typu string obsahující název dané výpočetní metody, eventuálně jsou deklarovány pole s parametry a proměnné potřebné pro specifický vzorec. Pro samotný výpočet je využita třída Math, jež je součástí jazyka C#. Výsledná hodnota je zaokrouhlena při výchozím nastavení na tři desetinná místa, avšak uživatel si může v nastavení počet řádů změnit.

Pro znázornění jsem zvolil přepsanou metodu CalculateVolume (viz Ukázka kódu 14) ze třídy CSN480009, ve které je implementován vzorec dle normy ČSN 48 0009. Parametry využívané ve vzorci jsou obsaženy ve dvoudimenzionálním poli parameters, jehož instance probíhá v konstruktoru. V případě vzniku výjimky typu FormatException je nastaven objem na výchozí hodnotu.

Ukázka kódu 14 Metoda CalculateVolume

```
public override void CalculateVolume()
{
    try
    {
        Volume = Math.Round(Math.PI
            * TreeLength
            * Math.Pow(DiameterMiddle
             - ((2 * parameters[SelectedTreeClass][0])
             + (parameters[SelectedTreeClass][1]
            * Math.Pow(DiameterMiddle, parameters[SelectedTreeClass][2]))), 2)
            / 40000
            , DecimalPlacesCount);
            }
    catch (FormatException)
    {
        Volume = 0;
    }
}
```

Zdroj: Vlastní zpracování

4.4.4 Service třídy

Pomocné třídy Service (viz Příloha 10) obsahují metody zaměřené na jednotlivé tabulky databáze, kdy tyto třídy jsou propojeny pomocí generické třídy BaseService<T>, jenž obsahuje metody určené k identifikaci a získání primárních klíčů k entitám uložených v databázi. Jedná se o metody GetEntity sloužící k získání entit z databáze na základě primárního klíče identifikovaného pomocí metody FindEntity, jejímž účelem je získání veškerých entit z tabulky a jejich prohledání dle zadaných parametrů. Vzhledem k odlišnosti funkcí využitých při prohledávání entit je nezbytné metodu přepsat dne nutnosti, proto je označena jako virtuální. Na ukázce kódu 15 je vyobrazena daná metoda v třídě QualityService. V následujících kapitolách jsou popsány vybrané metody z těchto tříd.

Ukázka kódu 15 Metoda FindId

```
public override int FindId(Quality entity)
{
    List<Quality> list = new List<Quality>(dataService.GetAll());
    return list[
           list.FindIndex(x => x.QualityClass == entity.QualityClass)].Id;
}
```

Zdroj: Vlastní zpracování

4.4.4.1 Metoda GetParameters

Metoda GetParameters (viz Ukázka kódu 16) je součástí třídy CalculationParametersService. Jejím účelem je vytvoření kolekce List<double[]> s dvoudimenzionálními poli obsahující parametry pro výpočet dle normy ČSN 48 0009 a je využita při načítání hodnot v konstruktoru.

Ukázka kódu 16 Metoda GetParameters

```
public List<double[]> GetParameters()
{
    parameters = new List<double[]>();
    for (int i = 0; i < calculationParametersList.Count; i++)
    {
        parameters.Add(new double[] {
            calculationParametersList[i].E0,
            calculationParametersList[i].E1,
            calculationParametersList[i].E2 });
    }
    return parameters;
}
```

Zdroj: Vlastní zpracování

4.4.4.2 Metoda TreeClasses

Metoda TreeClasses (viz Ukázka kódu 17) slouží k načtení skupin dřevin. K jednotlivým skupinám z kolekce parameterClasses jsou přirazeny druhy dřevin nacházejíce se v kolekci trees. Obě zmíněné kolekce jsou inicializovány v konstruktoru třídy. Návratovou hodnotou je generická kolekce typu IEnumerable<string>.

Ukázka kódu 17 Metoda TreeClasses

```
public IEnumerable<string> TreeClasses()
{
    List<string> returnList = new List<string>();
    for (int i = 1; i < 6; i++)
    {
        string values = parameterClasses[i - 1] + "\n";
        foreach (var item in trees)
        {
            if (item.CalculationParametersId == i)
            {
                values = values + item.TypeOfTree + ", ";
            }
        }
        returnList.Add(values.Remove(values.Length - 2));
    }
    return returnList;
}
```

Zdroj: Vlastní zpracování

4.5 Export dat

Veškerá uložená data v databázi lze exportovat do formátů podporující širokou škálu tabulkových editorů. Při exportu může uživatel filtrovat data dle metod výpočtů a zároveň má možnost volby souborového formátu. Aplikace data před exportem vizualizuje. Třída DataExport následně realizuje samotný export dat.

4.5.1 Třída DataExport

Třída DataExport slouží k vytvoření a načtení dat do souborů formátu CSV a XLSX. Konstruktor třídy vyžaduje parametr typu List<Log> s daty určenými k exportu. K vytvoření a zápisu dat do souboru slouží metody CreateExcel a CreateCSV. Pro vytvoření formátu .XLSX je použit balíček ClosedXML šířen pod licencí MIT[6]. Nabízela se možnost využití knihoven Microsoft Excel skrze COM Interop, ale daný program by musel být povinně nainstalován na zařízení a v případě jeho absence by export nefungoval.

4.5.1.1 Metoda CreateExcel

Pro vytvoření souboru typu .XLSX slouží metoda CreateExcel. Jak lze vidět v ukázce kódu 18, metoda vyžaduje parametr path, který obsahuje zvolené umístění

[6] Aplikaci lze využít v jakékoliv podobě a lze ji distribuovat i jako součást proprietárního software

uživatelem. V metodě je deklarována proměnná typu XLWorkBook, pomocí které jsou k dispozici metody pro práci s tabulkovým editorem. Načtení dat předchází vytvoření nového listu sešitu, jenž je pojmenován „Export dat“. Následuje vytvoření záhlaví tabulky a následné načtení hodnot. Posledním krokem je uložení souboru, při kterém je využit parametr metody.

Ukázka kódu 18 Metoda CreateExcel

```
public void CreateExcel(string path)
{
    using XLWorkbook workbook = new XLWorkbook();
    var worksheet = workbook.Worksheets.Add("Export dat");

    worksheet.Cell(1, index++).Value = "ID";
    worksheet.Cell(1, index++).Value = "Typ kalkulace";
    worksheet.Cell(1, index++).Value = "Objem";
    worksheet.Cell(1, index++).Value = "Délka";
    worksheet.Cell(1, index++).Value = "Průměr čepu";
    worksheet.Cell(1, index++).Value = "Průměr středu";
    worksheet.Cell(1, index++).Value = "Průměr čela";
    worksheet.Cell(1, index++).Value = "Cena";
    worksheet.Cell(1, index++).Value = "Jakost";
    worksheet.Cell(1, index++).Value = "Druh dřeviny";
    worksheet.Cell(1, index++).Value = "Kůra";
    worksheet.Cell(1, index).Value = "Poznámka";

    for (int i = 0; i < dataList.Count; i++)
    {
        int j = i + 2;
        worksheet.Cell(j, 1).Value = i + 1;
        worksheet.Cell(j, 2).Value = dataList[i].Calculation.TypeOfCalculation;
        worksheet.Cell(j, 3).Value = dataList[i].Volume;
        worksheet.Cell(j, 4).Value = dataList[i].Length;
        worksheet.Cell(j, 5).Value = dataList[i].DiameterTop;
        worksheet.Cell(j, 6).Value = dataList[i].DiameterMiddle;
        worksheet.Cell(j, 7).Value = dataList[i].DiameterBottom;
        worksheet.Cell(j, 8).Value = dataList[i].Value;
        worksheet.Cell(j, 9).Value = dataList[i].Quality.QualityClass;
        worksheet.Cell(j, 10).Value = dataList[i].Tree.TypeOfTree;
        worksheet.Cell(j, 11).Value = dataList[i].Bark;
        worksheet.Cell(j, 12).Value = dataList[i].Tag;
        j++;
    }
    workbook.SaveAs(path + "/export.xlsx");
}
```

Zdroj: Vlastní zpracování

4.5.1.2 Metoda CreateCSV

Soubory typu CSV (viz Ukázka kódu 19) jsou vytvářeny metodou CreateCSV za použití třídy StreamWriter. Při její deklaraci jsou nastaveny základní parametry souboru jako je umístění pro uložení a nastavení kódování. Poté lze již pomocí metody Write zapisovat do souboru. Obdobně jako u předchozí metody je nejprve vytvořeno záhlaví a následně jsou zapisovány hodnoty. Na začátku průchodu cyklem je vytvořen nový řádek. Uložení souboru je dokončeno prostřednictvím metody Close.

Ukázka kódu 19 Metoda CreateCSV

```
public void CreateCSV(string path)
{
    using StreamWriter sw = new StreamWriter(path + "/export.csv", false,
      System.Text.Encoding.Unicode);
    sw.Write("ID;Typ Kalkulace;Objem;Délka;Průmět čepu;Průměř středu;
              Průměr čela;Jakost,Druh dřeviny;Kůra;Poznámka");
    for (int i = 0; i < dataList.Count; i++)
    {
        sw.Write(sw.NewLine);
        sw.Write("{0};{1};{2};{3};{4};{5};{6};{7};{8};{9};{10}",
        i,
        dataList[i].Calculation.TypeOfCalculation,
        dataList[i].Volume,
        dataList[i].Length,
        dataList[i].DiameterBottom,
        dataList[i].DiameterMiddle,
        dataList[i].DiameterTop,
        dataList[i].Quality.QualityClass,
        dataList[i].Tree.TypeOfTree,
        dataList[i].Bark,
        dataList[i].Tag);
    }
    sw.Close();
}
```

Zdroj: Vlastní zpracování

4.6 Uživatelské rozhraní

Uživatelské rozhraní aplikace je tvořeno knihovnami WPF. Při vývoji byl kladen důraz na oddělení uživatelského rozhraní od logické části aplikace, proto byla zvolena architektura MVVM (Model-View-ViewModel), která aplikaci dělí na tři části. Modelem lze označit třídy uchovávající data, se kterými je možno následně pracovat. V případě této aplikace se jedná o třídy popsané v předchozích kapitolách. View je název pro grafické

rozhraní aplikace, se kterým uživatel pracuje. Je tvořeno značkovacím jazykem XAML, jenž umožňuje snadné vytvoření ovládacích prvků. Poslední částí je ViewModel, který propojuje předchozí dvě části. V těchto třídách se nacházejí např. zformátovaná data určená k prezentaci či příkazy reagující na chování uživatele. Hlavní myšlenkou této architektury je separace logické a prezentační vrstvy, kde tyto dvě vrstvy jsou navzájem nezávislé. Výhodou tohoto přístupu je možnost změny uživatelského rozhraní bez nutnosti provedení změn na modelech a naopak. V praxi se může jednat např. o převedení desktopové aplikace na webovou aplikaci.

4.6.1 Propojení s logickou vrstvou

Jak už bylo zmíněno, o propojení modelů a jednotlivých zobrazení se starají třídy ViewModel, které jsou vyobrazeny pomocí diagramu tříd v příloze 11. Na každý ViewModel je následně vázáno zobrazení View.

Jednotlivá zobrazení obsahují ovládací prvky jako jsou např. kombinovaná pole, textová pole apod., u kterých je využito vázání (anglicky Binding) na vlastnosti obsažené ve ViewModels. Datové vazby jsou určeny pro přenos dat mezi vrstvami. Pro znázornění jsem zvolil vlastnost SelectedCalculation vyobrazenou na ukázce kódu 20.

Ukázka kódu 20 Vlastnost SelectedCalculation

```
public string SelectedCalculation
{
    get => selectedItem;
    set
    {
        if (selectedItem != value)
        {
            selectedItem = value;
            ChangeClass(selectedItem);
            OnPropertyChanged();
        }
    }
}
```

Zdroj: Vlastní zpracování

Na danou vlastnost je vázáno kombinované pole clCalMethods (viz Ukázka kódu 21). Konkrétně se jedná o vlastnost SelectedItem reprezentující aktuálně vybranou položku ze seznamu. V tomto případě je vázání oboustranné, tedy jakákoliv změna je promítnuta buďto do uživatelského prostředí nebo do vlastnosti ve třídě.

Ukázka kódu 21 Ovládací prvek cbCalMethods

```
<ComboBox x:Name="cbCalMethods" Width="155"
          SelectedItem="{Binding Path=SelectedCalculation,
                         UpdateSourceTrigger=PropertyChanged, Mode=TwoWay}"
          SelectedIndex="{Binding Path=DefaultIndex, Mode=OneWay}"
          ItemsSource="{Binding Path=CalculationMethods}">
</ComboBox>
```

Zdroj: Vlastní zpracování

Jedním z nejdůležitějších údělů grafického rozhraní je interakce mezi uživatelem a aplikací, která probíhá prostřednictvím ovládacích prvků. Typickým příkladem je kliknutí na tlačítko. Většina ovládacích prvků využívá pro komunikaci vlastnost Command, jenž je vyvolána při interakci. Příkazy jsou objekty implementující rozhraní ICommand, jenž obsahuje dvě základní metody: CanExecute a Execute. Metoda Execute slouží k vykonání určité akce, zatímco metoda CanExecute ověřuje, zdali jsou splněny podmínky pro vykonání. Pro znázornění jsem zvolil dva typy příkazů vyobrazených v ukázce kódu 22. Oba příkazy jsou deklarovány prostřednictvím třídy RelayCommand, jenž implementuje rozhraní ICommand. Benefitem tohoto přístupu zefektivnění tvorby příkazů, kdy není potřeba pro každý příkaz vytvářet vlastní třídu.

Ukázka kódu 22 Příkazy DeleteCommand a ClearCommand

```
public ICommand ClearCommand =>
        clearCommand ??= new RelayCommand(ClearForm);

public ICommand DeleteCommand =>
        deleteCommand ??= new RelayCommand<Log>(RemoveLog, CanRemove);
```

Zdroj: Vlastní zpracování

První bezparametrový příkaz s názvem ClearCommand slouží k vyčištění formuláře. Při deklaraci je využit pouze povinný parametr Execute, kde je tato metoda skryta pod názvem ClearForm. Příkaz DeleteCommand, určený pro smazání specifického výpočtu z tabulky, již vyžaduje parametr typu Log. Daný parametr předán prostřednictvím vlastnosti CommandParameter obsažené v ovládacím prvku Button (viz Ukázka kódu 23). Při deklaraci je využita metoda Execute (RemoveLog) i CanExecute (CanRemove).

Ukázka kódu 23 Ovládací prvek DeleteButton

```
<Button x:Name="DeleteButton" Background="Transparent" BorderThickness="0"
        ToolTip="Smazat výpočet"
        Command="{Binding RelativeSource=
                    {RelativeSource AncestorType={x:Type DataGrid}},
                    Path=DataContext.DeleteCommand}"
        CommandParameter="{Binding}">
    <Image Source="/Icon/delete-2-16.png"/>
</Button>
```

Zdroj: Vlastní zpracování

4.6.2 Obrazovka Kalkulace

Primárním zobrazením aplikace je obrazovka Kalkulace (viz Příloha 12). Na této obrazovce je uskutečněn výpočet objemu. V pravé části se nachází formulář pro vstupní data. Dle zvoleného výpočtu z kombinovaného pole se přizpůsobují textová pole. Pole jsou seřazena hierarchicky, kdy povinná pole jsou v horní části a volitelná pole v dolní části. Při vyplnění veškerých povinných polí se aktivuje tlačítko Přidat. Nad textovými poli je vyobrazen objem a cena kalkulovaná ze zadaných dat. Tyto hodnoty jsou aktualizovány při změně dat. V pravém horním rohu se nachází tlačítko pro vyčištění formuláře. Po stisknutí jsou veškerá data z formuláře smazána.

V pravé části obrazovky se nachází tabulka zobrazující data dle zvoleného výpočtu. Jednotlivé výpočty mohou být smazány či upraveny tlačítky v pravé části tabulky. V záhlaví tabulky se nachází tlačítko pro smazání veškerých záznamů. Veškerá tlačítka jsou intuitivně označena ikonami a současně při najetí myší je zobrazen popisek. V dolní části obrazovky se nachází celkový objem a celková hodnota veškerých kalkulací v tabulce.

4.6.3 Obrazovka Export dat

Pro export dat slouží stejnojmenná obrazovka vyobrazená v příloze 13. Obrazovka je rozdělena na dvě části. V levé části jsou vizualizovány data v podobě, ve které budou následně exportována. Tyto data mohou být filtrována dle výpočtu. Pravá část obrazovky slouží k zadání údajů nezbytných pro export dat. Uživatel musí zadat cílový adresář a zvolit formát generovaného souboru. Pro výběr adresáře lze využít grafické rozhraní FolderBrowserDialog, které je součástí systému Windows a lze ho spustit tlačítkem.

5 Závěr

Cílem této práce bylo vytvoření desktopové aplikace pro výpočet objemu kulatiny. K vývoji aplikace byl využit programovací jazyk C#, platforma .NET Core 3.1 a framework Windows Presentation Foundation. Při návrhu i následném vývoji bylo dbáno na dodržení principů objektově orientovaného programování a pravidel architektury MVVM. Aplikace uchovává data v relační databázi SQLite, ke které je přistupováno pomocí Entity Framework Core. Uložená data lze exportovat do tabulkových procesorů. Podporovanými formáty exportů jsou XLSX a CSV. Uživatel má možnost volby vzorce či normy, dle kterého jsou provedeny výpočty. Každý výpočet krom povinných údajů o rozměrech může obsahovat typ dřeviny, jakost a tloušťku kůry. Pro zajištění maximální kompatibility je využit Self-Contained Deployment (SCD), jenž umožňuje distribuci aplikace společně s frameworkem .NET Core.

Aplikace splňuje zadané požadavky a neobsahuje žádné kritické chyby, proto ji lze používat v reálném prostředí. Slabou stránkou aplikace je nedostatečná zpětná vazba, která se projevuje například při exportu dat, kdy při uložení souboru není indikováno dokončení procesu, a pouze základní validace vstupních dat.

Na aplikaci plánuji dále pracovat a přidávat nové funkcionality. Co se týče technické stránky, primárním krokem v následující vývoji bude migrace aplikace na novou verzi platformy .NET s označením 5. Z pohledu funkcionalit plánuji rozšít výpočetní metody o vzorce a normy využívané v sousedních státech a přidat více formátů pro export dat.

I. Summary and keywords

The main goal of this bachelor thesis is to develop an application suitable for calculating log volume using different equations. The first part of this work describes the usage of a relational database, C# programming language and individual components of the .NET platform. In addition to that, the thesis specifies equations used for calculating volume, which are based on the measurement method and the type of wood. Data are then recorded in a relational database. A user can display and filter database records by various types of fields, which can be exported to basics common spreadsheet formats such as an XLSX and a CSV. The thesis describes the process of developing a desktop application using Microsoft Visual Studio an integrated development environment.

Key words: desktop app, C#, .NET, WPF, database, log volume

II. Seznam použitých zdrojů

Albahari, J., & Johannsen, E. (2019). *C# 8.0 in a Nutshell: The Definitive Reference* (8. vyd.). O'Reilly Media, Incorporated.

Bellis, M. (2019, říjen 4). The Unusual History of Microsoft Windows. Získáno 15. březen 2021, z ThoughtCo website: https://www.thoughtco.com/unusual-history-of-microsoft-windows-1992140

Blevins, L., Wojciakowski, M., & Graham, L. (2021, únor 3). Choose your Windows app platform—Windows applications. Získáno 15. březen 2021, z https://docs.microsoft.com/en-us/windows/apps/desktop/choose-your-platform

ČSN EN 844: Kulatina a řezivo – Terminologie. (2020) (2020 ed.). Praha: Úřad pro technickou normalizaci, metrologii a státní zkušebnictví.

ČSN 48 0007: Tabulky objemu kulatiny podle středové tloušťky. (1959). Praha: Úřad pro technickou normalizaci, metrologii a státní zkušebnictví.

ČSN 48 0009: Tabulky objemu kulatiny bez kůry podle středové tloušťky měřené v kůře. (1977). Praha: Úřad pro technickou normalizaci, metrologii a státní zkušebnictví.

De George, A. (2017, březen 30). Common Type System. Získáno 25. leden 2021, z https://docs.microsoft.com/en-us/dotnet/standard/base-types/common-type-system

Definition of desktop application. (b.r.). Získáno 16. březen 2021, z PCMAG website: https://www.pcmag.com/encyclopedia/term/desktop-application

Desktop Windows Version Market Share Worldwide. (2020, prosinec). Získáno 15. březen 2021, z StatCounter Global Stats website: https://gs.statcounter.com/os-version-market-share/windows/desktop/worldwide

Douglas, J. (2019, květen 24). An introduction to NuGet. Získáno 28. leden 2021, z Microsoft Docs website: https://docs.microsoft.com/en-us/nuget/what-is-nuget#the-flow-of-packages-between-creators-hosts-and-consumers

Elmasri, R., & Navathe, S. (2016). *Fundamentals of database systems* (Seventh edition). Hoboken, NJ: Pearson.

Harrington, J. L. (2016). *Relational database design and implementation: Clearly explained* (4th edition). Cambridge, MA: Elsevier.

Hunter, S. (2018, říjen 4). Update on .NET Core 3.0 and .NET Framework 4.8. Získáno 1. únor 2021, z .NET Blog website: https://devblogs.microsoft.com/dotnet/update-on-net-core-3-0-and-net-framework-4-8/

Johnson, J. (2020, srpen 25). What is Xamarin? Získáno 31. leden 2021, z https://docs.microsoft.com/en-us/xamarin/get-started/what-is-xamarin

Lambson, B. (2018, září 19). Entity Framework Core tools reference—EF Core. Získáno 27. leden 2021, z https://docs.microsoft.com/en-us/ef/core/cli/

NetMarketShare. (2020). Získáno 15. březen 2021, z NetMarketShare website: https://netmarketshare.com/

Price, M. J. (2019). *C# 8.0 and .NET Core 3.0—Modern cross-platform development, fourth edition*.

Produkty ukončující podporu. (2020, červenec 17). Získáno 25. leden 2021, z https://docs.microsoft.com/cs-cz/lifecycle/end-of-support/end-of-support-2020

Schildt, H. (2006). *C# 2.0: The complete reference* (2. ed). New York, NY: McGraw-Hill/Osborne.

Schofield, M., Toliver, K., Parente, J., & Hickey, S. (2020, 09). Fluent Design System for Windows—Windows applications. Získáno 15. březen 2021, z https://docs.microsoft.com/en-us/windows/apps/fluent-design-system

Tanenbaum, A. S. (2015). *Modern operating systems* (Fourth edition). Boston: Pearson.

Varma, J. (2015). *Xcode 6 Essentials*. Packt Publishing Ltd.

Vaughan-Nichols, S. J. (2011, duben 13). Twenty Years of Linux according to Linus Torvalds. Získáno 31. leden 2021, z ZDNet website: https://www.zdnet.com/article/twenty-years-of-linux-according-to-linus-torvalds/

Wagner, B., Kulikov, P., Yoshioka, H., & McMahon, B. (2020, duben 9). What's new in C# 9.0. Získáno 21. leden 2021, z https://docs.microsoft.com/en-us/dotnet/csharp/whats-new/csharp-9

Warren, G. (2019, listopad 15). Fundamentals of garbage collection. Získáno 25. leden 2021, z https://docs.microsoft.com/en-us/dotnet/standard/garbage-collection/fundamentals

Warren, G. (2020, říjen 5). .NET Standard. Získáno 20. leden 2021, z https://docs.microsoft.com/cs-cz/dotnet/standard/net-standard

III. Seznam tabulek

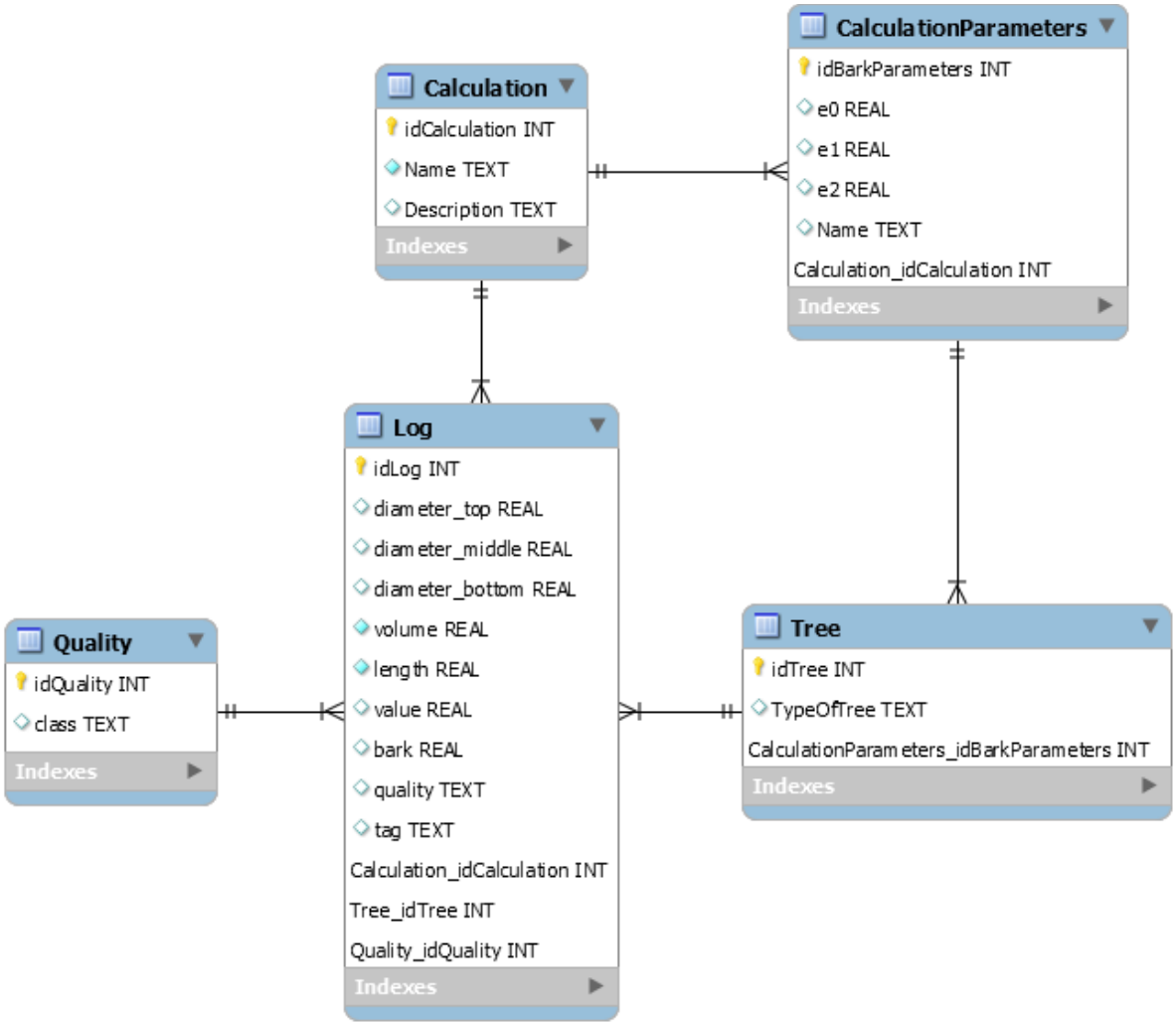
IV. Seznam ukázek kódu

V. Seznam grafů

VI. Seznam příloh

VII. Přílohy

Příloha 1 Entity-relationship model



Zdroj: Vlastní zpracování

Příloha 2 Model Entity

```
public class Entity
{
    public int Id { get; set; }
}
```

Zdroj: Vlastní zpracování

Příloha 3 Model Log

```
public class Log : Entity
{
    public double DiameterTop { get; set; }
    public double DiameterMiddle { get; set; }
    public double DiameterBottom { get; set; }
    public double Volume { get; set; }
    public double Length { get; set; }
    public double Value { get; set; }
    public double Bark { get; set; }
    public string Tag { get; set; }

    public Quality Quality { get; set; }
    public Tree Tree { get; set; }
    public Calculation Calculation { get; set; }
}
```

Zdroj: Vlastní zpracování

Příloha 4 Model CalculationParameters

```
public class CalculationParameters : Entity
{
    public double E0 { get; set; }
    public double E1 { get; set; }
    public double E2 { get; set; }
    public string Name { get; set; }
    public int CalculationId { get; set; }
    public ICollection<Tree> Trees { get; set; }
    public Calculation Calculation { get; set; }
}
```

Zdroj: Vlastní zpracování

Příloha 5 Model Tree

```
public class Tree : Entity
{
    public string TypeOfTree { get; set; }
    public int CalculationParametersId { get; set; }
    public ICollection<Log> Logs { get; set; }
    public CalculationParameters CalculationParameters { get; set; }
}
```

Zdroj: Vlastní zpracování

Příloha 6 Model Calculation

```
public class Calculation : Entity
{
    public string TypeOfCalculation { get; set; }
    public string Description { get; set; }
    public ICollection<Log> Logs { get; set; }
    public ICollection<CalculationParameters> CalculationParameters {get;set;}
}
```

Zdroj: Vlastní zpracování

Příloha 7 Model Quality

```
public class Quality : Entity
{
    public string QualityClass { get; set; }
    public ICollection<Log> Logs { get; set; }
}
```

Zdroj: Vlastní zpracování

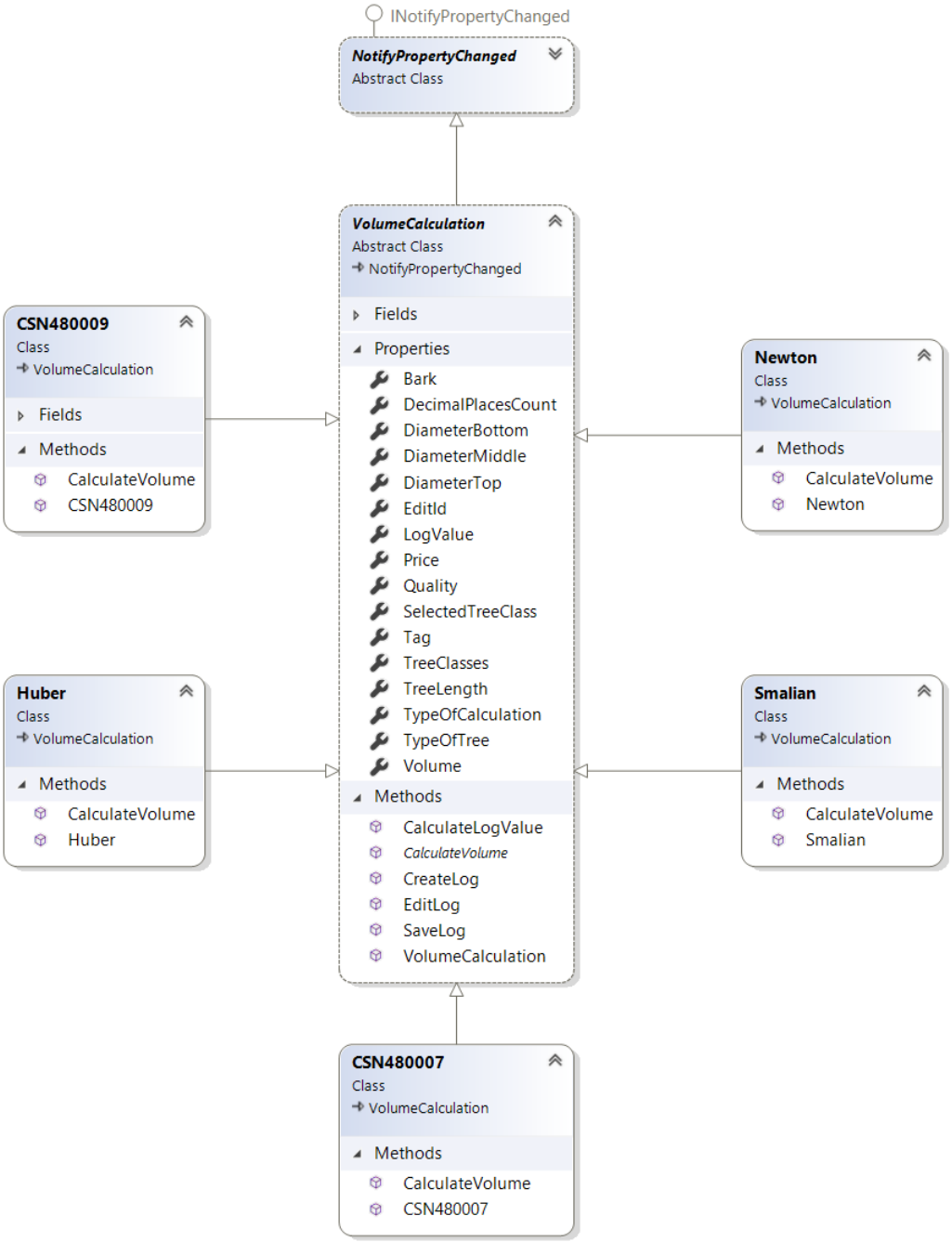
Příloha 8 Model LogContext

```
public class LogContext : DbContext
{
    public virtual DbSet<Log> Logs { get; set; }
    public virtual DbSet<Tree> Trees { get; set; }
    public virtual DbSet<Calculation> Calculations { get; set; }
    public virtual DbSet<Quality> Qualities  { get; set; }
    public virtual DbSet<CalculationParameters> CalculationParameters {get;set;}

    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseSqlite("Data Source=logs.db");
        base.OnConfiguring(optionsBuilder);
    }
    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Calculation>()
            .HasMany(c => c.Logs)
            .WithOne(e => e.Calculation)
            .IsRequired();
        modelBuilder.Entity<Tree>()
            .HasMany(c => c.Logs)
            .WithOne(e => e.Tree)
            .HasForeignKey("CalculationParametersId");
        modelBuilder.Entity<Quality>()
            .HasMany(c => c.Logs)
            .WithOne(e => e.Quality);
        modelBuilder.Entity<CalculationParameters>()
            .HasOne(c => c.Calculation)
            .WithMany(e => e.CalculationParameters)
            .HasForeignKey("CalculationId");
        modelBuilder.Entity<CalculationParameters>()
            .HasMany(c => c.Trees)
            .WithOne(e => e.CalculationParameters)
            .IsRequired();
    }
}
```

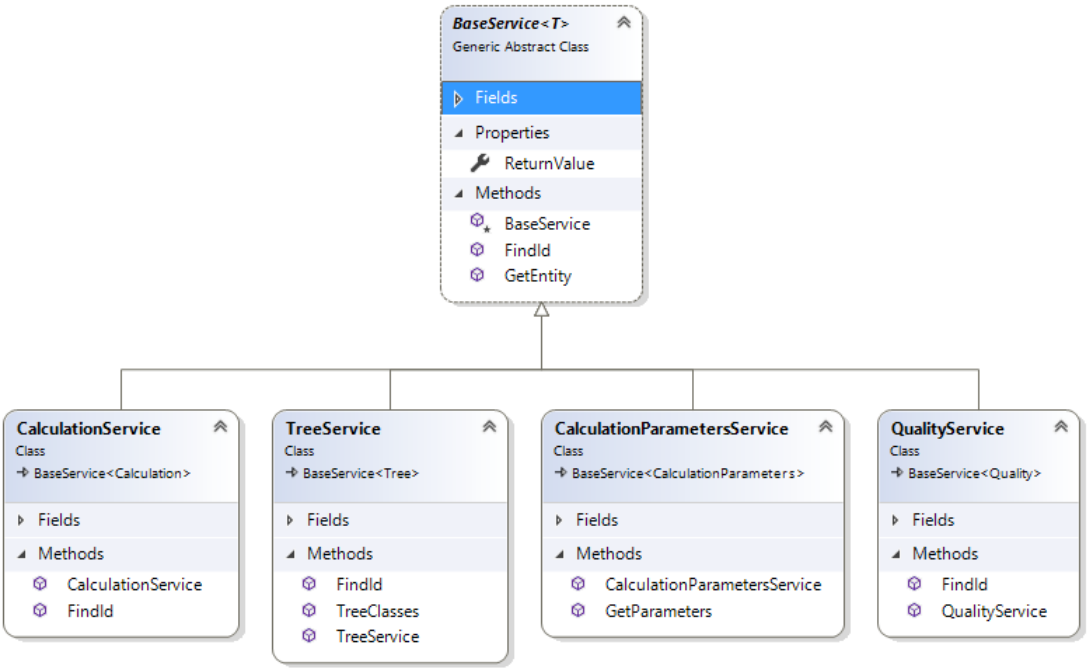
Zdroj: Vlastní zpracování

Příloha 9 Diagram tříd pro výpočet objemu



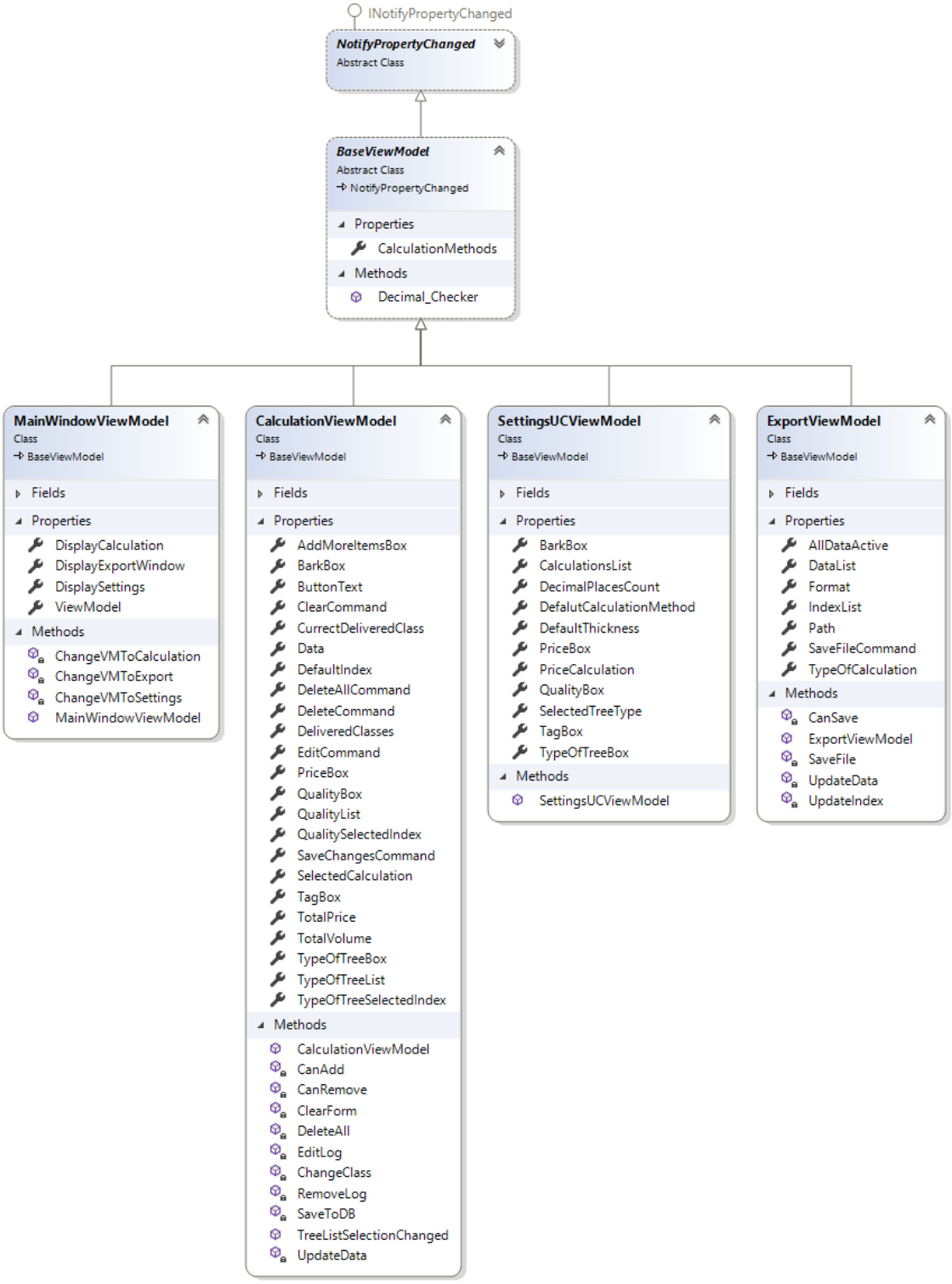
Zdroj: Vlastní zpracování

Příloha 10 Diagram tříd obsahující Service třídy

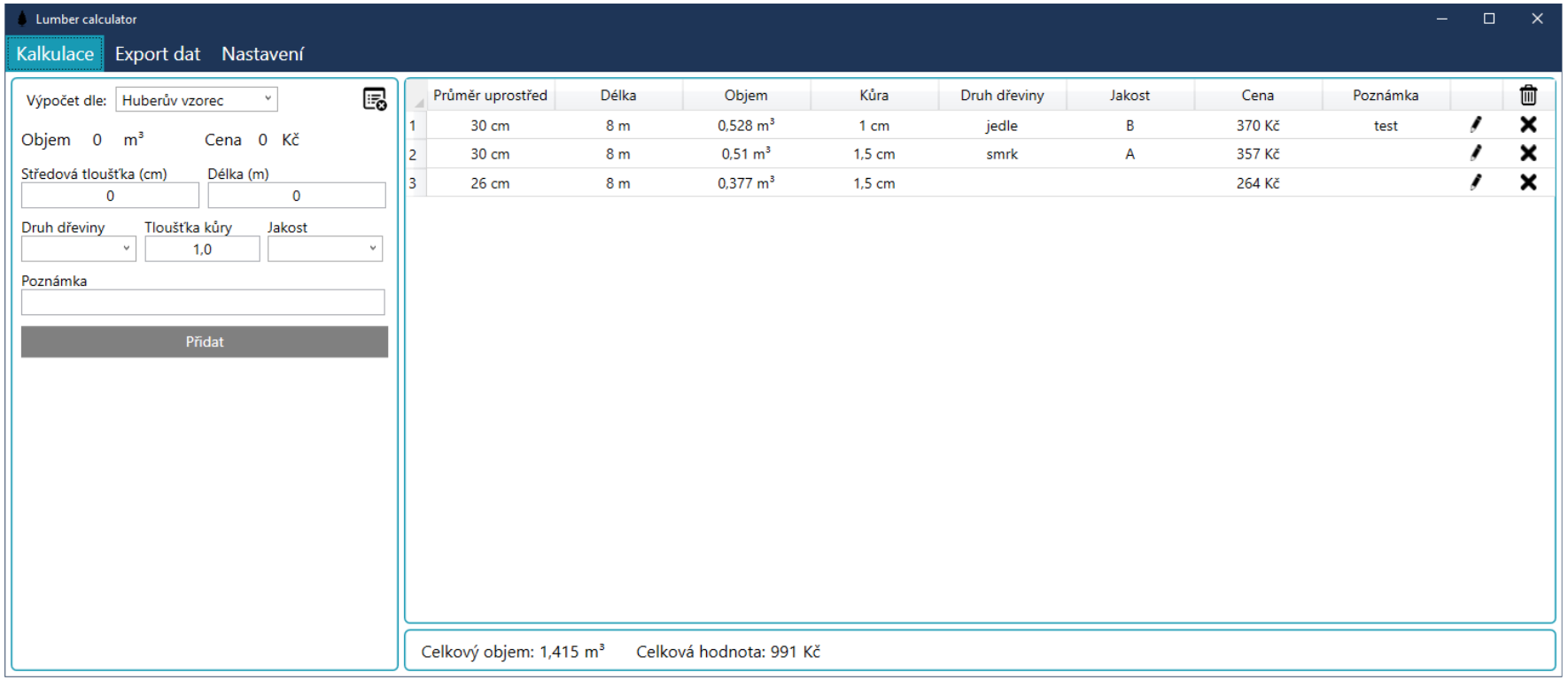


Zdroj: Vlastní zpracování

Příloha 11 Diagram tříd vrstvy ViewModel



Zdroj: Vlastní zpracování



Zdroj: Vlastní zpracování

Příloha 12 Obrazovka Kalkulace

Příloha 13 Obrazovka Export dat

ID	Výpočet	Objem (m³)	Délka (m)	Průměr čepu	Průměr středu	Průměr čela	Cena (Kč)	Jakost	Druh Dřeviny	Kůra	Poznámka
1	Huberův vzorec	0,528	8	0	30	0	370	B	jedle	1	test
2	Huberův vzorec	0,51	8	0	30	0	357	A	smrk	1,5	
3	Huberův vzorec	0,377	8	0	26	0	264			1,5	
4	Newtonův vzorec	0,558	10	20	26	34	391	B	modřín	0	
5	ČSN 48 0009	0,315	8	0	24	0	220	C	smrk	1	
6	ČSN 48 0009	0,244	8	0	24	0	171	C	modřín	1	
7	ČSN 48 0009	0,378	8	0	28,5	0	265	A	dub	1	

Zdroj: Vlastní zpracování