Návrh a implementace WPF aplikace pro 3D detekci a analýzu hracích kostek

Ing. Jaroslav Janků

Bakalářská práce
2024

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky
Ústav informatiky a umělé inteligence

Akademický rok: 2023/2024

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Ing. Jaroslav Janků**
Osobní číslo:	**A21292**
Studijní program:	**B0613A140020 Softwarové inženýrství**
Forma studia:	**Kombinovaná**
Téma práce:	**Návrh a implementace WPF aplikace pro 3D detekci a analýzu hracích kostek**
Téma práce anglicky:	**Design and Implementation of a WPF Application for 3D Detection and Analysis of Dice**

Zásady pro vypracování

1. Popište současný stav technologií a architektonických vzorů pro vývoj desktopových aplikací.
2. Popište princip a možnost využití hloubkových kamer v technické praxi. Zdůrazněte výhody těchto kamer oproti standardním technologiím snímání.
3. Navrhněte řešení dané aplikace s využitím popsaných technologií, popište případy užití, funkční a nefunkční požadavky.
4. Realizujte navrženou aplikaci a popište její klíčové části.
5. Zhodnoťte dosažené výsledky včetně návrhu na další rozvoj aplikace.

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

Seznam doporučené literatury:

1. PRICE, Mark J. C# 9 and .NET 5 – Modern Cross-Platform Development. 5. Birmingham: Packt Publishing, 2020. ISBN 9781800568105.
2. KOVALEVSKY, Vladimir. Modern Algorithms for Image Processing: Computer Imagery by Example Using C#. 1. Berkeley (California): Apress Berkeley, CA, 2019. ISBN 978-1-4842-4237-7
3. J. DU PREEZ, Ockert. Visual Studio 2022 In-Depth: Explore the Fantastic Features of Visual Studio 2022. 2. Noida (India): BPB Publications, 2022. ISBN 9355512457.
4. TADIC V. et al. Perspectives of RealSense and ZED Depth Sensors for Robotic Vision Applications. Machines 2022, 10, 183. https://doi.org/10.3390/machines10030183
5. DOBEŠ, Michal. Zpracování obrazu a algoritmy v C#. Praha: BEN – technická literatura, 2008. ISBN 978-80-7300-233--6.

Vedoucí bakalářské práce: **doc. Ing. Petr Šilhavý, Ph.D.**
Ústav počítačových a komunikačních systémů

Datum zadání bakalářské práce: **5. listopadu 2023**
Termín odevzdání bakalářské práce: **13. května 2024**

UNIVERZITA TOMÁŠE BATI VE ZLÍNĚ
Fakulta aplikované informatiky
DĚKAN
L.S.

doc. Ing. Jiří Vojtěšek, Ph.D. v.r.
děkan

prof. Mgr. Roman Jašek, Ph.D., DBA v.r.
ředitel ústavu

Ve Zlíně dne 5. ledna 2024

Prohlašuji, že

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

Prohlašuji,

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

Ve Zlíně, dne: 12.5.2024 Jaroslav Janků, v. r.

Zásady pro vypracování:

1. Popište současný stav technologií a architektonických vzorů pro vývoj desktopových aplikací.
2. Popište princip a možnost využití hloubkových kamer v technické praxi. Zdůrazněte výhody těchto kamer oproti standardním technologiím snímání.
3. Navrhněte řešení dané aplikace s využitím popsaných technologií, popište případy užití, funkční a nefunkční požadavky.
4. Realizujte navrženou aplikaci a popište její klíčové části.
5. Zhodnoťte dosažené výsledky včetně návrhu na další rozvoj aplikace

Seznam doporučené literatury:

1. PRICE, Mark J. *C# 9 and .NET 5 - Modern Cross-Platform Development*. 5. Birmingham: Packt Publishing, 2020. ISBN 9781800568105.
2. KOVALEVSKY, Vladimir. *Modern Algorithms for Image Processing: Computer Imagery by Example Using C#.* 1. Berkeley (California): Apress Berkeley, CA, 2019. ISBN 978-1-4842-4237-7
3. J. DU PREEZ, Ockert. *Visual Studio 2022 In-Depth: Explore the Fantastic Features of Visual Studio 2022*. 2. Noida (India): BPB Publications, 2022. ISBN 9355512457.
4. Tadic, V.; Toth, A.; Vizvari, Z.; Klincsik, M.; Sari, Z.; Sarcevic, P.; Sarosi, J.; Biro, I. Perspectives of RealSense and ZED Depth Sensors for Robotic Vision Applications. *Machines* 2022, *10*, 183. https://doi.org/10.3390/machines10030183
5. DOBEŠ, Michal. *Zpracování obrazu a algoritmy v C#.* Praha: BEN - technická literatura, 2008. ISBN 978-80-7300-233-6.

ABSTRAKT

Tato bakalářská práce se zaměřuje na návrh a vývoj aplikace Windows Presentation Foundation, která využívá architektonický vzor Model-View-ViewModel pro efektivní rozdělení logiky a uživatelského rozhraní v kontextu 3D zpracování a analýzy obrazu hracích kostek. Pro detekci a hodnocení jednotlivých hodů byla použita hloubková kamera RealSense od společnosti Intel Corporation. Součástí práce je implementace systému pro správu uživatelských profilů v souladu s pravidly ochrany osobních údajů a vytvoření žebříčku nejlepších hráčů.

Klíčová slova: WPF, MVVM, hrací kostky, 3D hloubková kamera, zpracování obrazu

ABSTRACT

This bachelor thesis is focused on the design and development of a WPF application that uses the Model-View-ViewModel (MVVM) architectural pattern to efficiently partition the application logic and user interface in the context of 3D dice image processing and analysis. A RealSense depth camera from Intel Corporation was used to detect and evaluate individual rolls. The work also includes the implementation of a privacy-compliant user profile management system and the creation of a top player ranking system. The proposed system serves as a marketing tool for the presentation of the development department of Meopta-Optika, s.r.o. at trade fairs and other business events.

Keywords: WPF, MVVM, dice, 3D depth camera, image processing

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

OBSAH

ÚVOD

S prudkým rozvojem digitálních technologií narůstá potřeba porozumět stávajícím i nově vznikajícím trendům v oblasti vývoje software. Tento vývoj není omezen pouze na oblast mobilních technologií a webových aplikací, ale významně zasahuje i do světa desktopových programů. Digitalizace proniká do všech sfér našeho života, od podnikání a obchodu až po zábavní a herní průmysl. Výjimkou nejsou ani klasické stolní hry, které se čím dál častěji transformují do virtuální podoby. Na tento trend reaguje tato bakalářská práce, která si klade za cíl digitalizaci populární hry Yahtzee.

Navržený systém bude sloužit jako marketingový nástroj pro propagaci vývojového oddělení společnosti Meopta-Optika, s.r.o. na pracovních veletrzích a jiných obchodních akcích. Jeho cílem je zaujmout a aktivně zapojit návštěvníky těchto akcí do interaktivní hry a hravou formou je seznámit s firmou a jejími produkty. Součástí projektu je osvojení práce s hloubkovou kamerou RealSense od společnosti Intel a vytvoření uživatelsky přívětivé knihovny pro snadnou integraci kamery do dalších aplikací. Tato knihovna bude vývojovým oddělením dále upravována a rozvíjena.

První kapitola práce se věnuje analýze současných technologií a architektonických vzorů používaných při vývoji desktopových aplikací, se zaměření na .NET a WPF platformy. Druhá kapitola se soustředí na principy a aplikace hloubkových kamer, včetně popisu technologií snímaní jako je strukturované světlo či měření paralaxy. V praktické částí je popsán návrh a implementace aplikace, která využívá hloubkové kamery pro 3D zpracování obrazu hracích kostek. Závěr práce hodnotí dosažené výsledky a navrhuje možnosti dalšího rozvoje aplikace. Práce propojuje teoretické poznatky s praktickým vývojem aplikace pomocí moderních technologických řešení ve vývoji softwaru.

I. TEORETICKÁ ČÁST

1 VÝVOJ DESKTOPOVÝCH APLIKACÍ

Vývoj desktopových aplikací se zaměřuje na tvorbu softwaru určeného pro běh na stolních a přenosných počítačích. Tyto aplikace jsou zpravidla navrženy tak, aby plně využívaly hardwarové a systémové zdroje daného počítače, díky čemuž mohou nabídnout vysoký výkon a pokročilé uživatelské funkce. Desktopové aplikace mohou rovněž plně a efektivně fungovat „*offline*“. To je užitečné zejména v prostředích s omezeným nebo žádným přístupem k síti. Vzhledem k přímého přístupu k systémovým zdrojům[1] počítače, je v průběhu vývoje desktopových aplikací kladen zvláštní důraz kybernetickou bezpečnost. Často je třeba implementovat sofistikovaná bezpečnostní opatření, včetně šifrování dat a zabezpečení komunikačních protokolů. Metodologie a techniky vývoje desktopových aplikací lze [1] rozdělit do čtyř základních oblasti:

- **Programovací jazyky** – Výběr programovacího jazyka závisí především na požadavcích projektu a jeho cílové platformě. Mezi v současnosti běžně používané jazyky pro vývoj desktopových aplikací můžeme zařadit např. C++, C#, Java a Python.
- **Nástroje a vývojová prostředí** – Pro samotný vývoj desktopových aplikací se používají integrovaná vývojová prostředí (IDE), která poskytují podporu pro kódování, ladění a design uživatelského rozhraní. Zde můžeme jmenovat např. Visual Studio, IntelliJ IDEA nebo QT Creator.
- **Frameworky a knihovny** – Ke zvýšení produktivity vývojáře desktopových aplikací a redukci množství rutinního kódu existuje také celá řada frameworků a specializovaných knihoven. Pro aplikace Windows jde např. o frameworky skupiny .NET. Pro aplikace psané v jazyce JAVA pak o frameworky Swing nebo JavaFX. Pro vývoj multiplatformních aplikací lze využit framework QT.
- **Správa závislostí a verzí** – Pro správu závislostí se často používají nástroje jako Maven, Gradle, nebo NuGet. Verzování kódu řeší např. nástroje GIT nebo MS Azure.

Rozdíl mezi vývojovým prostředím a frameworkem spočívá v jejich účelu a způsobu použití při vývoji softwaru. Framework je soubor knihoven a nástrojů, které poskytují předdefinovanou strukturu souboru funkcí a pravidel pro vývoj softwaru. Můžeme jej chápat

[1] Pod pojmem systémový zdroj rozumíme např. CPU, GPU, systémovou paměť nebo periferie.

jako šablonu nebo „*rámec*“ pro vývoj aplikací. Naopak vývojové prostředí (IDE) slouží k centralizaci všech potřebných vývojových nástrojů[2] do jednoho uživatelského rozhraní. IDE může podporovat vývoj s mnoha různými frameworky. Pojem Framework se tedy týká softwarové architektury a vývojové metodologie, zatímco vývojové prostředí je konkrétní pracovní nástroj. Obecné rozdíly mezi frameworkem a vývojovým prostředím shrnuje tabulka č. 1.

Tabulka 1: Rozdíl mezi Frameworkem a vývojovým prostředím[3]

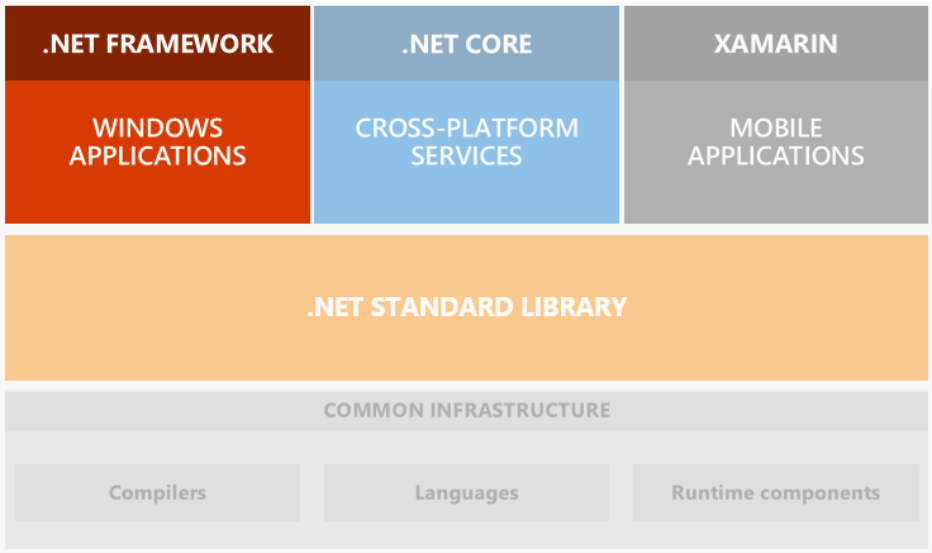
Aspekt	**Vývojové prostředí (IDE)**	**Framework**
Definice	Software pro vývoj kódu	Sada nástrojů a knihoven
Zaměření	Univerzální	Specifické pro jazyk
Komponenty	Editor, debugger, kompilátor...	API, šablony, předdefinované komponenty...
Cíl	Zjednodušení procesu vývoje	Poskytnutí struktury pro aplikace
Rozšiřitelnost	Pluginy	Vlastní komponenty a moduly
Vývojový cyklus	Celý vývojový cyklus	Konkrétní fáze vývoje
Uživatelská interakce	Grafické rozhraní	Programovací rozhraní
Příklady	Visual Studio, Eclipse, IDEA	.NET, Angular, React…

Frameworky často implementují specifické architektonické a návrhové vzory, které jsou optimalizovány pro konkrétní architektonické a návrhové principy. Tyto vzory umožňují vývojáři lépe organizovat a strukturovat kód, minimalizovat riziko vzniku běžných chyb a zvýšit robustnost a udržitelnost softwarových projektů. Příkladem může být .NET framework, který využívá komplexní vícevrstvou architekturu a širokou paletu návrhových vzorů k implementaci svých klíčových funkcionalit. Mezi tyto vzory patří např. návrhový vzor Singleton, který je použit třídou System.Windows.Application k zajištění toho, že aplikace bude mít pouze jednu instanci hlavního okna. V následujících kapitolách se zaměříme na využití .NET frameworku a jeho grafické knihovny WPF pro zpracování obrazu. Dále probereme některé návrhové a architektonické vzory implementované v praktické části této práce.

[2] To zahrnuje např. editor kódu, debugger, kompilátor, interpret, nástroje pro správu verzí a často i nástroje pro vizuální návrh uživatelského rozhraní.
[3] Tato tabulka je pouze obecným přehledem. Některá IDE se mohou zaměřovat na konkrétní jazyky. Některé frameworky naopak nabízí více flexibility apod.

1.1 .NET framework a WPF

.NET framework, vyvinutý společností Microsoft, představuje jeden z vůbec nejrozšířenějších nástrojů pro vývoj desktopových aplikací. V kontextu cílů této práce je klíčová jeho grafická knihovna Windows Presentantion Foundation (WPF), která nabízí rozsáhlou sadu nástrojů pro tvorbu flexibilních uživatelských rozhraní (UI) prostřednictvím jazyka XAML. WPF poskytuje nativní podporu pro multimediální ovládací prvky a díky hluboké integraci s MS Windows umožňuje aplikacím přistupovat k hardwarovým zdrojům. Technologie WPF je postavena na DirectX, což zajišťuje vysoký výkon grafických operací, který je pro zpracování snímků z kamery nezbytný. Vrstvenou architekturu WPF a její vztah k .NET frameworku, ilustruje obr. č. 1. Mezi hlavní nevýhody WPF patří zejména jeho omezená multiplatformní podpora. [2].



Obrázek 1: Vztah WPF k .NET Frameworku [2]

Jazyk XAML je založen na XML (Extensible Markup Language) a funguje na principu hierarchie a vlastností, kde každý element v XAMLu reprezentuje jeden objekt v .NET frameworku a atributy těchto elementů odpovídají vlastnostem objektů. Pro zobrazování snímků z kamery poskytuje uživatelský prvek Image, který umožňuje zobrazovat bitmapové obrázky. Přímou podporu pro připojení ke kameře a získávání snímků WPF ani XAML neposkytuje. Pro pokročilejší funkce zpracování a analýzy obrazu je nutné využít další knihovny a framework. WPF obsahuje základní infrastrukturu pro integraci těchto knihoven do aplikace. Zpravidla je však nutné vytvořit mezivrstvu mezi kamerou a aplikací, která zajišťuje jejich vzájemnou komunikaci. [1, 2].

1.2 Architektonické vzory

Softwarová architektura [1] definuje klíčové koncepty a charakteristiky systému z hlediska je ho struktury, interakcí, prostředí a designových principů. V obecném vnímání zahrnuje uspořádání softwarového sytému, jeho behaviorální komponenty a interakce těchto komponent do komplexnějších subsystémů. Abychom této definici lépe porozuměli, můžeme si softwarový systém představit jako dům. Architektura softwaru tvoří jeho základní konstrukci – rozvržení pokojů, nosné zdi, rozvody elektřiny a další klíčové prvky. Stejně jako v našem modelového příkladu, určuje softwarová architektura vlastnosti daného systému (domu) a ovlivňuje jeho budoucí vývoj.

Architekturou softwaru tedy rozumíme souhrn principů a pravidel, které definují vlastní fungování a strukturu softwarového systému. Implementace vhodné softwarové architektury je klíčovým krokem pro zajištění toho, aby byl daný software schopen zachovat optimální úroveň výkonu a stability, a to i v kontextu integrace nových funkcionalit a technologií. Na rozdíl od návrhových vzorů, které jsou v čase relativně stabilní, se softwarové architektury neustále vyvíjejí a adaptují na nové požadavky a technologické trendy. V posledních několika letech jsme byli v oblasti softwarové architektury svědky značných posunů, a to zejména díky rostoucímu významu cloudových služeb a webových aplikací, což přineslo významné změny také v prostředí desktopových aplikací. Tyto změny vedly k postupnému odklonu od tradičních monolitických struktur ve prospěch více modularizovaných, flexibilních a multiplatformních přístupů. [1, 3].

Vysoký počet softwarových architektur reflektuje různorodost potřeb a specifik vývoje softwaru. Některé softwarové architektury lze použít obecně, jiné slouží pro konkrétní účely. Softwarové architektury [3] je pro větší přehlednost možné efektivně klasifikovat na základě různých parametrů jako např. typ zaměření, strukturální aspekty, etapy životního cyklu softwaru nebo mechanismy komunikace. Vzhledem k omezenému rozsahu této práce se v následujícím výčtu budeme věnovat pouze těm specifickým architektonickým vzorům, které jsou určeny pro vývoj UI s využitím WPF. V tomto omezení patří z hlediska technické praxe mezi v současnosti nejpoužívanější architektury Model-View-ViewModel, Model-View-Presenter, Event-Driven Architecture, a Domain-Driven Design.

1.1.1 Architektura Model-View-Presenter

Architektonický vzor Model-View-Presenter (MVP) vychází ze staršího vzoru Model-View-Controller (MVC[4]), který je tradičně více spojován spíše s vývojem webových aplikací. Stejně jako většina moderních architektur podporuje MWP rozdělení aplikace na jednotlivé moduly/vrstvy s jasně definovanými úlohami. Na rozdíl od MVC architektury poskytuje MVP architektura striktnější oddělení těchto modulů a usnadňuje tak testování a údržbu jednotlivých komponent softwaru. Zmíněnými moduly rozumíme zpravidla oddělení logiky aplikace od uživatelského rozhraní (UI) a jejich propojení. Jak už název architektury napovídá v případě vzoru MVP se jedná o [1,3]:

- **Model** – Sem patří veškerá business logika, data aplikace a pravidla pro jejich zpracování.
- **View** – Slouží k prezentaci uživatelských dat skrze UI. V ideálním případě neobsahuje žádnou logiku pro rozhodování a manipulaci s daty, pouze informuje presenter o uživatelských akcích.
- **Presenter** – Ovládá tok dat mezi Modelem a View. Resp. zprostředkovává jejich vzájemnou komunikaci.

Jelikož desktopové aplikace často vyžadují složitější unit testing a komplexnější uživatelské interakce než aplikace webové, je oddělení business logiky od UI a zabránění jejich prolínaní pro vývoj těchto aplikací klíčové. Za nevýhodu MVP architektury lze považovat její složitost, která může v některých případech vést k nadbytečnému kódu v presenteru. Striktní implementaci vzoru MVP pro desktopové aplikace požaduje např. framework Caliburn.Micro. WPF a ostatní výše zmíněné frameworky přímou implementaci vzoru MVP nepodporují, ale lze je k němu pomocí vlastních tříd a rozhraní přizpůsobit.

1.1.2 Architektura Model-View-ViewModel

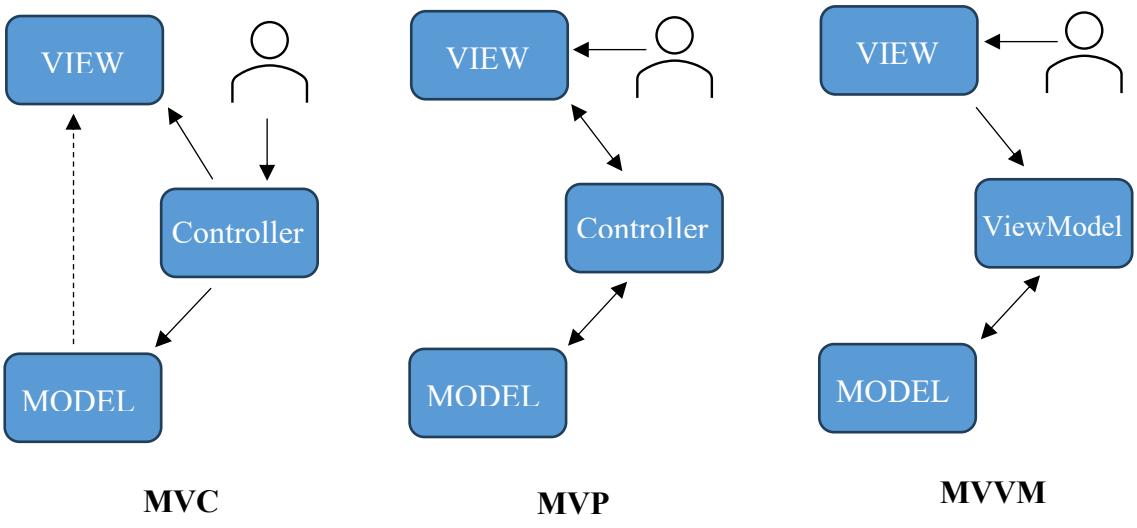
Architektonický vzor MVVM byl původně navržený pro MS Silverlight a následně implementován právě v platformě Windows Presentation Foundation (WPF). Jedná se o variantu výše zmíněných architektur, která je vhodná zvláště pro ty desktopové frameworky, které nativně podporují dvoucestnou datovou vazbu (two-way data binding). Mezi tyto frameworky můžeme vedle WPF zařadit i některé další frameworky z MS ekosystému jako

---

[4]V případě MVC architektury má prezentační vrstva View aktivnější roli. Může přímo komunikovat s Modelem, což vede k těsnějšímu provázání těchto komponent.

například Universal Windows platform (UWP) nebo Prism. Pro multiplatformní vývoj s moderními technologiemi a podporou MVVM může být rozumnou volbou například Framework Avalonia. Mezi multiplatformní frameworky, které architekturu MVVM nativně nepodporují, ale jsou „*ochotné*“ k její manuální implementaci, lze zařadit např. frameworky QT nebo Electron[5] . [4].

Největší rozdíl mezi MVP a MVVM architekturou je ten, že MVVM namísto modulu presenter zavádí nový koncept ViewModel. Tento modul můžeme chápat jako abstrakcí vrstvy uživatelského rozhraní, která umožňuje automatickou a obousměrnou datovou vazbu mezi View a Modelem. To znamená, že ViewModel již obsahuje určitý stav a vnitřní logiku, která slouží pro transformaci dat z Modelu do formátu, který lze snadno zobrazit ve View a naopak. Díky tomuto uspořádání, umožňuje MVVM ještě o něco snazší testování a údržbu aplikace, jelikož Viewmodel lze na rozdíl od Presenteru testovat nezávisle na View. Nevýhodou MVVM oproti architektuře MVP je obvykle složitější UI a větší nároky na prezentační logiku aplikace. Z řečeného vyplývá, že MVP architektura je rovněž o něco univerzálnější a lze ji implementovat v široké škále technologií. MVVM je více vázáno na frameworky, které ho nativně podporují. Rozdíly v architekturách MVC, MVP a MVVM včetně zobrazení závislostí jednotlivých komponent ilustruje obr. č. 2. [1, 4].



Obrázek 2: Softwarové architektury

[5] Electron je open-source framework, který umožňuje vytvářet nativní desktopové aplikace pomocí webových technologií jako HTML, CSS a JavaScript. Aplikace pak fungují jako webové stránky, ale mají přístup k nativním funkcím operačního systému.

1.1.3 Even driven architektura (EDA)

EDA [5] je komplexí architektonický styl, ve kterém se ústředním bodem pro komunikaci a spolupráci mezi komponentami stávají události. Událost je typicky asynchronní[6] zpráva, která spotřebitelům signalizuje, že se v systému něco stalo. To mohou být například změny stavu jednotlivých komponent softwaru, transakce, chyby apod. Jednotlivé komponenty v této architektuře pak jednoduše vystupují jako buďto producenti nebo spotřebitelé těchto událostí. Jako producenty událostí označujeme ty komponenty, které událost generují a publikují ji do systému. Spotřebitelé událostí události odebírají a určitým způsobem na ně reagují. Většina moderních desktopových frameworků dnes vzor EDA využívá a poskytuje vývojáři nástroje, pomocí nichž lze tento vzor efektivně implementovat. Například framework QT implementuje vzor EDA v podobě svého vlastního mechanismu signálů a slotů. WPF podporuje EDA prostřednictvím mechanismů jako Data Binding a Event Handlers. QT implementuje vzor EDA v podobě svého vlastního mechanismu signálů a slotů.

EDA je možné s úspěchem kombinovat s výše zmíněnou architekturou MVVM. Obě architektury jsou do určité míry komplementární. Zatímco MVVM se soustředí na strukturu UI, EDA řídí asynchronní a event-driven aspekty aplikace. Správným použitím EDA v kombinaci s MVVM je možné dosáhnout ještě vyššího stupně oddělení UI od business logiky a testovatelnosti aplikace. A to jednoduše proto, že událostmi řízené modely umožňují jednotlivým ViewModelům a dalším komponentám systému vzájemně komunikovat bez přímé závislostí. Právě tento způsob komunikace mezi Modelem a ViewModelem je použit i v praktické části této práce, což mimo jiné vede i k plynulejšímu a reaktivnějšímu uživatelskému zážitku. Nevýhodou použití EDA spolu s MVVM je potřeba důkladného pochopení obou architektur a zvýšená komplexnost návrhu a implementace.

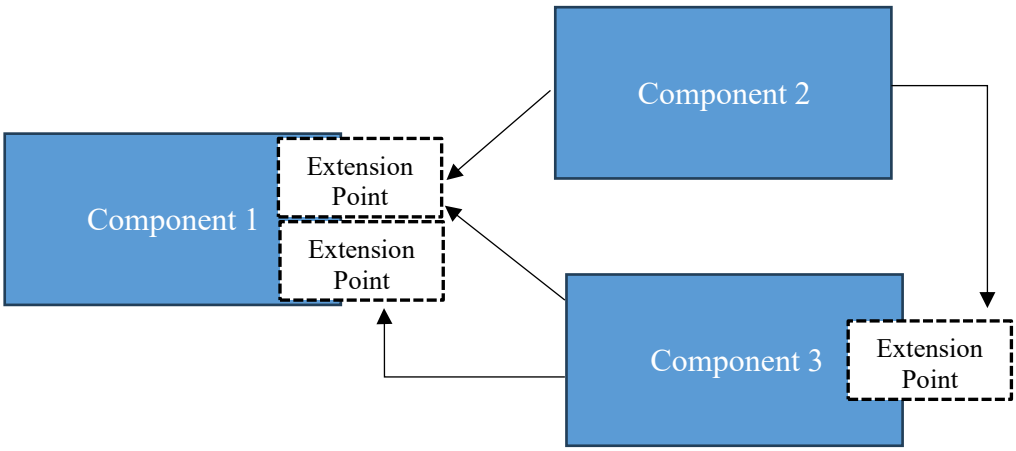
1.1.4 Component-Based Design

Posledním architektonickým vzorem, který si v této kapitole představíme je tzv. Component-Based Desing (CBD). CBD definuje [4] jako přístup k vývoji softwaru, který zdůrazňuje opakované použití softwarových komponent. Tento přístup umožňuje vývojářům sestavit aplikace z předem definovaných komponent, což může výrazně zkrátit dobu vývoje

[6] Slovo asynchronní v tomto kontextu znamená, že producent po odeslání zprávy nečeká na reakci spotřebitele, ale pokračuje dál ve své činnosti.

a zlepšit kvalitu softwaru. To je do značné míry podobné definici objektu z oblíbeného programovacího paradigmatu Object-Oriented programming (OOP). A stejně tak jako objekty, umožňují i komponenty lepší modularitu, standardizaci, snadnou rozšiřitelnost a opakované použití kódu. Ten asi největší rozdíl mezi objektem a komponentou je v jejich zaměření. Zatímco v OOP jsou objekty instancemi tříd a důraz je zde kladen především na jejich vzájemné vztahy, CBD staví na myšlence vytváření softwaru pomocí předem definovaných komponent a důraz klade na jejich interakci Z tohoto pohledu lze komponenty uvažovat za nezávislejší a více standardizované než objekty v OOP. OOP techniky mohou pomoci vytvořit jednotlivé komponenty, zatímco CBD poskytuje rámec pro jejich sestavování do fungujících aplikací. [1, 3].

Framework WPF sám o sobě koncept COP přímo neimplementuje, ale silně podporuje jeho principy a poskytuje nástroje pro jeho využití. Tím je v daném kontextu myšlena především podpora uživatelských ovládacích prvků, stylů a šablon. WPF nabízí celou řadu ovládacích prvků, které lze považovat za základní stavební kameny komponent. Tyto prvky je možné libovolně vnořovat a vytvářet tak komponenty s vlastní logikou a prezentací. V praktické části práce se jedná o např. komponenty IconButton nebo YahtzeeInteractiveRow. Šablony se využívají k definování vzhledu a chování těchto komponent. Mohou obsahovat vazby na data a případně i další rozšiřující komponenty. Možnou konstrukci jedné takové komponenty ilustruje obr. č. 3.



Obrázek 3: Komponenta

1.2 Návrhové vzory

V kapitole 1.1 jsme definovali pojem architektura softwaru. Na rozdíl od softwarové architektury, která se zaměřuje na celkovou strukturu a koncepci softwarového systému, se návrhový vzor zabývá řešením specifických a opakujících se problémů v rámci návrhu menších komponent, kterými mohou být například moduly a třídy. V naší analogii se návrhové vzory zaměřují na detaily konstrukce a specifikují, jak budou jednotlivé komponenty domu, jako jsou okna, dveře nebo schody navrženy a vyrobeny. Ačkoli návrhové vzory obvykle nepovažujeme za součást softwarové architektury, mohou svým působením na mikroúrovni návrhu k celkové struktuře softwaru přispívat. Návrhové vzory [5] představují osvědčené a standardizované přístupy k designovým úkolům, čímž přispívají především k:

- **Zvýšení opakovatelnosti kódu** – Jednou implementovaný vzor lze snadno znovu použit v dalších částech systému.
- **Zlepšení čitelnosti kódu** – Použití známých vzorů usnadňuje pochopení struktury a fungování kódu.
- **Zvýšení udržovatelnosti kódu** – Díky struktuře a modularitě dané vzorem je snazší kód upravovat a opravovat.

Návrhové vzory tak můžeme chápat jako jakési šablony, které lze aplikovat na běžné problémy v rámci konkrétního kontextu. Nejedná o konkrétní implementaci či fragment kódu, ale spíše o obecný a flexibilní přístup k softwarovému řešení daného problému, který je třeba specificky upravit pro využití v tom kterém konkrétním projektu. Důležité je pochopit základní principy návrhových vzorů a vědět, kdy a jak je efektivně použít. Před implementací návrhového vzoru by měly být zváženy jednodušší alternativy k dosažení stejných cílů. Návrhové vzory by se měly použít pouze tam, kde to má z pohledu vývojáře skutečný význam. Jejich nesprávné použití může vést k celé řadě problémů a snížit celkovou kvalitu softwarového projektu.

Stejně jako v případě softwarových architektur existuje i velmi mnoho různých návrhových vzorů. Asi nejznámější skupinou návrhových vzorů jsou vzory patřící pod tzv. GoF[7]. Tato skupina představuje 23 návrhových vzorů rozdělených do tří kategorií, které se

[7] GOF (Gang of Four) je skupina autorů, kteří v roce 1994 publikovali knihu „Design Patterns: Elements fo Reussable Object-Oriented Software [5].

staly základem OOP. Ačkoli je většina GoF vzorů je pro současné systémové inženýrství stále široce využívána[8], v průběhu let byly představeny některé modernější a efektivnější řešení vyvinuty v rámci individuálních programátorských týmů či velkých technologických komunit. Ve světě .NET to jsou např. návrhové vzory Repository nebo Unit of Work. Pro účely praktického použití v rámci této bakalářské práce se v následujících kapitolách zaměříme pouze na některé příklady těch návrhových vzorů, které jsou relevantní pro UI a MVVM architekturu. Výše zmíněné rozdíly mezi architektonickými a návrhovými vzory ilustruje tabulka č. 2. [5].

Tabulka 2: Rozdíl mezi architektonickými a návrhovými vzory

Aspekt	**Architektonický vzor**	**Návrhový vzor**
Účel	Struktura celého systému/Aplikace	Řešení konkrétních problémů
Rozsah	Makro-úroveň	Mikro-úroveň
Zaměření	Interakce mezi komponentami	Interakce mezi objekty
Implementace	Technologicky závislá	Technologicky nezávislá
Cíl	Škálovatelnost, udržitelnost…	Jednoduchost, opětovná použitelnost
Vztah	Struktura pro návrhové vzory	Použití v rámci architektury
Příklady	MVC, MVVM, EDA, DDM	Factory, Observer, Singleton

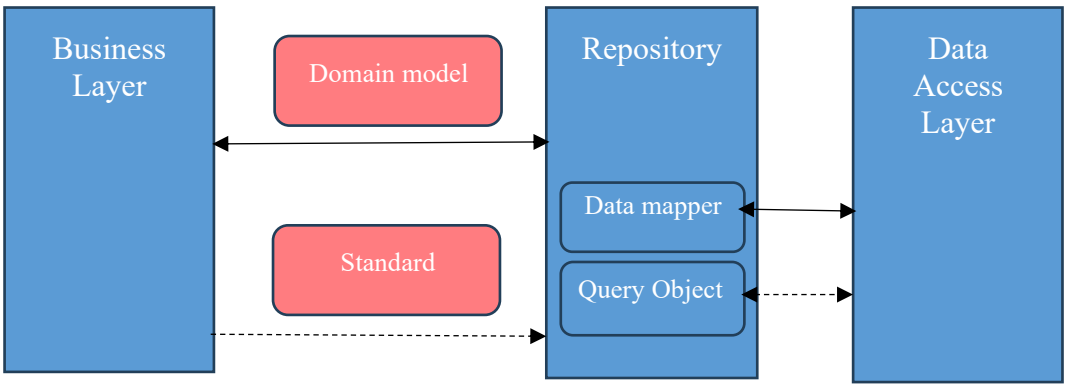
1.2.1 Repository Pattern

Prvním představeným návrhovým vzorem v rámci této práce bude tzv. Repository Pattern. Tento návrhový vzor bývá zpravidla implementován v rámci softwarové filozofie DDD (Domain-Driven Desing), která se soustředí na pochopení a následném modelování tzv. domény. Domenou v tomto pohledu rozumíme specifický problém, na který se software zaměřuje. Modelem takové domény jsou pak v OOP doménové objekty, které reprezentují jednotlivé entity, jejich vztahy a pravidla řízení. K oddělení doménové logiky od externích technologií určených pro specifické datové operace s doménovými objekty (UI, Databázová systém…) používá DDD vrstevnaté architektury (viz kap. 1). Myšlenka návrhové vzoru Repository pak slouží ke zjednodušení doménové logiky tím, že poskytuje abstrakci pro přístup k datům doménového modelu. Jednotlivé doménová modely tak zůstávají nezávislé

[8] Za všechny lze jmenovat např. návrhové vzory Factory, Singleton nebo Decorator.

na konkrétním mechanismu zpracování dat a umožňují vývojářům vytvářet složité domény bez nutnosti řešit technické detaily práce s daty. [3].

V kontextu představené aplikace je návrhový vzor Repository implementován jako rozhraní, kterým jsou data ukládána a načítána z databázového systému. Toto rozhraní definuje základní CRUD[9] operace, které jsou prováděny prostřednictvím jednoduchých metod daného jazyka C#, aniž by se zabývaly problematikou databázového systému obecně. Infrastrukturu pro implementaci repozitáře v rámci této aplikace a celého světa .NET vůbec poskytuje ORM (Object-Relational Mapping) nástroj Entity Framework (EF), a to pomocí třídy DbContext. Ve složitějších aplikacích je často požadováno, aby byla datová operace provedena přes více doménových modelů, resp. entit. V tom případě mohou být dvě nebo více operací z konkrétních repozitářů propojeny do jedné transakce v návrhovém vzoru Unit of Work. To mimo jiné zajistí, aby při selhaní operace na jakékoli entitě došlo k selhání celé transakce. Myšlenkový koncept návrhového vzoru Repository shrnuje obr. č. 4.



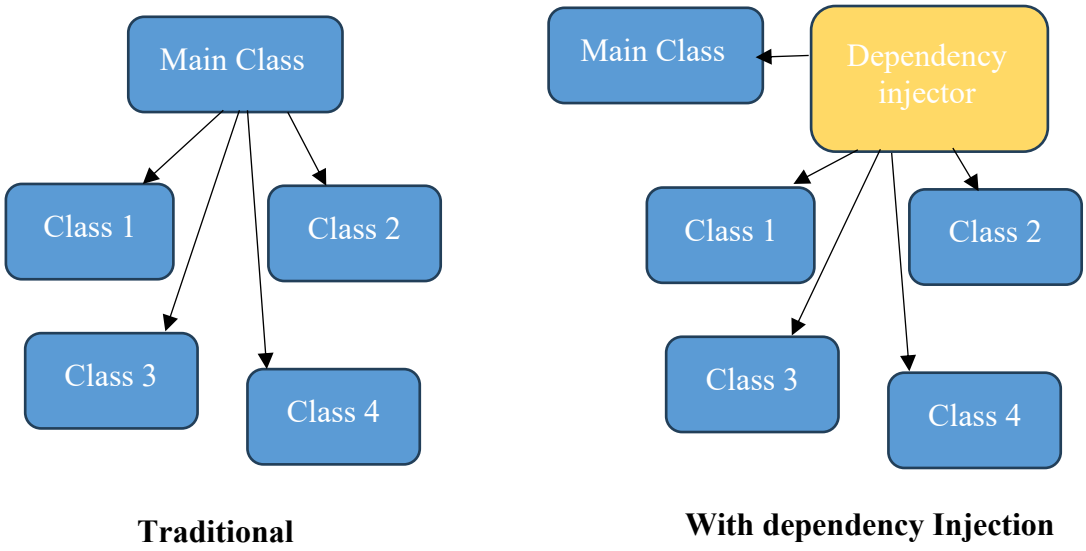
Obrázek 4: Repository Pattern

1.2.2 Dependency Injection

Pro hlubší pochopení smyslu existence návrhového vzoru dependency injection (DI) je potřeba definovat nejprve princip IoC (Inversion of control). Ačkoli spolu tyto dva pojmy souvisí, jsou často zaměňovány a mylně interpretovány. IoC je programovací princip, jehož cílem je snížení závislostí mezi komponentami softwaru. Jeho koncept představili [6] ve svém článku *Designing Reusable Class*. Zde uvádějí, že jednou z nejdůležitějších vlastností aplikačního frameworku je to, že metody definované uživatelem pro přizpůsobení daného frameworku budou často volány ne z uživatelova aplikačního kódu, ale z frameworku

[9] Jedná se o akronym čtyř základních operací pro manipulaci s daty. Těmi jsou Create, Read, Update a Delete.

samotného. Tento princip tedy odstraňuje potřebu toho, aby si komponenty softwaru samy vyráběly a vyhledávaly své závislosti. Namísto toho jsou jim tyto závislosti poskytovány zvnějšku nějakým externím systémem. Návrhový vzor DI je jedna možných implementací[10] představeného principu. Jedná se o konkrétní techniku, pomocí které lze dosáhnout cíle IoC. DI poskytuje praktický způsob, jak tento princip implementovat v kódu. K tomu obvykle využívá tří hlavních metod, kterými jsou konstruktorová, setterová a interfaceová injektáž. Rozdíl mezi klasickým obraceným řízením životního cyklu komponent ilustruje obr. č. 5.



Obrázek 5: Dependency Injection

Návrhový vzor DI se nejčastěji implementuje pomocí tzv. IoC kontejnerů. Tyto kontejnery představují nástroje nebo frameworky, které automaticky vytvářejí a poskytují instance tříd, které jsou potřebné jako závislosti pro jiné třídy. Umožňují spravovat životní cyklus závislosti, jejich konfiguraci a samotnou injektáž. Ve WPF a desktopových aplikací obecně je použití DI méně běžné než u např. aplikací webových. Z toho důvodu WPF plnohodnotný IoC kontejner nenabízí. K dispozici je nicméně velké množství IoC kontejnerů třetích stran. Mezi ty nejpopulárnější a nejvíc komplexní lze zařadit např. Autofac či Ninject. V praktické části práce se pro implementaci návrhového vzoru DI využívá knihovna DependencyInjection z balíku Microsoft.Extensions, která je nativně podporována a integrována s frameworkem ASP.NET Core.

[10] Za další implementace IoC můžeme považovat např. návrhové vzory Service Locator nebo Factory, představená v kap. 1.3.4.

1.2.3 Factory Pattern

V úvodu této kapitoly byla zmíněna skupina návrhových vzorů GOF. Do této skupiny patří i Factory pattern, který je v rámci jejího vnitřního členění klasifikován jako tzv. Creational Pattern. Creational patterns se zaměřují na abstrakci a flexibilitu v procesu instancování objektů. Vedle Factory sem patří ještě návrhové vzory singleton, builder a prototyp. Myšlenka návrhové vzoru Factory spočívá v delegování odpovědnosti za proces instancování objektů na jinou třídu, resp. na její speciální „*tovární*“ metodu. Tato třída (*továrna*) umožňuje vracet instance různých tříd, které implementují stejné rozhraní nebo stejnou abstraktní třídu. Díky tomu je v rámci nadřazené třídy možné vytvářet objekty bez specifikace konkrétních tříd, čímž se snižuje pevná vazba mezi třídou požadující objekt a třídami objektů, které jsou vytvářeny. GOF [5] definuje několik typů Factory pattern, které se liší v závislosti na tom, jak je rozhodováno o typu vytvářeného objektu. Jsou to:

- **Factory method –** Jedná se o základní typ Factory patternu. Továrna implementuje metodu Create, která vrací objekt instanci požadovaného typu objektu. Typ objektu je určen parametrem metody.
- **Simple Factory –** Jedná se o zjednodušený typ Factory patternu. Továrna poskytuje statické metody pro každý typ objektu.
- **Abstract Factory –** Továrna implementuje abstraktní metodu Create, která vrací abstraktní objekt. Konkrétní továrny pak z této továrny dědí a implementují metodu Create pro specifický typ objektu.

Factory Pattern je univerzální návrhový vzor, který se dá použít v široké škále programovacích jazyků a frameworků. V architektuře MVVM může být použit pro např. dynamické vytváření ViewModelů na základě určitých podmínek a konfigurací. To může být užitečné v případě, že je ViewModel závislý na specifických datech. Často se používá také ve spojení s technikami IoC pro vytváření objektů a služeb, které ViewModely ke své činnosti využívají. V kontextu desktopových, a zvláště pak WPF aplikací, se návrhový vzor factory stává mocným nástrojem pro dynamické vytváření komponent uživatelského rozhraní. Oddělení logiky objektů od jejich používání umožňuje centralizovat a zjednodušit proces tvorby UI komponent a zároveň zachovat vysokou míru flexibility a volnosti v jejich používání bez porušení principů MVVM.

1.2.4 Command Pattern

Druhým z návrhových vzorů skupiny GoF použitých v praktické části této bakalářské práce je tzv. Command Pattern, který spadá do kategorie behaviorálních návrhových vzorů. Tyto návrhové vzory se soustředí na spolupráci a efektivní komunikaci mezi jednotlivými objekty aplikace. Command pattern zjednodušuje komunikaci mezi objekty tím, že rozlišuje mezi definicí specifického požadavku na provedení akce a jeho implementací. Toho je dosaženo tím, že se požadavky zapouzdří do samostatných objektů, které obsahují veškeré informace k potřebné pro provedení toho kterého požadavku. Takto zapouzdřený požadavek se následně předává tzv. *invokeru*, jehož úkolem je aktivovat požadavek a následně jej po uskutečnění nějaké uživatelské akce předat jednotlivým vykonavatelům (*receivers*). [3, 5].

V architektuře MVVM je použití Command Patternu zvlášť prospěšné. Umožňuje ViewModelu reagovat na akce uživatelů v UI bez nutnosti přímé komunikace s View. Toto oddělení logické a prezentační vrstvy aplikace představuje základní princip architektury MVVM tak, jak je uvedeno v kapitole 1.2.3. V rámci WPF frameworku je vzor Command Pattern (na rozdíl od výše zmíněných vzorů) nativně podporován prostřednictvím rozhraní ICommand. Rozhraní ICommand má dvě základní metody, kterými jsou Execute(object parameter) a CanExecute(object parameter). Metoda Execute provádí samotnou logiku příkazu a metoda CanExecute určuje, zda je možné příkaz v daném okamžiku spustit. Pro zjednodušení práce s příkazy je implementace ICommand v praktické části této bakalářské práce realizována pomocí RelayCommand („*příkaz*“). Ačkoli tato implementace není součásti standardní knihovny WPF, u menších projektů se z důvodu minimalizace množství kódu často využívá. RelayCommand umožňuje předat logiku Execute a CanExecute pomocí lambda výrazů nebo delegátů, což eliminuje potřebu definovat samostatné třídy pro každou uživatelskou akci.

2 ZPRACOVÁNÍ OBRAZU V POČÍTAČI

Zpracování obrazu (ZPO) v počítači lze definovat jako interdisciplinární oblast informatiky a počítačového vidění, která se zabývá analýzou a interpretací vizuálních informací zaznamenaných digitálními obrazovými senzory. Jedná o komplexní a technicky náročný obor, který vyžaduje hluboké porozumění jak matematickým a algoritmickým základům, tak i jejich nástrojům a praktickým aplikacím. V současné chvíli je dostupná je celá řada technik ZPO, z nichž většina je založena na analýze obrazových pixelů[11]. Pixel představuje nejmenší možnou jednotku informace v obraze, která je navenek reprezentována určitou číselnou hodnotou (viz kapitola 2.1). Samotný digitální obraz je tedy v podstatě dvojrozměrnou maticí těchto pixelů, kde každý z nich nese informaci o malé části zaznamenané scény. Vedle zjevných vlastností jako je barva a intenzity/jasu[12] světla má pixel ještě několik dalších pro zpracování obrazu důležitých vlastností. Mezi tyto vlastnosti můžeme [7] zařadit:

- **Poloha** – Umístění pixelu v obraze pomocí souřadnici (x, y).
- **Rozlišení** – Počet pixelů v obraze (např. 1920 x 1080)
- **Průhlednost** – Míra schopnosti prolínání pixelu s jinými objekty v obraze.
- **Kontrast** – Rozdíl jasu mezi sousedními pixely.

Vzhledem k obrovské šíři technik ZPO není na prostoru této práce možné, věnovat se každé z nich detailně. Z toho důvodu stručně zmíníme pouze ty, které přímo souvisí s detekcí a analýzou hracích kostek. V daném případě je prvním a základním krokem tzv. segmentace obrazu, která spočívá v rozdělení obrazu na oblasti se společnými vlastnostmi (na jednotlivé hrací kostky). K tomu lze použít řadu dostupných technik jako je např. prahování, detekce hran nebo shlukování. Dalším krokem je rozpoznání vzoru na povrchu kostky. Tzn. hledání specifických znaků (čísel nebo teček), které hrací kostku jednoznačně identifikují. K tomu se používají techniky jako je např. filtrování či morfologické operace. V případě použití 3D kamery pro snímání obrazu je navíc možné využít hloubkové informace pro 3D rekonstrukci

---

[11] Z anglického „*picture element*“ (prvek obrázku).

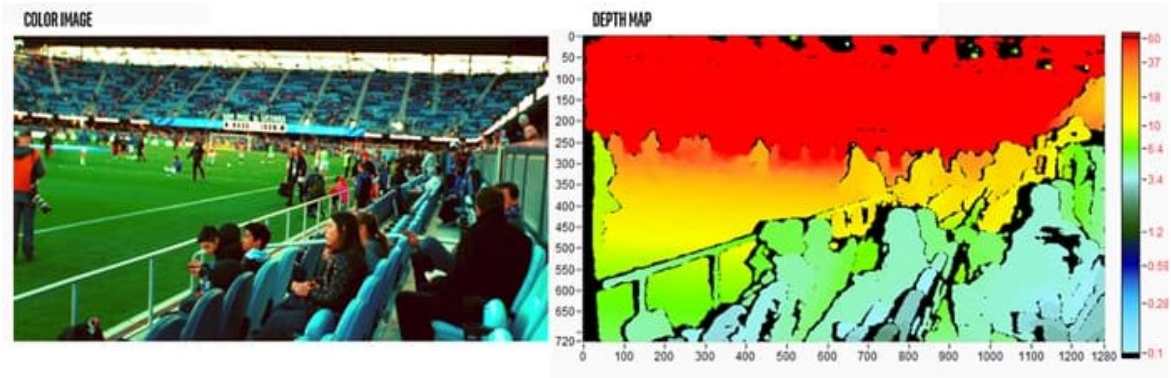
[12] V praxi se pojmy jas a intenzita používají synonymně. V kontextu zpracování obrazu se však jedná o dva odlišné pojmy. Intenzitou rozumíme objektivní měřítko síly světla. Jedná se o fyzikální veličinu, která se obvykle měří v jednotkách W/m^2. V případě jasu jde o subjektivní charakteristiku, která závisí na lidském oku a vnímaných barvách.

scény, což umožňuje detekci hracích kostek v různých polohách a orientacích, čímž se zvyšuje robustnost a spolehlivost detekčního systému. [8].

2.1 Standardní vs Hloubkové kamery

Abychom mohli přistoupit k aplikaci některé z výše zmíněných technik zpracování obrazu, je nejprve potřeba nějakým způsobem získat digitální obrazová data, resp. hodnoty jednotlivých pixelů. K tomu účelu se používají různé druhy snímacích zařízení, které transformují optický signál ze scény do digitálního formátu. Jedním z nejběžnějších typů snímacích zařízení jsou digitální kamery. Standardní digitální kamery fungují na principu zachycení definovaného spektra vlnových délek elektromagnetického spektra pomocí objektivu a jeho přeměny na elektrický signál. O tuto přeměnu se starají obrazové snímače, které jsou tvořeny z milionů malých světlocitlivých prvků zvaných fotodiody. Každá z těchto fotodiod odpovídá jednomu pixelu v digitálním obraze. Intenzita a barva světla, které dopadá na každou fotodiodu, určuje konkrétní hodnotu daného pixelu. [9].

V případě standardních digitálních kamer je tato hodnota typicky vyjádřena pomocí kombinace tří složek určitého barveného modelu. Pro např. 24bitovou kameru a barevný model RGB jde o kombinaci červené, zelené a modré barvy, kde každá barva může nabývat hodnot intenzity v rozsahu 0-255. Takže např. černá barvu bude reprezentována hodnotou (0,0,0) a jasně červená hodnotou (255, 0 0). To umožňuje reprezentaci přes 16 milionů různých barevných odstínů v jednom pixelu. Hloubkové kamery pracují oproti standardním digitálním kamerám s jednou dimenzí navíc – hloubkou scénou, tj. vzdálenost scény od objektivu kamery. Pixel hloubkové kamery je tedy určen celkem čtyřmi hodnotami. V případě použití barevného modelu RGB půjde o systém RGBD. Rozdíl mezi barevným a hloubkovým snímkem ilustruje obr. č. 6. Výstup z hloubkové kamery je zobrazen v hloubkové mapě, kde každá barva představuje určitou vzdálenost od objektivu kamery. [10].



Obrázek 6: Rozdíl mezi barevným a hloubkovým snímkem [9]

Hloubkové kamery zachycují 3D informace o scéně, díky čemuž je možné měřit vzdálenosti objektů, vytvářet 3D mapy a modely, rozlišovat objekty v popředí a pozadí a implementovat efekty rozšířené reality. S novým rozměrem zároveň dramaticky stoupá i variabilita pixelů, čímž se významně zlepšují možnosti strojového vidění. Mezi nevýhody hloubkových kamer oproti standardním kamerám můžeme zařadit jejich vyšší cenu, typicky nižší rozlišení a velkou náročnost na hardwarové a softwarové zdroje. Výkon hloubkových kamer může být ovlivněn extrémními světelnými podmínkami nebo interakcí s průhlednými či velmi reflektujícími povrchy. [9].

2.2 Hloubkové kamery Intel REALSENSE

Intel RealSense je řada softwarových a hardwarových produktů od společnosti Intel, které se zaměřují na počítačové vidění a hloubkové snímání. Softwarovou složku tohoto ekosystému tvoří RealSense SDK, který poskytuje balíčky a knihovny kompatibilní s .NET Frameworkem pro vývoj aplikací na Windows 10 (v 64bitové verzi) a Ubuntu 18.04 (64bit). Pro práci s RealSense technologií v C# aplikacích na platformě .NET slouží knihovna RealSense.NET, která nabízí vyšší úroveň abstrakce a zjednodušuje práci se základní knihovnou *librealsense*[13] prostřednictvím .NET wrapperů. Hardwarové komponenty Intel RealSense zahrnují [11]:

- Stereoskopické kamery
- Kamery se strukturovaným a kódovaným světlem
- Time-of-Flight (ToF) kamery a LiDAR senzory

2.1.1 Hloubkové kamery se strukturovaným světlem

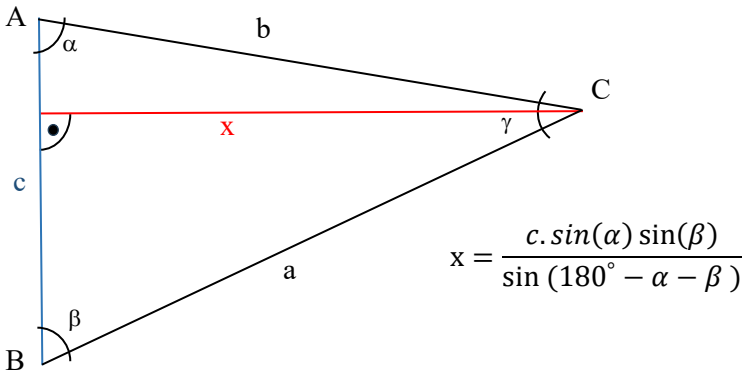
Hloubkové kamery se strukturovaným světlem používají k mapování 3D prostoru aktivní 3D projekci. Na scénu promítají předem definovaný vzor světla (obvykle pruhy, body nebo vlnovky)[14] a analyzují jeho deformaci na snímaném obrazu. Například, je-li na sférickou plochu promítnut vzor ve formě pruhů, budou se tyto pruhy podle povrchu sféry specificky ohýbat a deformovat. Jestli se nyní začne sféra pohybovat směrem od nebo k emitoru vzorovaného světla, změní se také promítnutý vzor. Využitím tohoto rozdílu mezi

---

[13] Knihovna *librealsense*, poskytuje základní funkcionalitu pro přímou interakci s hardwarovými komponentami kamer.

[14] Projektované světlo může být vzorované i časově. Případně lze použít i kombinaci obou – pruhy v pravidelných časových intervalech mění svou geometrii. Princip výpočtu zůstává stejný.

očekávaným a skutečně zaznamenaným obrazem kamery je pomocí znalosti geometrie projekce a trigonometrických výpočtů možné pro každý jeden pixel vypočítat vzdálenost od kamery. Princip výpočtu je podobný triangulaci (obr. č. 7), kde se ze známých úhlů a vzdáleností mezi známými body odvozuje poloha bodu neznámého. [10].



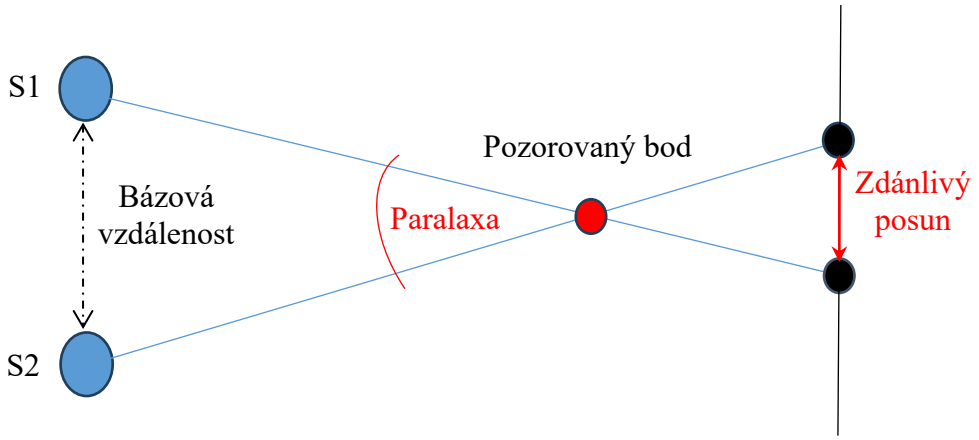
Obrázek 7: Obecný trojúhelník – triangulace

Výhody a nevýhody této techniky shrnuje příručka k technologii hloubkového snímání od společnosti Intel [9]. V kapitole "*Structured light and coded light*" je uvedeno, že díky řízenému osvětlení funguje technika strukturovaného světla spolehlivě i v tmavém prostředí a není ovlivněna barvou snímaného povrchu. Detailní analýzou deformovaného vzoru lze v závislosti na intenzitě vyzařovaného světla dosáhnout relativně vysoké přesnosti měření. Na druhou stranu, kamery se strukturovaným světlem bývají dražší a kvůli nutnosti integrovat projektor světla také objemnější a náročnější na kalibraci.

2.1.2 Stereoskopické kamery

Stereoskopická kamera funguje víceméně analogicky k binokulárnímu vidění člověka. Kamera je vybavena dvěma optickými senzory (objektivy), které jsou umístěny vedle sebe v malé, přesně definované bázové vzdálenosti. Tato konfigurace umožňuje zachycení dvojice obrazu téže scény z mírně odlišných perspektiv. Softwarové algoritmy tyto dva snímky porovnají a identifikují mezi nimi paralaktické[15] rozdíly. Na základně geometrických vztahů mezi bázovou vzdáleností senzorů a jednotlivými disparitami obrazu je následně odvozená hloubka jednotlivých bodů na scéně [10]. Princip určování hloubky objektu pomocí paralaxy ilustruje obr. č. 8.

[15] Paralaxou rozumíme zdánlivou změnu pozice objektu v závislosti na pozici pozorovatele.



Obrázek 8: Princip určování vzdálenosti objektu pomocí paralaxy.

Dva pozorovací body S1 a S2 na obr. 8 odpovídají dvěma objektivům stereoskopické kamery. Při pozorování bodu z S1 a S2 dochází k rozdílu v detekované pozici objektu vůči pozadí (černá čára), což je na obrázku označeno jako „*Zdánlivý posun*“. Tento posun je měřítkem pro výpočet vzdálenosti bodu od kamery. Pohybuje-li se pozorovaný objekt směrem ke kameře, zvětší se i rozdíl ve zdánlivém posunu obou snímků (větší paralaxa) a naopak. Jestliže necháme pozorovaný bod na místě a začneme zvyšovat bázovou vzdálenost, bude se úměrně tomu zvětšovat i paralaktický úhel. Díky tomu jsme schopni detekovat menší změny vzdálenosti – kamera bude mít vyšší rozlišení. Nicméně, v technické praxi je pro hloubkové kamery z důvodu omezených prostorových podmínek často nevhodné nebo nepraktické velké bázové vzdálenosti zavádět.

Stereometrické kamery jsou oproti kamerám se strukturovaným světlem jednodušší a kompaktnější. Ke své činnosti nepotřebují externí projektory a díky podobnosti principu funkce k lidským očím mohou působit intuitivněji. Na rozdíl od kamer se strukturovaným světlem nejsou tolik citlivé na rušení v prostředí z dalších kamer a jiných zářičů emitujících IR záření. Jejich nevýhodou je fakt, že mají v porovnání s kamerami se strukturovaným světlem nižší přesnost a mohou selhávat v oblastech s nižším kontrastem nebo opakujícími se vzory, kde může docházet k problémům s nalezením odpovídajících dvojic bodů v obou snímcích. [9, 10].

2.1.3 Time of flight a LIDAR kamery

Všechny typy hloubkových kamer využívají k extrapolaci třetího rozměru (hloubce) určité, předem známé informace. U např. stereoskopických kamer to byla bázová vzdálenost mezi senzory. U kamer se strukturovaným světlem je dopředu znám vizuální tvar

emitovaného světelného svazku. V případě Time of Flight (ToF) je touto konstantou rychlost světla. Všechny typy ToF kamer vysílají určitý druh světla, snímají jím danou scénu a měří čas, kdy se světlo vrátí zpět k senzoru. U LiDAR (Light Detection and Ranging) kamer je toto světlo monochromatické a koherentní[16], díky čemuž mohou Lidar senzory v závislosti na výkonu W[V] a vlnové délce λ[m] měřit hloubku na významné vzdálenosti. Hlavní nevýhodou tohoto typu kamer je jejich náchylnost na rušení. Jakákoli situace, kdy na senzor dopadá světlo o dané λ vyzářené třetím zdrojem, může snížit kvalitu a rozpoznatelnost výsledného obrazu. [9].

[16] Má stejnou rovinu kmitání, frekvenci a rozdíl fází.

II. PRAKTICKÁ ČÁST

3 APLIKACE DICEMASTER3600

Softwarové řešení aplikace "*DiceMaster3600*" představuje komplexní systém rozdělený do čtyř projektů, které reflektují požadavky multi-vrstvé architektury. Hlavní projekt tvoří prezentační vrstvu. Nacházejí se zde definice uživatelských komponent, styly jednotlivých komponent a přidružené zdroje jako je např. reference na korporátní písma a barvy. V rámci tohoto projektu je implementován koncept MVVM (viz kapitola 1.1.2). Druhý projekt "*DiceMaster3600.Core*" představuje aplikační jádro a definuje obchodní logiku systému. Obsahuje Data Transfer Objects (DTOs), které zajišťují přenos dat mezi jednotlivými vrstvami aplikace bez nutnosti odhalovat vnitřní strukturu entit. Součástí tohoto projektu jsou rovněž soubory enumerátorů a definice rozhraní, které zajišťují typovou bezpečnost a usnadňují sdílení funkcionality napříč celou aplikací. Datovou vrstvu zastupuje projekt "*DiceMaster3600.Data*", který je zaměřen na perzistenci a manipulaci s daty. Obsahuje entity reprezentující datové modely v databázi a jednotlivé repozitáře (kap. 1.2.1) pro přístup k těmto datům. Poslední projekt „*DiceMaster3600.Devices"* implementuje logiku komunikace s kamerou a poskytuje základní abstrakci pro zpracování kamerových snímků.

Aplikace DiceMaster3600 využívá několik knihoven třetích stran, které jsou dostupné a spravované balíčkovacím systémem NuGet. NuGet usnadňuje instalaci, aktualizaci a správu knihoven v .NET projektech. Všechny použité knihovny jsou open-source, dostupné pod licencemi umožňující volné šíření. Další informace o NuGetu a licencích open-source knihoven lze nalézt na oficiálním vzdělávacím portálu společnosti Microsoft [11]. V rámci aplikace bylo mimo jiné využito např. knihoven:

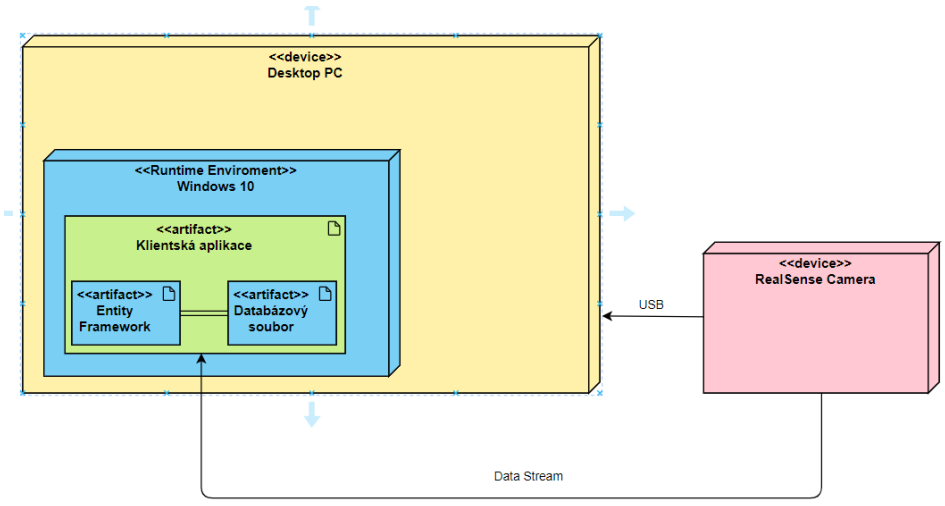
- **EntityFrameworkCore** – Implementace ORM frameworku pro .NET
- **Serilog –** Knihovna pro logování v .NET aplikacích
- **Microsoft.Extension.Hosting –** Podpora pro Dependency Injection (DI) a lifecycle management v .NET Core aplikacích.
- **CommunityToolkit.MVVM –** Framework poskytující sadu nástrojů pro implementaci MVVM.
- **Intel.RealSenseWithNativeDll –** Multiplatformní knihovna pro hloubkové kamery IntelRealse řady D400 a SR300.

V následujících kapitolách budou jednotlivé projekty představeny v rámci kontextu celkové architektury systému. Jelikož úplné pokrytí všech implementačních detailů aplikace přesahuje možnosti této práce, budou okomentovány pouze ty vybrané úseky kódu, které

nejlépe ilustrují dané koncepty a implementují architektonické a návrhové vzory představené v teoretické části. Pozornost bude věnována rovněž popisu use-case scénářů a definici funkčních a nefunkčních požadavků, které stály u základu návrhu a implementace daných modulů. Celý zdrojový kód aplikace je k dispozici na přiloženém CD.

3.1 Analýza a specifikace požadavků

Ve fázi návrhu a plánování softwarového projektu je důležité pečlivě analyzovat a zdokumentovat, jakým způsobem bude software fungovat a jaké kvalitativní parametry musí splňovat. Výsledky této analýzy formují základ, na kterém je celý sw projekt postaven, a přímo ovlivňují jak jeho technické řešení, tak i „*obchodní*“ výsledky. K tomu [12] uvádí analogii, přirovnávající softwarové inženýrství k architektuře: „*podobně jako architekt nemůže stavět dům bez plánu, ani vývojáři by se neměli pustit do kódování bez jasné představy o tom, co má systém dělat*“. Specifikaci požadavků na systém lze považovat za most mezi potřebami uživatelů a technickým provedením softwaru. V rámci této specifikace se podrobně zaměříme na dva klíčové aspekty: funkční a nefunkční požadavky a případy užití, které tyto požadavky ilustrují. Fyzickou architekturu systému ilustruje obr. č. 9.



Obrázek 9: Diagram nasazení

3.1.1 Funkční a nefunkční požadavky

Funkční požadavky jsou klíčovým elementem specifikace softwarového systému. Definují jeho chování a funkce v reakci na uživatelské akce a externí podněty. Odpovídají na otázku, co má systém dělat a jak v daných situacích reagovat. Pro účely této práce byly funkční požadavky rozděleny do samostatných kategorií dle jejich základních funkcí a

charakteristik. Výčet všech funkčních požadavků obsahuje příloha P. I. Pro názornou ilustraci funkčních požadavků si uvedeme dva příklady:

1. REGISTRACE UŽIVATELE
    - RQ1: Umožnit uživatelům vytvořit účet zadáním jména, e-mailu a hesla
    - RQ2: Ověřit zda je e-mailová adresa unikátní
    - RQ3: Zobrazit notifikaci o úspěšné/neúspěšné registraci
    - RQ4: Po úspěšné notifikaci zobrazit okno pro přihlášení
2. STATISTICKÉ VYHODNOCENÍ
    - RQ5: Počítat a zobrazovat distribuci pravděpodobností výsledků hodu kostkami
    - RQ6: Analyzovat úspěšnost hráčů podle fakulty
    - RQ7: Zobrazit úspěšnost hráčů podle genderu
    - RQ8: Vyhodnocovat a zobrazovat statistiky tří nejlepších hráčů

Nefunkční požadavky se týkají vlastností systému, jako je výkon, spolehlivost, použitelnost apod. [12]. I tyto požadavky jsou, podobně jako požadavky funkční, roztříděny do kategorií dle specifické funkční oblasti. Kompletní seznam nefunkčních požadavků obsahuje příloha P. II. Pro ilustraci opět uvedeme dva příklady:

1. BEZPEČNOST
    - RQ16: Zašifrovat hesla uživatelů při ukládání do databáze.
    - RQ17: Zajistit ochranu osobních údajů v souladu s GDPR.
    - RQ18: Implementovat bezpečné přihlašování a komunikaci mezi klientem a serverem.
2. VÝKON
    - RQ19: Optimalizace zpracování dat z 3D kamery pro minimální latenci
    - RQ20: Zajištění rychlého načítání hrací plochy a hladké interakce

3.1.2 Případy užití

Pro efektivní mapování a implementaci funkčních požadavků[17] do softwarového systému byly vypracovány případy užití (anglicky „Use Cases“), které modelují interakce

[17] Případy užití se primárně zaměřují na popis interakcí systém-uživatel (funkční požadavky). Nefunkční požadavky se v krocích případů užití obtížně modelují. Pro práci s nefunkčními požadavky existují vhodnější nástroje jako NFR frameworky a modely vlastností a chování [13].

mezi tzv. aktéry a softwarovým systémem. Případy užití jsou strukturované specifikace, jež krok za krokem popisují jednotlivé uživatelské scénáře – od identifikace aktéru, kteří se systémem komunikují, až po podrobný popis toku akcí. Počet aktérů v rámci jednoho případu užití se může lišit v závislosti na specifickém scénáři. Zpravidla jsou definování nejméně dva aktéři, a to aktéři primární a sekundární. Primární aktér iniciuje daný proces, zatímco sekundární aktér má možnost tento proces ovlivňovat nebo k němu přispívat. Každý případ užití se zaměřuje na jeden specifický úkol, který může aktér/uživatel realizovat, a detailně demonstruje jeho implementaci v rámci celého systému. Scénáře pokrývající případy užití navrhované aplikace obsahuje příloha P. III. Jako názorný přiklad je níže (Tab. č. 3) uveden popis případu užití registrace uživatele. [12, 13].

Tabulka 3: Případ užití – Registrace uživatele

Název: Registrace uživatele		
ID: UC01		
Charakteristika: Vytvoření nového účtu		
Primární aktér: Nový uživatel		
Vedlejší aktér: Žádný		
Vstupní podmínky: Uživatel není přihlášen		
Výstupní podmínky: Do systému je uložen nový uživatel. Uživatel je informován o výsledku registrace.		
Hlavní scénář:		
Krok	Aktér/Systém	Popis
1	Uživatel	Uživatel zvolí možnost registrace
2	Sytém	Systém zobrazí registrační formulář
3	Uživatel	Uživatel vyplní registrační formulář
4	Sytém	Systém ověří, zda je e-mail unikátní
5	Sytém	Databázový server uloží nového uživatele
6	Systém	Systém zobrazí notifikaci o úspěšné registraci
7	Systém	Systém zobrazí okno pro přihlášení
8	Sytém	Případ užití končí
Alternativní scénáře: UC01a – Registrovaný email není unikátní UC02b – Chyba při vyplňování registračního formuláře.		

3.2 DiceMaster3600.Devices

Projekt „*DiceMaster3600.Devices*“ představuje dynamickou knihovnu (DLL), určenou pro integraci a řízení funkcí kamer Intel RealSense. Základní abstrakci pro zpracování snímků z kamery tvoří třída BaseFrameProcess<T>, která je specializací rozhraní IFrameProcess. Klíčovou metodou této třídy je ProcessFrameAsync(), která umožňuje asynchronní zpracování datových rámců („*FrameSet*“) získaných z kamery. Po dokončení zpracování snímku je jeho výsledek uložen do vlastnosti Result. Jsou-li přítomní nějací posluchači, spustí tato akce událost OnResultChanged (kap. 1.1.3), která slouží k upozornění ostatních částí aplikace na nově zpracovaný snímek. O registraci, resp. odregistraci procesu se stará provider v Projektu „*DiceMaster3600*“. Bázovou třídu BaseFrameProcess<T> ilustruje obr. č. 9.

```
public abstract class BaseFrameProcess<T> : IResultFrameProcess<T>
{
    private T result;

    public T Result
    {
        get => result;
        private set
        {
            result = value;
            OnResultChanged?.Invoke();
        }
    }

    protected abstract T ProcessFrame(FrameSet frameSet);

    public async Task ProcessFrameAsync(FrameSet frameSet)
    {
        Result = await Task.Run(() => ProcessFrame(frameSet));
    }

    public event Action? OnResultChanged;
}
```

Obrázek 10: Třída BaseFrameProcess<T>

Správu připojení a zpracování dat z kamer Intel Realse řeší třída RealSenseCamera prostřednictvím implementace rozhraní ICamera, které rozšiřuje základní funkcionalitu rozhraní IDevice. Pro řízení životního cyklu kamery využívá asynchronní metody ConnectAsync() a DisconnectAsync(), které inicializují, resp. zastavují datové proudy. Zásadním prvkem třídy je kontinuální „*while*“ smyčka v metodě ProcessFrameAsync(), která v nekonečném cyklu čeká na zpracování nových snímků. Synchronizace přístupu ke zdrojům je řešena pomocí „*SemaphoreSlim*“, díky čemuž je možné spravovat konkurenční

přístup ke zpracování snímků bez rizika deadlocku nebo ztráty dat. Asynchronní operace zpracovávání snímků jsou prováděny v izolovaných úlohách („*TASK*“), což snižuje zátěž hlavního vlákna a zvyšuje celkovou responzivitu aplikace.

Metodu ProcessFrameAsync() zobrazuje obr. 10. V metodě se nejprve synchronně čeká na příchod nového snímku prostřednictvím příkazu pipeline.WaitForFrames(). Jakmile snímek dorazí, je uložen do proměnné a pokusí se získat semafor. Pokud je semafor získán, tj. žádný jiný snímek není právě zpracováván, zahájí se asynchronní zpracování nově příchozího snímku. Metoda ProcessFrameAsync() prochází seznamem registrovaných procesů, které na získaný snímek/rámec asynchronně aplikují svou logiku. Po zpracování obrazu jsou posluchačům poskytnuty nejnovější snímky k dalšímu použití (např. zobrazení v UI). Jakékoli snímky příchozí během zpracovávání aktuálního snímku jsou zahozeny.

```
        private async Task ProcessFramesAsync(CancellationToken token)
        {
            try
            {
                while (!token.IsCancellationRequested)
                {
                    var frameset = pipeline.WaitForFrames();
                    frameset.Keep();

                    if (await processingSemaphore.WaitAsync(0, token))
                    {
                        _ = Task.Run(async () =>
                        {
                            try
                            {
                                var bitmap = await ProcessFrameAsync(frameset);
                                OnNewFrame?.Invoke(this, bitmap);
                            }
                            finally
                            {
                                frameset.Dispose();
                                processingSemaphore.Release();
                            }
                        }, token);
                    }
                }
            }
            catch (OperationCanceledException)
            {
                AppLogger.Information($"{DateTime.Now}
                        : Processing frames has been canceled in");
            }
            catch (Exception ex)
            {
                AppLogger.Error($"Error in processing frames: {ex.Message}");
            }
        }
```

Obrázek 11: Metoda ProcessFramesAsync

3.2.1 DiceRecognitionProcess

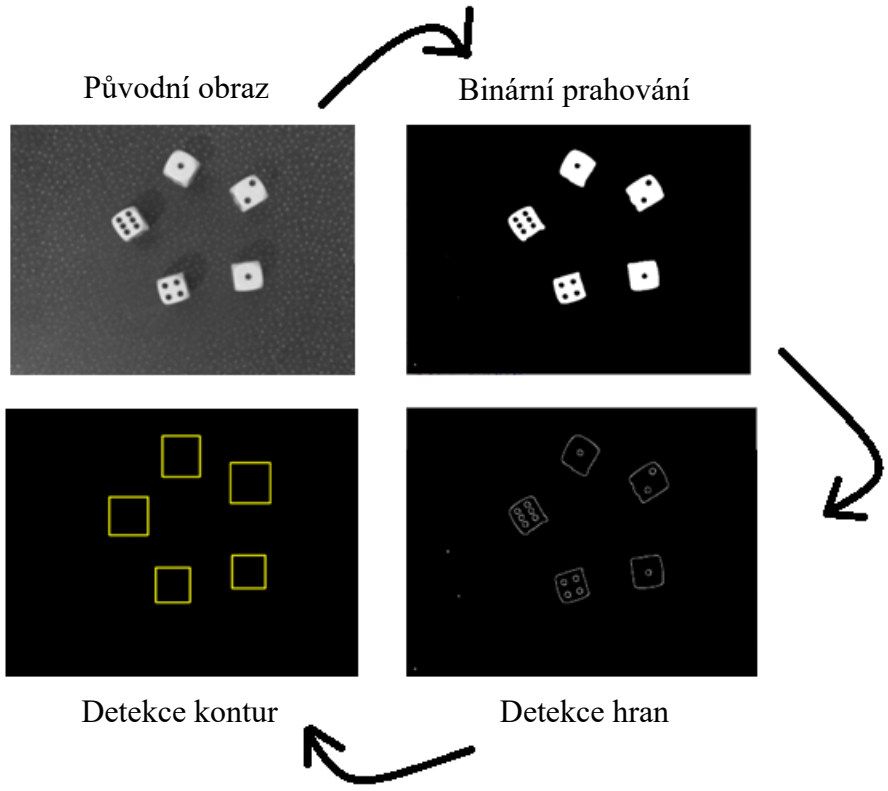
Konkrétní implementací bázové třídy BaseFrameProcess<T> je např. proces DiceRecognition. Tento proces je v aplikaci použit pro rozpoznání kostky, resp. pro určení počtu teček na každé kostce umístěné na herním plánu. Třída přijímá konfigurační objekt, který definuje parametry jako prahové hodnoty intenzity nebo minimální přípustný poměr mezi výškou a délkou kostky. Na začátku procesu jsou video rámce („*Frames*“) převedeny do formátu, který umožňuje přímou manipulaci s jednotlivými pixely (viz kap. 2). K tomu účelu je v knihovně implementován obecný konvertor ConvertToImage<Tcolor, TDepth> (obr. č. 11), resp. jeho konkrétní specifikace pro barevný rámec ve formátu Image<Bgr, byte> a ConverterDepthFrameToImage pro hloubkové rámce ve formátu Image<Gray, ushort>, kde každý pixel nese hodnotu hloubky.

```
public static Image<TColor, TDepth> ConvertToImage<TColor, TDepth>(VideoFrame f)
    where TColor : struct, IColor
    where TDepth : new()
{
    try
    {
        int width = frame.Width;
        int height = frame.Height;

        TDepth[] data = new TDepth[frame.Stride * height];
        frame.CopyTo(data);

        Image<TColor, TDepth> image = new(width, height);
        Buffer.BlockCopy(data, 0, image.Data, 0, data.Length);

        return image;
    }
    catch (Exception ex)
    {
        AppLogger.Error($"Failed to convert depth frame to image : {ex}");
        throw new InvalidOperationException(
                        $"Failed to convert depth frame to  image: {ex}");
    }
}
```

Obrázek 12: Konverze VideoFrame - Image

Samotný detekční proces je rozdělen do dvou kroků. V prvním kroku jsou detekovány jednotlivé hrací kostky. Nejprve je snímek převedený na šedo-tónový obraz. Tento krok redukuje složitost obrazu tím, že eliminuje nepotřebné barevné informace. Poté je na snímek aplikován Gaussovský filtr SmoothGaussian(kernelSize) pro redukci šumu. Šum může ovlivnit detekci hran, proto je důležité obraz nejprve vyhladit. Následuje binární prahování, které zjednodušuje rozpoznávání objektů tím, že transformuje obraz do formátu s bílými objekty a černým pozadím. Nakonec jsou pomocí Cannyho detektoru hran

Canny(min, max) získány obrysy objektů. Z nich jsou vypočteny ohraničující obdélníky, které identifikují možné kostky. Pro každou detekovanou kostku se navíc vypočítá průměrná hloubka v oblasti kostky a zobrazí se jako text na výsledném obrazu. K tomu slouží metoda CalcuclateAverageDepth, která bere v úvahu pouze pixely s platnými hloubkovými daty. Postup detekce ilustruje obr. č. 13.



Obrázek 13: Detekce kostek

Druhým krokem procesu detekce teček na kostkách. Pro každou kostku se nejprve vytvoří výřez pomocí GetSubRect(die). Izolace kostky umožní, zaměřit se na detekci teček bez rušení z okolního obrazu. Přesný výřez zajišťuje, že se následné zpracování aplikuje přímo na obraz s tečkami. Na izolované oblasti kostky se použije inverzní binární prahování – tečky na kostkách jsou typicky tmavší než světlejší pozadí. Tento krok transformuje tmavé tečky na bílé oblasti, což usnadňuje jejich následnou detekci pomocí technik pro vyhledávání kontur. S využitím funkce CvInvoke.FindCountours(), která prohledá oblast a detekuje uzavřené oblasti (bílé objekty na černém pozadí), jsou nakonec detekovány kontury těchto oblastí. Pro každou detekovanou konturu je použito několik filtrů, včetně kruhovitosti a očekávaných rozměrů. Zdrojový kód popsaného algoritmu je obsahem přílohy P. IV.

3.3 DiceMaster3600.Data

V úvodu kapitoly bylo řečeno, že projekt *DiceMaster3600.Data* reprezentuje datovou vrstvu aplikace. Tato vrstva je vnitřně strukturovaná do několika částí tak, aby reflektovala různé aspekty práce s daty a podporovala snadnou rozšiřitelnost a údržbu systému. Klíčovou částí je soubor SqlEfDataContext.cs, který je implementací DbContextu z Entity Framework. SqlEfDataContext spravuje databázové spojení a datové modely pro jednotlivé entity. Entity samotné reprezentují strukturu dat a vztahy v databázi. Jsou určeny a primárně využívány pro mapování databázových tabulek na objekty pomocí techniky ORM. Abstrakci nad datovým přístupem k jednotlivým datovým modelům poskytují konkrétní repozitáře, jejichž účel a princip funkce byl popsán v kapitole 1.2.1. Poslední částí projektu je soubor RepositoryAdapter.cs, který implementuje návrhový vzor Adaptér[18] a slouží pro komunikaci mezi repozitáři a ostatními částmi aplikace.

3.3.1 User Repository

Všechny implementace repozitářů v aplikaci jsou odvozeny od abstraktní třídy Repository, která je parametrizována generickým typem „*TEntity*“. To umožňuje její využití s různými typy entit bez potřeby duplicity kódu. V konstruktoru prostřednictvím DI přijímá implementaci kontextu z Entity Frameworku a poskytuje sadu asynchronních metod pro CRUD operace nad databází. Za zmínku stojí metoda DeleteAsync(), která podporuje tzv. koncept „*soft delete*“ prostřednictvím rozhraní ISoftDeleteEntity (obr. 14). Tato technika umožňuje označit data jako smazaná, aniž by byla fyzicky odstraněna z databáze. V praxi to obvykle funguje tak, že má entita v databázi přidán dodatečný sloupec, který definovaným způsobem označuje, že byla entita z pohledu uživatele smazána. V aplikaci DiceMaster3600 se do tohoto sloupce ukládá datum smazání entity. Třída dále implementuje rozhraní IDisposable, což umožňuje správu zdrojů a uvolnění prostředků kontextu o dokončení operací. [14]

[18] Návrhový vzor adaptér (někdy též Wrapper) nebyl v teoretické části aplikace zmíněn. Jedná se o velmi jednoduchý návrhový vzor, který umožňuje spolupráci objektů s nekompatibilními rozhraními.

```
public abstract class Repository<TEntity> : IDisposable where TEntity : class
{
     protected readonly SqlEFDataContext context;
     public Repository(SqlEFDataContext context) => this.context = context;

     public async Task InsertAsync(TEntity entity)
     {
         await context.Set<TEntity>().AddAsync(entity);
         await context.SaveChangesAsync();
     }

     public async Task DeleteAsync(TEntity entity)
     {
         if (entity is ISoftDeleteEntity softDeleteEntity)
         {
             softDeleteEntity.DeletedDate = DateTime.UtcNow;
         }
         else
         {
             context.Set<TEntity>().Remove(entity);
         }

         await context.SaveChangesAsync();
      }

      //...Ostatní CRUD operace a metody pro správu entit

      public void Dispose() => context.Dispose();
  }
```

Obrázek 14: Implementace návrhového vzoru Repository

Konkrétní implementací abstraktní třídy Repository je např. třída „*UserRepository*“ (viz příloha P. V), která se zaměřuje na specifické potřeby správy uživatelů v databázi. Třída UserRepository rozšiřuje základní funkcionalitu generického repozitáře o několik dalších asynchronních metod. Těmito metodami jsou:

- **DeleteByFacultyIdAsync(int facultyID) –** Tato metoda umožňuje hromadný soft delete uživatelů podle identifikátoru fakulty.
- **GetUserByEmailAsync(string email, string plainPassword) –** Tato metoda slouží k autentizaci uživatele. Vyhledává konkrétního uživatele podle e-mailové adresy a následně ověřuje zadané heslo pomocí hashovací funkce BCrypt[19]
- **AddUserAsync(UserDTO user, string plaiPassword, int facultyID) –** Tato metoda přidává nového uživatele do databáze. Přijímá uživatelská data ve formátu DTO a heslo v „*plain*“ formě, které před uložením do databáze hashuje pomocí BCrypt.

[19] BCrypt je bezpečný hashovací algoritmus založený na Blowfish šifře.

3.3.2 RepositoryFasade

Třída RepositoryFasade tvoří jakýsi „*most*“ mezi datovou a prezentační vrstvou aplikace. Z hlediska vnitřní funkcionality slouží jako prostředník mezi různými repozitáři, zajišťuje jejich agregaci a navenek poskytuje jednotné a srozumitelné rozhraní pro práci s daty. Repository adaptér podporuje tzv. lazy loading, díky čemuž se instance repozitářů vytvářejí až v momentě, kdy jsou potřeba. Díky tomuto přístupu mohou metody (obr. 15) v RepostoryFasade implementovat návrhový vzor Unit of Work (viz kapitola 1.2.1), kde se v rámci jedné transakce provádějí multiple databázové operace, což zajišťuje atomičnost a konzistenci změn. Selhání takové metody umožňuje reverzi všech operací prostřednictvím Rollback, čímž se zabraňuje nekonzistenci dat.

```
public async Task DeleteUniversityById(int UniversityID)
{
    using var transaction = repos.BeginTransaction();
    try
    {
       var university = repos.UniversityRepository.GetById(UniversityID)
             ?? throw new KeyNotFoundException($"University with ID
                {UniversityID} was not found.");

       foreach (var faculty in university.Faculties.ToList())
       {
           await repos.UserRepository.DeleteAllByFacultyIdAsync(faculty.Id);
           await repos.FacultyRepository.DeleteAsync(faculty);
       }

       await repos.UniversityRepository.DeleteAsync(university);
       await transaction.CommitAsync();
    }
    catch (Exception ex)
    {
         await transaction.Rollback();
         throw new ApplicationException($"Failed to delete university with ID
               {UniversityID}: {ex.Message}", ex);
    }
 }
```

Obrázek 15: Implementace Unity of work

3.4 DiceMaster3600

Struktura hlavního projektu „*DiceMaster3600*“ je navržena podle architektury MVVM (kap. 1.1.2). Složka Models obsahuje jádro aplikační logiky a služby, jako např. DataAccesManager pro správu přístupu k datům nebo IMessageService, která poskytuje rozhraní pro zasílání zpráv. Složka Views představuje prezentační vrstvu aplikace. Obsahuje .xaml soubory definující vizuální obrazovky aplikace (příloha P. VI) a vlastní uživatelské

komponenty, resp. jejich styly a šablony. Součástí View je také několik konvertorů, které transformují data mezi vrstvami. Složka ViewModels obsahuje konkrétní viewmodely, které propojují aplikační model s prezentační vrstvu. Patří sem abstraktní třída NotifyViewModel nebo třídy HomeViewModel a LoginFormViewModel pro správu specifických obrazovek. Vstupním bodem aplikace je soubor App.xaml, tento soubor nastavuje DI kontejner a další vstupní konfigurace.

Mimo výše popsanou strukturu obsahuje projekt také továrnu (kap. 1.2.3) na vytváření jednotlivých viewmodelů, které jsou následně zasazovány do hlavní obrazovky aplikace. Tato továrna/třída není přímou součástí žádné z vrstev MVVM architektury. Implementace takovéto továrny demonstruje flexibilitu MVVM. Tento přístup umožňuje bezproblémové rozšíření MVVM o doplňkové mechanismy, aniž by byly porušeny její základní principy. Samotná továrna využívá službu z aplikace pro získání instancí, které implementují rozhraní IMenuConrolViewModel. Pokud se služba nepodaří získat, je vyvolána metoda HandlError, která zaznamená chybu do logu a vyhodí výjimku s informativním hlášením. Implementaci továrny ilustruje obr. 16.

```
public class ViewModelFactory : IViewModelFactory
{
    public T CreateViewModel<T>() where T : class, IMenuControlViewModel
    {
        return App.AppHost!.Services?.GetService<T>()
            ?? HandleError<T>();
    }

    private static T HandleError<T>() where T : class
    {
        var errorMessage = $"ViewModel of type {typeof(T).Name}
                             could not be  created.";
        AppLogger.Error(errorMessage);
        throw new InvalidOperationException(errorMessage);
    }
}
```

Obrázek 16: ViewModelFacotry

V následujících podkapitolách bude na praktickém příkladu demonstrován postup implementace návrhového vzoru MVVM. Pro ukázku byl vybrán relativně jednoduchý formulář pro registrování nového uživatele. Model má na starosti správu uživatelských dat a validaci vstupů. Součástí modelu je i služba pro notifikace, která umožňuje zobrazování vyskakovacích oken při chybných vstupech a úspěšných akcích. V části věnované komponentám View a ViewModel budou stručně popsány pouze některé zajímavé implementační detaily, které by měly přispět k celkovému pochopení konceptu MVVM. To

mimo jiné zahrnuje způsob, jakým ViewModel využívá veřejné vlastnosti k synchronní aktualizaci View, a jak struktura MVVM přispívá k udržitelné architektuře softwaru.

3.4.1 Model

Model registračního formuláře představuje generická třída NotificationService a třída DataAccesManager, která zajišťuje přístup k DTO aplikace. Jednotlivé datové operace ve třídě DataAccesManager volají privátní metodu ExecuteDataOperationAsync, která je obaluje a nastavuje příznak IsWorking. Tento příznak prostřednictvím volání události informuje viewmodel o tom, zda právě probíhá nějaká akce.

Třída NotificationService, implementuje rozhraní IMessageService a výše zmíněné služby pro správu notifikací ve formě zpráv. Zajišťuje možnost odloženého zpracování notifikací (mezi např. dvěma formuláři) pomocí fronty, která uchovává zprávy a umožňuje je zobrazit později. Vedle toho třída udržuje privátní slovník notifications, který propojuje různé typy notifikací reprezentované kontextem s příslušnými akcemi. Tyto akce („*callbacky*“) jsou spuštěny při zpracování požadované notifikace. Implementace třídy NotificationService je uvedena v příloze P. VII. Klíčové metody třídy jsou:

- **Subscribe:** Přidá callback k danému typu notifikace a uchová ho ve slovníku.
- **Unsubscribe:** Odstraní všechny callbacky pro daný typ notifikace.
- **TryNotify:** Spustí všechny callbacky pro daný typ notifikace a předá jim zprávu.
- **NotifyImmediately:** Zařadí zprávu do fronty k okamžitému zobrazení.

3.4.2 ViewModel

Hlavní ViewModel registračního formuláře dědí ze dvou bázových ViewModelů. Prvním z nich je *NotifyViewModel*, který poskytuje mechanismus pro zasílání notifikací prostřednictvím služby *IMessageService*. Tím druhým je *ReactiveViewModel*, který přidává možnost dynamické správy závislých příkazů. To je důležité proto, že jednoduché nastavení SetProperty ze třídy ObservableObject[20] pouze nastavuje hodnotu vlastnosti a vyvolává událost PropertyChanged. Nezahrnuje žádnou další logiku pro aktualizaci příkazů, které jsou na této vlastnosti závislé.

[20] Knihovna CommunityToolkit.Mvvm.Input z Nuget balíčku.

ViewModel EntryFormViewModel obsahuje osobní a institucionální údaje uživatele, které jsou bindovány k UI. Pokud se některá z hodnot těchto vlastní změní, dojde k aktivaci metody buďto SetProces nebo SetTrigger(), které informují pozorovatele o změně – dojde k aktualizaci příslušných prvků ve View. Mimo tyto vlastnosti obsahuje ViewModel taky několik příkazů. Mezi tyto příkazy patří např. příkaz SaveCommand, který zpracovává dva delegáty ve formě lambda funkcí. Jeden pro provedení příkazu. Druhý pro určení toho, zda může být příkaz proveden (kap. 1.2.4). Pokud je registrace uživatele úspěšná, dojde k vyvolání notifikace RegitrationSucces registrované formulářem LoginForm a spuštění RequestClose pro uzavření formuláře EntryFormViewModel. Po uzavření formuláře jsou uvolněny všechny zdroje a odhlášeny všechny sledované notifikace, aby nedošlo k jejich zbytečnému hromadění. Registrační ViewModel je součástí přílohy P. VIII.

```
public abstract class ReactiveViewModel : NotifyViewModel
{
    protected ReactiveViewModel(IMessageService messageService)
              : base(messageService){ }

    protected bool SetTrigger<T>(ref T storage, T value,
                  [CallerMemberName] string? propertyName = null)
    {
        if (SetProperty(ref storage, value, propertyName))
        {
            RefreshCommand();
            return true;
        }
        return false;
    }

    abstract public void RefreshCommand();
}
```

Obrázek 17: ReactiveViewModel

3.4.3 View

Obrazovka (View) registračního formuláře je navržena jako komponenta Window s vlastními zdroji a ovládacími prvky („*Controls*“). Mezi zdroje („*Resources*“) patří např. konvertory pro mapování hodnot na barvy, poskytovatelé dat pro výčty („*enum*“) nebo lokální styly pro konzistentní vzhled prvků. Hlavním prvkem zobrazení je MainGrid, který organizuje ostatní ovládací prvky do dvou sloupců a několika řad. Data jsou vázána přímo na vlastnostmi a příkazy v ReactiveViewModelu pomocí konceptu Bindings. Mimo MainGrid leží ovládací prvek SnackBar z knihovny Microsoft.MaterialDesingThems.WPF, která zobrazuje zprávy z fronty ve ViewModelu. Pro zjednodušení .xaml kódu a znovupoužitelnost napříč celou aplikací byly vedle nativních ovládacích prvků použity

vlastní uživatelské komponenty („*User Controls*“) dostupné skrze jmenný prostor Controls. Mezi ně patři například kontrolka PasswordToggleControl, která poskytuje UI pro zadávání hesla s možností přepínání mezi viditelným a skrytým režimem. Tato komponenta názorně demonstruje používání stylů a konvertorů pro efektivní správu vzhledu a chovaní UI. Xaml kód této komponenty (kap. 1.1.4) ilustruje obr. č. 18.

```
<UserControl x:Class="DiceMaster3600.View.Controls.RoundedIcon"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml“
             xmlns:local="clr-namespace:DiceMaster3600.View.Controls
             <!--Ostatní jmenné prostory-->

    <UserControl.Resources>
        <conv:BooleanToVisibilityConverter x:Key="BoolVisibilityConverter" />
        <conv:InvertBooleanToVisibilityConverter
                        x:Key="InvertBoolVisibilityConverter" />
    </UserControl.Resources>

    <Grid>
        <PasswordBox Name="PasswordBox"
                     Style="{StaticResource PlaceholderPasswordBox}"
                     PasswordChanged="OnPasswordChanged"
                     Visibility="{Binding IsPasswordHidden,
                     Converter={StaticResource BoolVisibilityConverter}}">
        </PasswordBox>
        <TextBox Name="VisiblePasswordBox"
                 Style="{StaticResource BaseTextBoxStyle}"
                 Text="{Binding Password, Mode=TwoWay,
                 UpdateSourceTrigger=PropertyChanged}"
                 Visibility="{Binding IsPasswordHidden,
                 Converter={StaticResource InvertBoolVisibilityConverter}}">
        </TextBox>
        <Button Content="👁"
                Foreground="Black"
                Background="Transparent"
                HorizontalAlignment="Right"
                Command="{Binding TogglePasswordVisibilityCommand}"
                Width="30" >
        </Button>
    </Grid>
</UserControl>
```

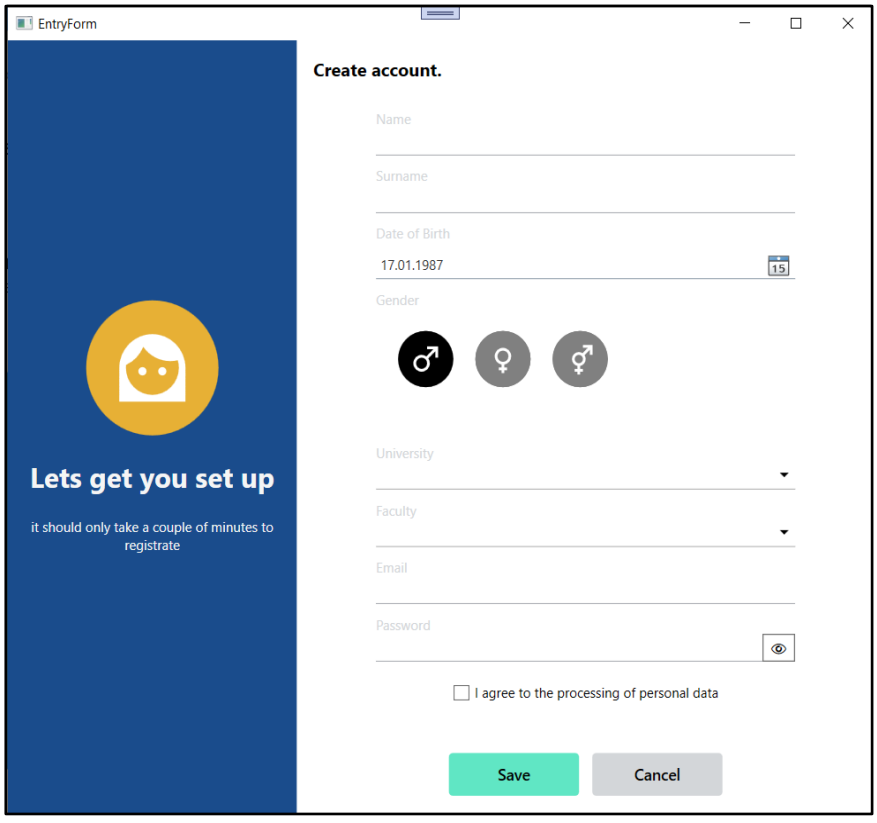
Obrázek 18: PasswordToggleControl

Pro udržení principů MVVM v komponentě PasswordToggleControl, je nutné svázat všechny uživatelské prvky s vlastním viewmodelem. V code-behind této komponenty je deklarována pouze závislá vlastnost Password, což je nativní funkce WPF plně kompatibilní s MVVM. ViewModel, resp. code-behind komponenty je uveden v příloze P. IX. Bez implementace MVVM je často možné funkcionalitu komponent zjednodušit. Příkladem je ovládací prvek DiceControl, který vlastní viewmodel nemá. Logika pro spuštění animace (rotace prvku o 360°) se nachází přímo ve view (obr. 19).

```
public DiceControl()
{
    InitializeComponent();
    HideAllGroups();

    Loaded += (s, e) => StartRotation();
    IsEnabledChanged += (s, e) =>
       { if (IsEnabled) StartRotation(); };
}

private void StartRotation()
{
    Storyboard? storyboard =
              FindResource("RotateAnimation") as Storyboard;
    storyboard?.Begin(this);
}
```

Obrázek 19: DiceControl

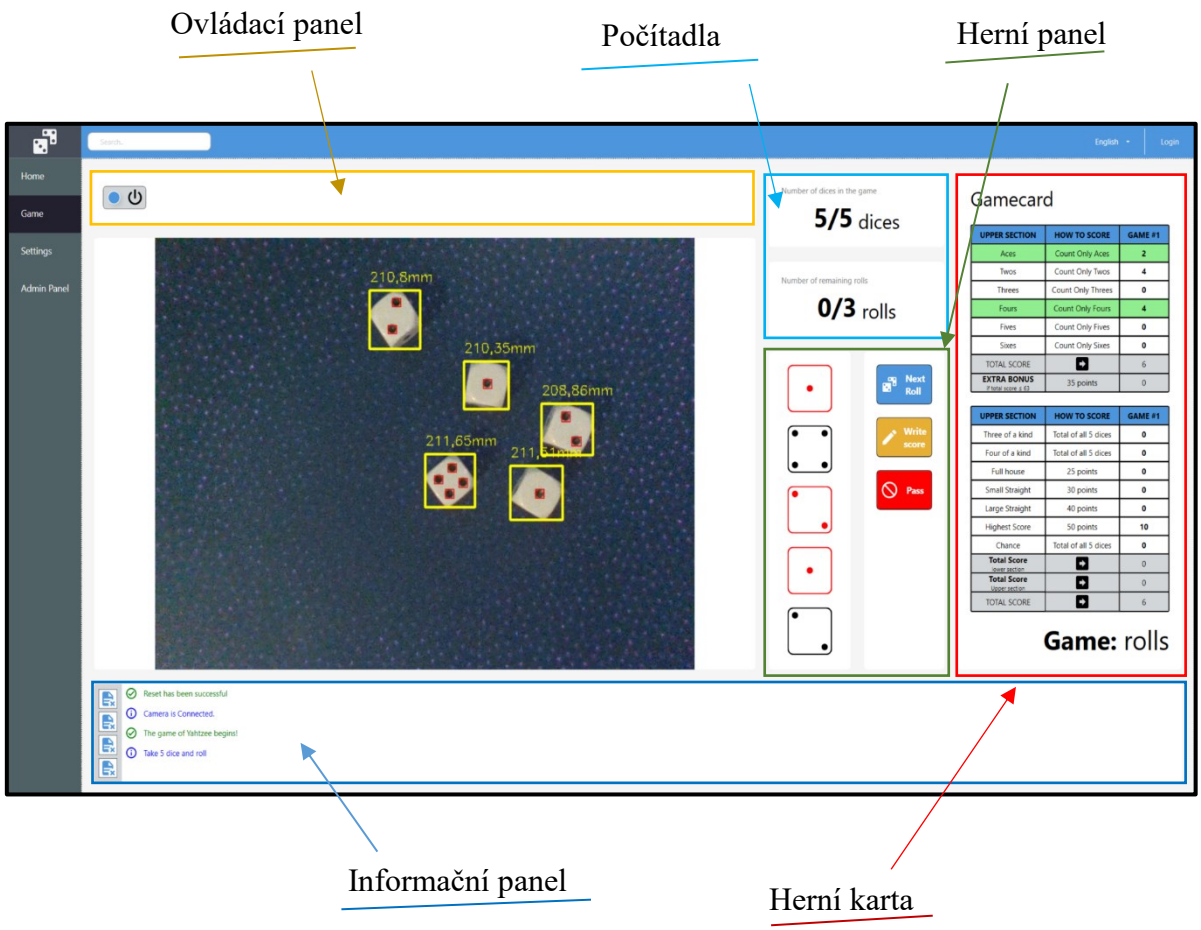
Náhled View registračního formuláře zobrazuje obr. 20. Jak již bylo zmíněno, vedle PasswordBoxu je formulář složen z několika dalších uživatelských komponent. Každá z těchto komponent je navržena tak, aby byla samostatně funkční a přitom plně esteticky a funkčně integrovaná do celkového designu formuláře. Rozdělení odpovědnosti na jednotlivé komponenty podporuje principy OOP. Využití modulárních komponent usnadňuje údržbu a aktualizace aplikace.



Obrázek 20: Registrační formulář

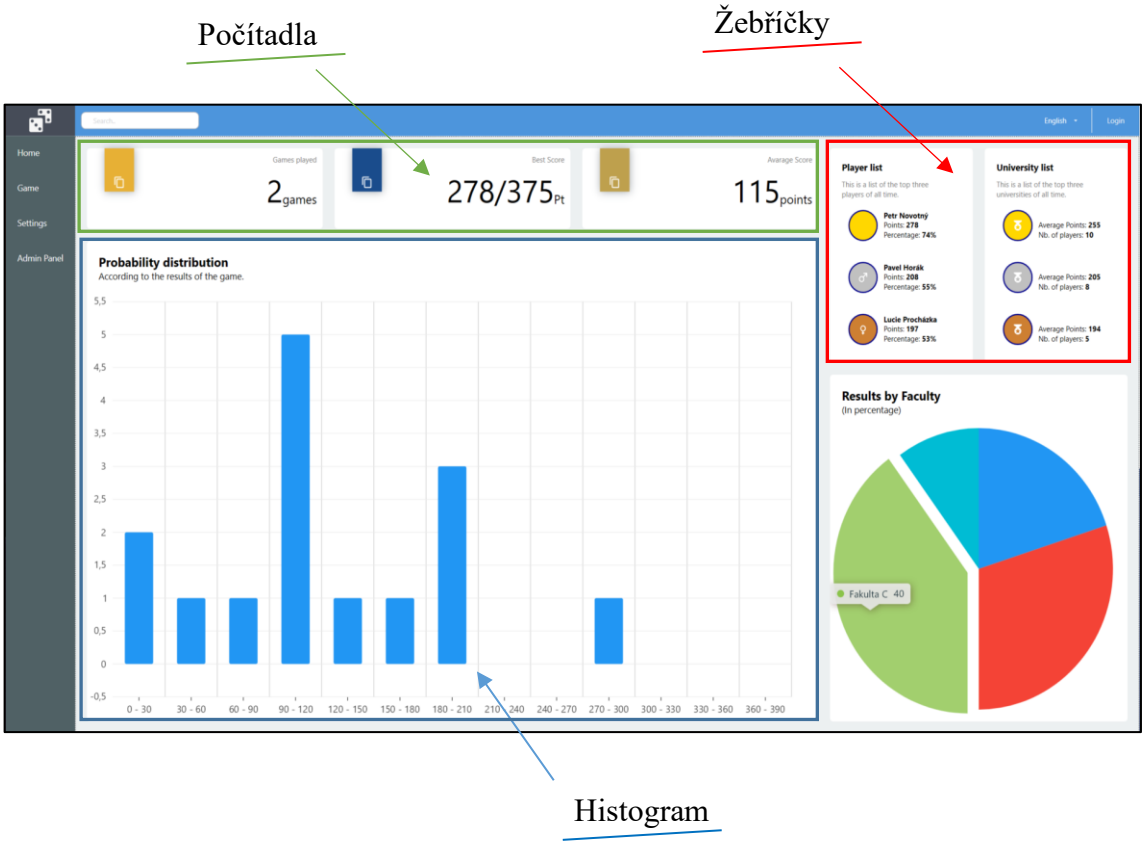
3.5 Dosažené výsledky

Samotná aplikace je poměrně rozsáhlá a obsahuje mnoho funkcionalit. Abychom si její možnosti přiblížili co nejblíže, budeme její klíčové funkce demonstrovat na dvou hlavních obrazovkách. První z těchto obrazovek se věnuje samotné hře (obr. č. 21). V herním panelu, který je centrální součástí tohoto UI, je zobrazena vizuální reprezentace hracího pole. Když uživatel hodí kostkami, systém pomocí kamery zaznamená a na herní panel zobrazí aktuální stav kostek na herní ploše. Kliknutím na kterýkoli z modelů kostek může hráč danou kostku „*zamknout*“. Tato kostka je vizuálně zvýrazněna červeným obrysem a vyřazena z dalšího hodu. Na tuto skutečnost je hráč upozorněn pomocí zprávy na informačním panelu. Systém při dalším hodu očekává přesně daný počet kostek, pokud požadovaný počet kostek a počet skutečně detekovaných kostek neodpovídá, není uživateli herní panel zpřístupněn. Hráč má k dispozici celkem tři pokusy na hod, během kterých se může rozhodnout, které kostky si ponechá, a které případně vrátí do hry.



Obrázek 21: Herní obrazovka

Během kteréhokoli z hodů, resp. nutně po hodu třetím, se může uživatel rozhodnout pro ukončení pokusu a zapsání hodu do skórovací karty. Tato karta obsahuje všechny možné kombinace, které mohou být skórovány podle pravidel hry Yahtzee. Karta je po každém hodu automaticky vyhodnocena a uživateli jsou zpřístupněny ty kombinace, do kterých lze daný hod zapsat. Jakmile hráč svůj pokus do hrací karty zapíše, podbarví se daná kombinace hodnot zeleně, a do konce hry už není k dispozici. Ve hře je celkem 13 kol, přičemž v každém kole se postupuje stejným postupem hodu, výběru kostek a skórování.



Obrázek 22: Statistický přehled

Druhou obrazovkou je statistika odehraných her. Hlavním prvkem je histogram, který zobrazuje rozdělení skóre podle výsledků hry. Histogram je rozdělen do sloupců, které reprezentují rozsahy skóre v bodech. Výška jednotlivých sloupů ukazuje, kolik her skončilo v daném bodovém rozsahu. Distribuce skóre má několik vrcholů, což není z hlediska teorie pravděpodobnosti úplně běžné. Tato neobvyklá distribuce je pravděpodobně způsobena nezkušeností testovacího hráče, který neoptimalizoval strategii pro maximální bodový zisk. V horní části formuláře jsou umístěny počítadla průměrného bodového zisku všech hráčů a počtu odehraných her. Koláčový digram v pravé části obrazovky zobrazuje procentuální rozdělení zisku bodů podle jednotlivých fakult.

3.6 Návrh na další rozvoj aplikace

Jednou z možností pro další rozšíření aplikace je vytvoření nové obrazovky pro manuální kalibraci rozpoznávacího procesu hracích kostek na kalibračním vzorku. Tato obrazovka by umožňovala upravit různé parametry procesu, jako například prahové hodnoty intenzity a hloubky pro detekci hracích kostek nebo minimální rozměry kontur. Díky tomu by bylo možné přesněji detekovat kostky různých velikostí a tvarů. V současné aplikaci jsou tyto hodnoty nastaveny defaultně na základě testovacího vzorku.

Aplikace by rovněž mohla implementovat např. systém oprávnění uživatelů, který by zahrnoval roli správce s přístupem ke speciálním obrazovkám pro správu uživatelských profilů. Na těchto obrazovkách by správce mohl jednoduše spravovat uživatelské profily a např. upravovat přístupová práva uživatelů k různým částem aplikace, včetně možnosti přidělit nebo odebrat roli správce jiným uživatelům. Tento systém by poskytoval lepší kontrolu nad uživatelskými účty, což by zvýšilo bezpečnost a konzistenci aplikace.

Díky modulární architektuře uživatelských prvků je možné snadno a efektivně přidávat nová vizuální témata, díky kterým by uživatelé mohli přizpůsobit vzhled a styl aplikace svým preferencím. Vedle změny vizuálního tématu aplikace umožňuje modulární architektura přidávání nových her. Poměrně snadno lze implementovat nové algoritmy pro počítání bodů či nové/upravené procesy pro zpracování obrazových dat. Stejně tak by bylo možné rozpoznávat hodnoty hracích karet nebo herních žetonů, přičemž základní proces rozpoznávání by zůstal konzistentní a spolehlivý.

Samotná aplikace je napsána v technologii WPF, která je zaměřena na tvoru UI na platformě Windows. Nicméně, pomocí např. frameworku MAUI (Multi-Platform App UI), který umožňuje psát kód v jazyce C# a využívat XAML pro tvorbu uživatelského rozhraní, by mohla aplikace běžet nejen na Windows, ale také na macOS nebo Linuxu. Přechod na MAUI by vyžadoval úpravu některých částí kódu pro kompatibilitu s multiplatformním prostředím [15]. Klíčové by bylo přizpůsobení UI tak, aby se správně zobrazovalo na různých zařízeních a velikostech obrazovky. Díky modulárnímu designu aplikace a oddělení logiky od uživatelského rozhraní by však tento přechod mohl být proveden postupně a efektivně.

ZÁVĚR

Tato bakalářská práce byl zaměřena na návrh a realizaci desktopové aplikace s využitím architektonického vzoru MVVM. Jednotlivé kapitoly práce korespondovaly se stanovenými cíli. První kapitola poskytla ucelený přehled o současném stavu technologií a architektonických vzorů v oblasti vývoje desktopových aplikací. Druhá kapitola se věnovala hloubkovým kamerám a jejich využití v současné technické praxi. Diskutovány byly hlavní výhody hloubkových kamer oproti standardním snímacím technologiím a představeny některé základní fyzikální principy, které tyto kamery pro detekci hloubky využívají.

V praktické částí práce byly specifikovány funkční a nefunkční požadavky na aplikaci a jejich pokrytí pomocí případů užití. Čtvrtá kapitola popisuje realizaci aplikace, včetně představení a popisu funkcí všech klíčových komponent, které byly navrženy s ohledem na modulárnost a rozšiřitelnost systému. Aplikace samotná je intuitivní a snadno ovladatelná. Systémové zprávy poskytují užitečný feedback a informace o průběhu hry, což hráčům pomáhá udržet přehled o hře a strategicky plánovat své tahy. Praktickým přínosem je vytvoření knihovny pro komunikaci s hloubkovou kamerou a vývoj systému pro zpracování obrazu pomocí registrace uživatelských procesů. V závěru práce byly zhodnoceny dosažené výsledky a navrženy další možnost rozvoje aplikace. Dosažené výsledky demonstrují, že implementované řešení splňuje stanovené cíle a představuje příležitost pro další rozvoj.

SEZNAM POUŽITÉ LITERATURY

1. PRICE, Mark J. C# 9 and .NET 5 - Modern Cross-Platform Development. 5. Birmingham: Packt Publishing, 2020. ISBN 9781800568105.
2. MICROSOFT CORPORATION. *Windows Presentation Foundation*. Online. 2024. Dostupné z: https://visualstudio.microsoft.com/cs/vs/features/wpf/. [cit. 2024-05-03].
3. J. DU PREEZ, Ockert. Visual Studio 2022 In-Depth: Explore the Fantastic Features of Visual Studio 2022. 2. Noida (India): BPB Publications, 2022. ISBN 9355512457.
4. MORAVČÍK, Oliver et al. Elements of the Modern Application Software Development. *International Journal of Computer Theory and Engineering* [online]. 2015[cit. 2024-03-23]. Dostupné z: https://www.researchgate.net/publication/272912523_Elements_of_the_Modern_Application_Software_Development
5. GAMMA, Erich. *Design patterns: elements of reusable object-oriented software*. Boston: Addison-Wesley, 1995. ISBN 978-0201633610.
6. JOHNSON, Ralph a Brian FOOTE. Designing Reusable Classes. *Journal of Objected-Oriented Programming*. 1988, **1**(6), 22-35.
7. DOBEŠ, Michal. Zpracování obrazu a algoritmy v C#. Praha: BEN - technická literatura, 2008. ISBN 978-80-7300-233-6.
8. KOVALEVSKY, Vladimir. Modern Algorithms for Image Processing: Computer Imagery by Example Using C#. 1. Berkeley (California): Apress Berkeley, CA, 2019. ISBN 978-1-4842-4237-7
9. TADIC V. et al. Perspectives of RealSense and ZED Depth Sensors for Robotic Vision Applications. Machines. Online. 2022. Dostupné z https://doi.org/10.3390/machines10030183
10. INTEL CORPORATION. *Beginner's guide to depth (Updated)*. Online. 2019. Dostupné z: https://www.intelrealsense.com/beginners-guide-to-depth/. [cit. 2024-03-23].
11. MICROSOFT CORPORATION. *NuGet documentation*. Online. In: Learn .microsoft. 2024. Dostupné z: https://learn.microsoft.com/en-us/nuget/. [cit. 2024-04-11].
12. PAULI, J. a DIANXIANG XU. Integrating functional and security requirements with use case decomposition. Online. In: *11th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'06)*. IEEE Comput. Soc, 2006, 10-. ISBN 0-7695-2530-X. Dostupné z: https://doi.org/10.1109/ICECCS.2006.1690355. [cit. 2024-04-28].
13. COCKBURN, Alistair. *Writing Effective Use Cases*. 2. Pearson Custom Publishing, 2007. ISBN 978-0201702255.

14. JOVANOVIČ, Milan. *Implementing Soft Delete With EF Core*. Online. In: Milanjovanovic.tech. 2024. Dostupné z: https://www.milanjovanovic.tech/blog/implementing-soft-delete-with-ef-core. [cit. 2024-05-03].
15. YE, Roger. . *NET MAUI Cross-Platform Application Development: Leverage a first-class cross-platform UI framework to build native apps on multiple platforms*. Packt Publishing Ltd, 2023.

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

WPF	Windows Presentation Foundation
MVVM	Model-View-ViewModel
CPU	Central processing unit
GPU	Graphics processing unit
IDE	Integrated Development Environment
PaaS	Platform as a Service
MVP	Model-View-Presenter
MVC	Model-View-Controller
UI	User Interface
UWP	Universal Windows Platform
EDA	Even-Driven architecture
CBD	Component-Based Design
GOF	Gang of Four
DDD	Domain-Driven Designt
ORM	Object-Relational Mapping
IoC	Inversion of Control
DI	Dependency Injection
ZPO	Zpracování obrazu
ToF	Time of Flight
LIDAR	Light Detection and Ranging
DTO	Data Transfer Object
MAUI	Multi-platform App UI

SEZNAM OBRÁZKŮ

SEZNAM TABULEK

SEZNAM PŘÍLOH

PŘÍLOHA P I: FUNKČNÍ POŽADAVKY

1. REGISTRACE UŽIVATELE
    - RQ1: Umožnit uživatelům vytvořit účet zadáním jména, e-mailu a hesla
    - RQ2: Ověřit zda je e-mailová adresa unikátní
    - RQ3: Zobrazit notifikaci o úspěšné/neúspěšné registraci
    - RQ4: Po úspěšné notifikaci zobrazit okno pro přihlášení
2. STATISTICKÉ VYHODNOCENÍ
    - RQ5: Počítat a zobrazovat distribuci pravděpodobností výsledků hodu kostkami
    - RQ6: Analyzovat úspěšnost hráčů podle fakulty
    - RQ7: Zobrazit úspěšnost hráčů podle genderu
    - RQ8: Vyhodnocovat a zobrazovat statistiky tří nejlepších hráčů
3. PŘIHLÁŠENÍ UŽIVATELE
    - RQ9: Umožnit uživateli přihlásit se pomocí e-mailu a hesla
    - RQ10: Zobrazit chybou zprávu ve formě vyskakovacího okna v případě nesprávně zadaného e-mailu nebo hesla.
4. HRA YAHTZEE
    - RQ11: Umožnit uživatelům založit a spustit novou hru
    - RQ12: Zaznamenat hod kostkami pomocí 3D kamery
    - RQ13: Automaticky vyhodnotit hod a zobrazit uživateli možnosti zápisu bodů
    - RQ14: Vyhodnotit celkový počet získaných bodů a uložit výsledek do databáze
5. DATABÁZE
    - RQ15: Umožnit uživateli uložení základních informací jako jsou jméno a heslo
    - RQ16: Umožnit uživateli aktualizovat své osobní údaje
    - RQ17: Automaticky ukládat detaily každé hry včetně datumu a konečného skóre.

PŘÍLOHA P II: NEFUNKČNÍ POŽADAVKY

1. BEZPEČNOST
    - RQ16: Zašifrovat hesla uživatelů při ukládání do databáze.
    - RQ17: Zajistit ochranu osobních údajů v souladu s GDPR.
    - RQ18: Implementovat bezpečné přihlašování a komunikaci mezi klientem a serverem.
2. VÝKON
    - RQ19: Optimalizace zpracování dat z 3D kamery pro minimální latenci
    - RQ20: Zajištění rychlého načítání hrací plochy a hladké interakce
3. UŽIVATELSKÁ PŘÍVĚTIVOST
    - RQ21: Intuitivní navigace v aplikaci
    - RQ22: Responzivní design vhodný pro různé typy obrazovek
    - RQ23: Jednoduché a srozumitelné UI
4. TESTOVATELNOST A MODULARITA
    - RQ24: Aplikace je snadno testovatelná s využitím aut. testů.
    - RQ25: Aplikace je modulární a snadno rozšiřitelná
5. KOMPATIBILITA
    - RQ26: Aplikace je kompatibilní s hlavními verzemi Windows
    - RQ27: Aplikace podporuje různé konfigurace HW, včetně různých modelů kamer.

PŘÍLOHA P III: USE CASES

Tabulka PII 1.1: Alternativní scénář UC01

Název: Alternativní scénář: Registrovaný e-mail není unikátní		
ID: UC01a		
Charakteristika: V bodě 4 UC01 selhalo ověření unikátnosti e-mailové adresy		
Alternativní scénář:		
Krok	Aktér/Systém	Popis
1	Systém	Systém zobrazí notifikaci, že daný e-mail byl v minulosti již registrován
2	Uživatel	Uživatel potvrdí přijetí zprávy
3	Systém	Systém vyzve uživatele k zadání nového e-mailu
4	Uživatel	Uživatel zadá nový e-mail
5	Sytém	Případ užití pokračuje bodem 4 UC01

Tabulka PII 1.2: Alternativní scénář UC01

Název: Alternativní scénář: Chyba při vyplňování registračního formuláře		
ID: UC01b		
Charakteristika: V bodě 5 UC01 selhalo ukládání do databáze		
Alternativní scénář:		
Krok	Aktér/Systém	Popis
1	Systém	Systém identifikuje jeden nebo více údajů v registračním formuláři jako neplatné
2	Systém	Systém zobrazí chybovou notifikaci
3	Uživatel	Uživatel potvrdí přijetí zprávy
4	Sytém	Případ užití pokračuje bodem 3 UC01

Tabulka PII.2: Případ užití UC02

Název: Přihlášení uživatele
ID: UC02
Charakteristika: Umožnit uživateli přístup do systému
Primární aktér: Registrovaný uživatel
Vedlejší aktér: Žádný
Vstupní podmínky: Uživatel je registrován

Výstupní podmínky: Uživatel je přihlášen do systému		
Hlavní scénář:		
Krok	Aktér/Systém	Popis
1	Uživatel	Uživatel zadá jméno a heslo
2	Sytém	Systém ověří údaje proti databázi (jméno a hash hesla)
3	Sytém	Systém umožní uživateli přístup ke hře.
4	Sytém	Případ užití končí.
Alternativní scénáře: UC02a – Zadané údaje neodpovídají záznamům v databázi		

Tabulka PII.2.1: Alternativní scénář UC02

Název: Alternativní scénář: Zadané údaje neodpovídají záznamům v databázi		
ID: UC02a		
Charakteristika: V bodě 2 UC02 selhalo ověření údajů proti databázi		
Alternativní scénář:		
Krok	Aktér/Systém	Popis
1	Systém	Systém identifikuje, že jeden nebo více údajů v registračním formuláři jako neplatné.
2	Systém	Sytém zobrazí chybovou notifikaci.
3	Uživatel	Uživatel potvrdí přijetí zprávy
4	Systém	Případ užití končí

Tabulka PII.3: Případ užití UC03

Název: Hraní hry		
ID: UC03		
Charakteristika: Uživatel hází hracími kostkami a zapisuje skóre		
Primární aktér: Přihlášený uživatel		
Vedlejší aktér: Žádný		
Vstupní podmínky: Uživatel je přihlášen		
Výstupní podmínky: Do systému je uloženo celkový dosažený počet bodů.		
Hlavní scénář:		
Krok	Aktér/Systém	Popis

1	Uživatel	Uživatel stisknutím tlačítka připojí kameru
2	Systém	Uživatel je vyzván k hodu pěti hracími kostkami
3	Uživatel	Uživatel hodí hracími kostkami
4	Systém	Systém vyhodnotí hod a zobrazí možnosti pro zápis bodů.
5	Uživatel	Uživatel zapíše body do vybrané kolonky.
6	Systém	Systém spočítá, zda byl dosažen limit počtu pokusů
7a	Systém	Pokud limit počtu pokusů nebyl dosažen, proces se vrací ke kroku 2.
7b	Systém	Přesáhne-li počet pokusů stanoveny limit, sytém zobrazí notifikaci o ukončení hry a pokračuje krokem 8.
8	Systém	Systém odešle žádost u uložení výsledku do databáze
9	Systém	Databázový server uloží výsledný počet bodů
10	Sytém	Případ užití končí
Alternativní scénáře: UC03a – Technická chyba při hodu		

Tabulka PII.3.1: Alternativní scénář UC03

Název: Alternativní scénář: Technická chyba při hodu		
ID: UC03a		
Charakteristika: V bodě 4 UC03 selhalo vyhodnocení hodu		
Alternativní scénář:		
Krok	Aktér/Systém	Popis
1	Systém	Při zaznamenání hodu systém detekuje technickou chybu
2	Systém	Systém upozorní uživatele na možnost tech. Problému.
3	Uživatel	Uživatel zvolí, zda chce hod opakovat
4a	Uživatel	Uživatel zvolil pokračování ve hře – případ užití pokračuje bodem 3 UC03
4b	Uživatel	Uživatel zvolil nepokračovat ve hře – systém zašle zprávu o ukončení hry
5	Systém	Případ užití končí.

Tabulka PII.4: Případ užití UC04

Název: Statistické vyhodnocení
ID: UC04

Charakteristika: Umožnit uživatelům přistup k statistickým analýzám her		
Primární aktér: Uživatel		
Vedlejší aktér: Databázový server		
Vstupní podmínky: Hry jsou dostupné v databází		
Výstupní podmínky: Systém spočítá a zobrazí statistiky		
Hlavní scénář:		
Krok	Aktér/Systém	Popis
1	Uživatel	Uživatel zvolí možnost zobrazit statistiky
2	Sytém	Systém odešle žádost na databázový server
3	Dat. server	Databázový server poskytne data
4	Sytém	Sytém zobrazí statistiky – distribuci pravděpodobnosti…
5	Systém	Případ užití končí.
Alternativní scénáře: UC04a – Databázový server nezjistí dostatečné množství dat		

Tabulka PII.3.1: Alternativní scénář UC04

Název: Alternativní scénář: Databázový server nezjistí dostatečný počet dat		
ID: UC04a		
Charakteristika: V bodě 3 UC04 databázový server nezjistí dostatečná počet dat		
Alternativní scénář:		
Krok	Aktér/Systém	Popis
1	Systém	Sytém informuje uživatele, že data nejsou dostupná.
2	Systém	Systém nahraje do UI defaultní hodnoty.
3	Systém	Případ užití končí.

PŘÍLOHA P IV: IMPLEMENTACE PROCESU

```
public class DiceRecognitionProcess : BaseResultFrameProcess<int[]>
{
    private const int KernelSize = 5;
    private IProccesSettings settings;
    //Ostatní privatní fiedy...

    public DiceRecognitionProcess(IProccesSettings settings)
    {
        this.settings = settings;
    }

    protected override BitmapSource ProcessFrame(FrameSet frameSet)
    {
        var colorImage = ConvertToColorImage(frameSet.ColorFrame);
        var depthImage = ConvertDepthFrameToImage(frameSet.DepthFrame);
        var diceRectangles = DetectDice(colorImage);
        var dotCounts = DetectDots(colorImage, diceRectangles);

        AnnotateDepth(colorImage, depthImage, diceRectangles);
        Result = dotCounts.ToArray();
        return BitMapConverter.ToBitmapSource(colorImage);
    }

    private List<Rectangle> DetectDice(Image<Bgr, byte> image)
    {
        var grayImage = image.Convert<Gray, byte>().SmoothGaussian(kernelSize);
        var binaryImage = grayImage.ThresholdBinary(new Gray(intensityThreshold),
                                                    new Gray(255));
        var cannyEdges = binaryImage.Canny(100, 200);
        var contours = new VectorOfVectorOfPoint();

        CvInvoke.FindContours(cannyEdges, contours, null, RetrType.List,
                              ChainApproxMethod.ChainApproxSimple);

        List<Rectangle> diceRectangles = new();
        for (int i = 0; i < contours.Size; i++)
        {
            Rectangle rectangle = CvInvoke.BoundingRectangle(contours[i]);
            if (HasValidSize(rectangle, minArea) && !diceRectangles.Any(d =>
                d.IntersectsWith(rectangle)))
            {
                diceRectangles.Add(rectangle);
                CvInvoke.Rectangle(image, rectangle,
                                   new Bgr(Color.Yellow).MCvScalar, 2);
            }
        }

        return diceRectangles;
    }

    private List<int> DetectDots(Image<Bgr, byte> image,
                                 List<Rectangle> diceRectangles)
    {
        List<int> dotsCount = new();
        foreach (var dice in diceRectangles)
        {
            var dieImage = image.GetSubRect(dice);
```

```
            var grayDieImage = dieImage.Convert<Gray, byte>().ThresholdBinaryInv(
                                new Gray(intensityThreshold), new Gray(255));

            var contours = new VectorOfVectorOfPoint();
            CvInvoke.FindContours(grayDieImage, contours, null, RetrType.List,
                                  ChainApproxMethod.ChainApproxSimple);

            int dotCounter = 0;
            foreach (var contour in contours)
            {
                if (HasValidShapeAndSize(contour, dice)) {
                    dotCounter++;
                }
            }
            dotsCount.Add(dotCounter);
        }
        return dotsCount;
    }

    private bool HasValidShapeAndSize(VectorOfPoint contour, Rectangle dice,
                                      out Rectangle validDot)
    {
        validDot = CvInvoke.BoundingRectangle(contour);
        bool isValid = HasValidSize(validDot, minArea / 5)
                               && HasCircularShape(contour);
        if (isValid) {
            validDot.Offset(dice.Location);
        }
        return isValid;
    }

    private bool HasCircularShape(VectorOfPoint contur, double circularityThreshold)
    {
        double area = CvInvoke.ContourArea(contour);
        if (area == 0) return false;

        double perimeter = CvInvoke.ArcLength(contour, true);
        if (perimeter == 0) return false;

        double circularity = (4 * Math.PI * area) / (perimeter * perimeter);
        return circularity > circularityThreshold;
    }

    private bool HasValidSize(Rectangle rect, double size)
    {
        double aspectRatio = (double)rect.Width / rect.Height;
        return aspectRatio >= minAspectRatio && aspectRatio <= maxAspectRatio &&
                rect.Width > size && rect.Height > size &&
                rect.Width < 2 * size && rect.Height < 2 * size;
    }

    private void AnnotateDepth(Image<Bgr, byte> image, Image<Gray, ushort>
                               depthImage, List<Rectangle> diceRectangles)
    {
        foreach (var dice in diceRectangles)
        {
            double averageDepth = CalculateAverageDepth(dice, depthImage);
            string depthText = $"{Math.Round(averageDepth, 2)}mm";
            CvInvoke.PutText(image, depthText, new Point(dice.X, dice.Y - 10),
                 FontFace.HersheySimplex, 0.5, new Bgr(Color.Yellow).MCvScalar);
        }
    }
```

```
private double CalculateAverageDepth(Rectangle diceBox,
                                     Image<Gray, ushort> depthImage)
{
    double sumDepth = 0;
    int pixelCount = 0;
    for (int y = diceBox.Top; y < diceBox.Bottom; y++)
    {
        for (int x = diceBox.Left; x < diceBox.Right; x++)
        {
            ushort depth = depthImage.Data[y, x, 0];
            if (depth > 0)
            {
                sumDepth += depth;
                pixelCount++;
            }
        }
    }
    return pixelCount > 0 ? sumDepth / pixelCount : 0;
}

private Image<Bgr, byte> ConvertToColorImage(VideoFrame colorFrame) =>
        BitMapConverter.ConvertToImage<Bgr, byte>(colorFrame);

private Image<Gray, ushort> ConvertDepthFrameToImage(VideoFrame depthFrame) =>
         BitMapConverter.ConvertToImage<Gray, ushort>(depthFrame);
```

PŘÍLOHA P V: USERREPOSITORY CLASS

```
public class UserRepository : Repository<UserEntity>
{
    public UserRepository(SqlEFDataContext context) : base(context) { }

    public async Task DeleteAllByFacultyIdAsync(int facultyID)
    {
        var entities = await context.Users
               .Where(x => x.FacultyId == facultyID && x.DeletedDate == null)
               .ToListAsync();

         if (entities.Any())
             await DeleteAsync(entities);
    }

    public async Task<UserDTO[]> GetTopThreePlayersAsync()
    {
        var topThreeUsers = await context.Users
             .Where(user => user.DeletedDate == null)
             .OrderByDescending(user => user.NumberOfPoints)
             .Take(3).ToListAsync();

            return topThreeUsers.Select(user => new UserDTO {
                Name = user.Name,
                Surname = user.Surname,
                EmailAddress = user.EmailAddress,
                Gender = user.Gender,
                NumberOfPoints = user.NumberOfPoints
            }).ToArray();
     }

       public async Task<UserEntity?> GetUserByEmailAsync(string email,
                                                          string plainPassword)
     {
         var userEntity = await context.Users.FirstOrDefaultAsync(u
              => u.EmailAddress == email && u.DeletedDate == null);

         return userEntity != null
            && BCrypt.Net.BCrypt.Verify(plainPassword, userEntity.PasswordHash)
            ? userEntit : null;
     }

     public async Task AddAsync(UserDTO userDTO,
                                string plainPassword,
                                int facultyId)
     {
          var userEntity = new UserEntity {
                Name = userDTO.Name,
                Surname = userDTO.Surname,
                Gender = userDTO.Gender,
                FacultyId = facultyId,
                PasswordHash = BCrypt.Net.BCrypt.HashPassword(plainPassword),
                EmailAddress = userDTO.EmailAddress,
                NumberOfPoints = userDTO.NumberOfPoints,
            };

            await context.Users.AddAsync(userEntity);
            await context.SaveChangesAsync();
        }
    }
```

PŘÍLOHA P VI: NOTIFICATIONSERVICE

```
public class NotificationService : IMessageService
{
    private readonly SnackbarMessageQueue messageQueue;
    private readonly ConcurrentDictionary<NotificationContext, List<Action<string>>>
                     notifications = new();

    public SnackbarMessageQueue MessageQueue => messageQueue;


    public NotificationService(SnackbarMessageQueue messageQueue)
    {
        this.messageQueue = messageQueue;
    }

    public void Subscribe(NotificationContext notificationType,
                          Action<string> callback)
    {
        notifications.AddOrUpdate(notificationType,
                       new List<Action<string>> { callback }, (key, oldValue) =>
        {
            lock (oldValue)
            {
                if (!oldValue.Contains(callback))
                {
                    oldValue.Add(callback);
                }
            }
            return oldValue;
        });
    }

    public void Unsubscribe(NotificationContext notificationType)
    {
        if (notifications.TryRemove(notificationType, out var callbacks))
        {
            lock (callbacks)
            {
                callbacks.Clear();
            }
        }
    }

    public void TryNotify(NotificationContext notificationType, string message)
    {
        if (notifications.TryGetValue(notificationType, out var callbacks))
        {
            lock (callbacks)
            {
                callbacks.ForEach(cb => cb(message));
            }
        }
    }

    public void NotifyImmediately(string message)
    {
        messageQueue.Enqueue(message, null, null, false, true,
                             TimeSpan.FromSeconds(1));
    }
    #endregion
}
```

PŘÍLOHA P VII: ENTRYFORMVIEWMODEL

```
public class EntryFormViewModel : ReactiveViewModel
{
    #region Fields
    private string? name;
    private Gender selectedGender = Gender.None;
    private readonly IDataAccessManager dataManager;
    // Ostatní privátní fieldy
    #endregion

    #region Property
    public string? Password
    {
        get => password;
        set => SetTrigger(ref password, value);
    }

    public DateTime DateOfBirth
    {
        get => dateOfBirth;
        set => SetProperty(ref dateOfBirth, value);
    }
    public ICommand SaveCommand { get; }
    public ICommand GenderSelectCommand { get; }
    public ICommand CancelCommand { get; }
    #endregion

    #region Constructors
    public EntryFormViewModel(IDataAccessManager dataManager, IMessageService
                              messageService) : base(messageService)
    {
        this.dataManager = dataManager;
        SubsribeNotification(NotificationContext.RegistrationFailure, m
              => Notify(m, MessageType.Failed));

        SaveCommand = new RelayCommand(async
            () => await ExecuteExampleCommand(),
            () => CanSaveExecute());

        dataManager.OnProcessingStatusChanged += (isInProgress)
            => IsSavingInProgress = isInProgress;

        CancelCommand = new RelayCommand(() => RequestClose?.Invoke());
        GenderSelectCommand = new RelayCommand<Gender>((g) => SelectedGender = g);
    }
    #endregion

    #region Methods
    private bool CanSaveExecute()
    {
        return !string.IsNullOrWhiteSpace(Name)
            && !string.IsNullOrWhiteSpace(Surname)
            && !string.IsNullOrWhiteSpace(Email)
            && Email.Contains('@')
            && !string.IsNullOrWhiteSpace(Password)
            && University != UniversityType.None
            && Faculty != FacultyType.None
            && (DateTime.Now.Year - DateOfBirth.Year) >= 15;
    }
```

```
    private async Task ExecuteExampleCommand()
    {
        try
        {
            var user = new UserDTO(Name!, Surname!, Email!, SelectedGender);
            await dataManager.RegisterUserAsync(user, Password,
                                                University, Faculty);

            Notify(NotificationContext.RegistrationSuccess,
                   "Registration was byla succesfull!");

            RequestClose?.Invoke();
        }
        catch (Exception ex)
        {
            Notify(NotificationContext.RegistrationFailure,
                   $"Nastala chyba: ex.Message}");
            AppLogger.Error(ex.Message);
        }
    }

    public override void Dispose()
        => UnsubsribeNotification(NotificationContext.RegistrationFailure);

    public override void RefreshCommand()
    {
        (SaveCommand as RelayCommand)?.NotifyCanExecuteChanged();
    }
    #endregion

    #region Events
    public event Action? RequestClose;
    #endregion
}
```

PŘÍLOHA P VIII: USERCONTROL VIEWMODEL A CODE-BEHIND

```
class PasswordToggleControlViewModel : ObservableObject
{
    private bool isPasswordHidden = true;

    public bool IsPasswordHidden
    {
        get => isPasswordHidden;
        private set => SetProperty(ref isPasswordHidden, value);
    }

    public ICommand TogglePasswordVisibilityCommand { get; }

    public PasswordToggleControlViewModel()
    {
        TogglePasswordVisibilityCommand = new RelayCommand(
            () => IsPasswordHidden = !IsPasswordHidden);
    }
}


public partial class PasswordToggleControl : UserControl
{
    public static readonly DependencyProperty PasswordProperty =
        DependencyProperty.Register(
            "Password",
            typeof(string),
            typeof(PasswordToggleControl),
            new PropertyMetadata(string.Empty, OnPasswordChanged)
        );

    public string Password
    {
        get => (string)GetValue(PasswordProperty);
        set => SetValue(PasswordProperty, value);
    }

    public PasswordToggleControl()
    {
        DataContext =
App.AppHost!.Services.GetService<PasswordToggleControlViewModel>();
        InitializeComponent();
    }

    private static void OnPasswordChanged(DependencyObject d,
DependencyPropertyChangedEventArgs e)
    {
        var control = (PasswordToggleControl)d;
        string newPassword = (string)e.NewValue;

        control.PasswordBox.Password = newPassword;
        control.VisiblePasswordBox.Text = newPassword;
    }

    private void OnPasswordChanged(object sender, RoutedEventArgs e)
    {
        PasswordBox passwordBox = (PasswordBox)sender;
        Password = passwordBox.Password;
    }
}
```