|  |  |
| --- | --- |
|  | |
| **Návrh a implementace WPF aplikace pro 3D detekci a analýzu hracích kostek** | |
|  | |
| Ing. Jaroslav Janků | |
|  | |
|  |  |
| Bakalářská práce  2024 |  |
|  |  |
|  | |

\*\*\* nascannované zadání s. 1 \*\*\*

\*\*\* nascannované zadání s. 2 \*\*\*

\*\*\* naskenované Prohlášení str. 1 \*\*\*

\*\*\* naskenované Prohlášení str. 2 \*\*\*

Zásady pro vypracovaní:

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 WPF aplikace, která využívá architektonický vzor Model-View-ViewModel (MVVM) pro efektivní rozdělení logiky a uživatelského rozhraní v kontextu 3D zpracování a analýzy obrazu hracích kostek. Pro detekci a vyhodnocení 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. Navržený systém slouží jako marketingový nástroj pro prezentaci vývojového oddělení společnosti Meopta-Optika, s.r.o. na veletrzích a jiných obchodních akcích.

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

Zde je místo pro případné poděkování, popř. motto, úryvky knih atp.

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

OBSAH

[Úvod 9](#_Toc22719940)

1. [TEORETICKÁ ČÁST 10](#_Toc22719941)

[1 vývoj desktopových aplikací 11](#_Toc22719942)

[1.1 Architektonické vzory 11](#_Toc22719943)

[1.1.1 Architektura Model-View-Presenter 11](#_Toc22719945)

[1.1.2 Architektura Model-View-ViewModel 11](#_Toc22719945)

[1.1.3 Event-driven Architektura 11](#_Toc22719945)

[1.1.4 Component-Based Design 11](#_Toc22719945)

[3.1 Návrhové vzory 14](#_Toc22719950)

[3.1.1 Podpodnadpis 14](#_Toc22719951)

[3.1.1 Podpodnadpis 14](#_Toc22719951)

[2 Hloubkové kamery 12](#_Toc22719946)

[2.1 Podnadpis 12](#_Toc22719947)

[2.1 Podnadpis 14](#_Toc22719950)

[2.1.1 Podpodnadpis 14](#_Toc22719951)

1. [PRAKTICKÁ ČÁST 13](#_Toc22719948)

[3 nadpis hlavní kapitoly 14](#_Toc22719949)

[3.1 Podnadpis 14](#_Toc22719950)

[3.1.1 Podpodnadpis 14](#_Toc22719951)

[3.2 Podnadpis 14](#_Toc22719952)

[4 NADPIS HLAVNÍ KAPITOLY 15](#_Toc22719953)

[4.1 Podnadpis 15](#_Toc22719954)

[závěr 16](#_Toc22719955)

[SEZNAM POUŽITÉ LITERATURY 17](#_Toc22719956)

[seznam použitých symbolů a zkratek 18](#_Toc22719957)

[seznam OBRÁZKŮ 19](#_Toc22719958)

[seznam TABULEK 20](#_Toc22719959)

[seznam PŘÍLOH 21](#_Toc22719960)

Úvod

text

|  |  |
| --- | --- |
|  | TEORETICKÁ ČÁST |

Vývoj desktopových aplikací

Vývoj desktopových aplikací se zaměřuje na tvorbu softwaru určeného pro běh na stolních a přenosných počítačích. Desktopové aplikace jsou zpravidla navrženy tak, aby plně využívaly hardwarové a systémové zdroje daného počítače, díky čemuž mohou nabídnout vysokou účinnost a pokročilé uživatelské funkce. Desktopové aplikace mohou rovněž plně a efektivně fungovat „*offline*“, což je užitečné zejména v prostředích s omezeným nebo žádným přístupem k internetu. Vzhledem k tomu, že mají desktopové aplikace přímý přístup k systémovým zdrojům[[1]](#footnote-1) je klíčovou otázkou bezpečnost těchto aplikací. Vývojáři by při vývoji desktopových aplikací měli implementovat bezpečnostní opatření, jako např. šifrování dat či zajištění bezpečné komunikace. Co se techniky vývoje desktopových aplikací týče můžeme tyto podle [1] rozdělit na čtyři základní 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 programovacím jazykem a verzovacím systémem je zřejmý. Co už tak zřejmé není, je rozdíl mezi vývojovým prostředím a frameworkem. Rozdíl mezi těmito pojmy je v jejich účelu a způsobu použití ve vývoji softwaru. Framework je soubor knihoven a nástrojů, které poskytují předdefinovanou strukturu souboru funkcí a pravidel pro vývoj softwaru. Framework tak můžeme chápat jako šablonu nebo „*rámec*“ pro vývoj aplikací. A naopak, vývojové prostředí slouží k centralizaci všech potřebných vývojových nástrojů[[2]](#footnote-2) do jednoho uživatelského rozhraní. Vývojové prostředí (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. 1.1.

Tabulka 1.1 Rozdíl mezi Frameworkem a vývojovým prostředím[[3]](#footnote-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 určité architektonické a návrhové vzory, které jsou optimalizované pro konkrétní architektonické a návrhové principy. Využití těchto vzorů v rámci frameworku umožňuje vývojáři lépe strukturovat kód své aplikace a zároveň se vyhnout běžným chybám a problémům. Jako konkrétní příklad můžeme uvést .NET framework, který používá vlastní multivrstvou architekturu a pro své funkce implementuje celou řadu návrhových vzorů. Za všechny jmenujme např. návrhový vzor Singleton, který používá třída *System.Windows.Application* k zajištění toho, že v aplikaci existuje pouze jedno hlavní okno. V následujících podkapitolách si architektonické a návrhové vzory trochu více přiblížíme. Zavedeme stručné definice těchto pojmů a uvedeme si několik konkrétních příkladů.

Architektonické vzory

Podle [1] definuje softwarová architektura 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. Pro lepší porozumění této poněkud krkolomné definice si můžeme software samotný představit jako rodinný dům. Architektura softwaru pak tvoří jeho základní konstrukci – rozvržení pokojů, nosné zdi, rozvody elektřiny a další klíčové prvky. Stejně jako v případě našeho 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. Kvalitní softwarová architektura je základem pro budoucí řešení výkonnosti, odolnosti, škálovatelnosti a spolehlivosti softwaru. Správným výběrem softwarové architektury zajišťujeme, že software bude i po rozšíření o novou funkcionalitu udržovat vysokou úroveň výkonu a stability. Investice do kvalitní architektury se v dlouhodobém horizontu zúročí v podobě spolehlivého a flexibilního systému, který bude schopen obstát v náročných podmínkách, a který má nejlepší předpoklad uspokojit rostoucí požadavky klientů. Při výběru vhodné softwarové architektury je důležité pozorně sledovat jejich neustálý vývoj a zdokonalování. Třebaže jsme v posledních letech svědky nárůstu popularity webových a mobilních aplikací[[4]](#footnote-4), s časem se výrazně mění i svět desktopových aplikací, s čímž úzce souvisí i změna trendů v oblasti vývoje softwarových architektur. Tradiční monolitické struktury ustupují modernějším přístupům, které kladou větší důraz na modularitu, flexibilitu a multiplatformní vývoj.

Softwarových architektur existuje velké množství. Můžeme je dělit podle např. zaměření, struktury, fáze vývoje či způsobu komunikace. Vzhledem k omezenému rozsahu a praktickému zaměření této práce se v následujícím výčtu moderních desktopových softwarových architektur omezíme pouze na ty specifické architektonické vzory, které jsou určeny pro vývoj uživatelských rozhraní, a to především pomocí technologie WPF. Z tohoto pohledu patři mezi v současnosti nejpoužívanější architektury Model-View-ViewModel, Model-View-Presenter, Event-Driven Architecture, a Domain-Driven Design.

Architektura Model-View-Presenter

Architektonický vzor Model-View-Presenter (MVP) vychází ze staršího vzoru Model-View-Controller (MVC[[5]](#footnote-5)), 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:

* **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.

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 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[[6]](#footnote-6).

V textu výše bylo zmíněno, že MVVM architektura je jakousi variací na MVP. A skutečně tomu tak je, zásadním rozdílem mezi oběma architekturami 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:

VIEW

MODEL

Controller



VIEW

MODEL

Controller



VIEW

MODEL

ViewModel



**MVC**

**MVP**

**MVVM**

Obrázek 2.1: Softwarové architektury. Vlastní zpracování.

Even driven architektura (EDA)

Podle [3] je EDA je komplexí architektonický styl, ve kterém se ústředním bodem pro komunikaci a spolupráci mezi komponentami stávají události. Událost je typicky asynchronní[[7]](#footnote-7) 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.

Component-Based Design

Posledním architektonickým vzorem, který si v této kapitole představíme je tzv. Component-Based Desing (CBD). CBD definuje [3] jako přístup k vývoji softwaru, který zdůrazňuje opakované použití softwarových komponent. A dodává, že tento přístup umožňuje vývojářům sestavit aplikace z předem definovaných komponent, což může výrazně zkrátit dobu vývoje 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.

Co se CBD a frameworku WPF tyče, tento samotný COP přímo neimplementuje, ale silně podporuje jeho koncepty a poskytuje nástroje pro jeho využití. Tím je v daném kontextu myšlena především podpora uživatelských ovládacích prvků 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 TADYTU a TAKYTUTO. Šablony se pak 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. 2.2

Component 1

Component 2

Component 3

Extension Point

Extension Point

Extension Point

Obrázek 2.1: Komponenta. Vlastní zpracování.

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. Podle [2] představují návrhové vzory 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[[8]](#footnote-8). Tato skupina představuje 23 návrhových vzorů rozdělených do tří kategorií, které se staly základem OOP. Ačkoli je většina GoF vzorů je pro současné systémové inženýrství stále široce využívána[[9]](#footnote-9), 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 1.2.

Tabulka 1.2 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 |

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

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[[10]](#footnote-10) 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. 2.2.

Business Layer

Data Access Layer

Domain model

Repository

Data mapper

Query Object

Standard

Obrázek 2.2: Komponenta. Vlastní zpracování.

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 [7] 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 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[[11]](#footnote-11) 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 2. 1.

Main Class

Class 1

Class 2

Class 3

Class 4

Main Class

Dependency injector

Class 1

Class 2

Class 3

Class 4

**Traditional**

**With dependency Injection**

Obrázek 2.2: Komponenta. Vlastní zpracování.

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.

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 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.

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*).

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. 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.

nadpis hlavní kapitola

text

Podnadpis

text

|  |  |
| --- | --- |
|  | PRAKTICKÁ ČÁST |

nadpis hlavní kapitoly

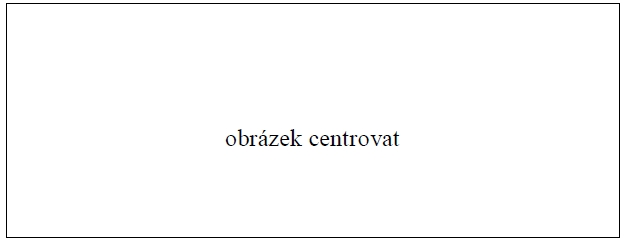
text

Podnadpis

text

Podpodnadpis

text



Obrázek 1 Popisek obrázku

text

Podnadpis

text

Tabulka 1 Popisek tabulky

|  |  |  |  |
| --- | --- | --- | --- |
| **Záhlaví tabulky 1** | **Záhlaví tabulky 2** | **Záhlaví tabulky 3** | **Záhlaví tabulky 4** |
| První řádek | 0,98 | 123,97 | 1258,58 |
| Druhý řádek | 1,5875 | 11,0334 | 251,005 |

NADPIS HLAVNÍ KAPITOLY

text

Podnadpis

text

závěr

Vývoj desktopových aplikací stále zůstává důležitým odvětvím softwarového inženýrství, nabízející vývojářům vysokou míru flexibility a uživatelům výkonné a spolehlivé aplikace. I když je kombinací těchto dvou architektur možné vytvořit robustní, modulární a snadno udržovatelné aplikace. Například, v jednoduchých aplikacích nebo když EF DbContext již poskytuje dostatečnou úroveň abstrakce, může být přímé použití EF bez explicitní vrstvy repositáře vhodnější.

Pak, v roce 1996, Michael Mattson používá termín IoC ve své práci "Object Oriented Frameworks: a survey on methodological issues" k rozlišení skutečného frameworku od knihovny tříd tím, že uvádí:

Hlavní rozdíl mezi objektově orientovaným frameworkem a knihovnou tříd je, že framework volá aplikační kód. Obvykle aplikační kód volá knihovnu tříd. Tato inverze řízení je někdy nazývána hollywoodským principem, "Nevoláme vás, my vás voláme".

SEZNAM POUŽITÉ 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.
6. GAMMA, Erich. *Design patterns: elements of reusable object-oriented software*. Boston: Addison-Wesley, 1995. ISBN 978-0201633610.
7. JOHNSON, Ralph a Brian FOOTE. Designing Reusable Classes. *Journal of Objected-Oriented Programming*. 1988, **1**(6), 22-35.

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

seznam OBRÁZKŮ

[Obrázek 1.1 Softwarové architektury 14](#_Toc22819285)

[Obrázek 1.2 Komponenta 14](#_Toc22819285)

seznam TABULEK

[Tabulka 1.1 Rozdíl mezi Frameworkem a vývojovým prostředím 14](#_Toc22819312)

[Tabulka 1.2 Rozdíl mezi architektonickými a návrhovými vzory 14](#_Toc22819312)

seznam PŘÍLOH

Příloha P I: Název přílohy

PŘÍLOHA P i: NÁZEV PŘÍLOHY

1. Pod pojmem systémový zdroj rozumíme např. CPU, GPU, systémovou paměť nebo periferie. [↑](#footnote-ref-1)
2. To zahrnuje např. editor kódu, debugger, kompilátor, interpret, nástroje pro správu verzí a často i nástroje pro vizuální návrh uživatelského rozhraní. [↑](#footnote-ref-2)
3. Tato tabulka je pouze obecným přehledem. Některá IDE se mohou zaměřovat na konkrétní jazyky. Některé frameworky naopak nabízí více flexibility apod. [↑](#footnote-ref-3)
4. Tento trend je mimo jiné podporován např. vývojem cloudových technologií a platformou jako službou (PaaS) které usnadňují vývoj a správu těchto aplikací. [↑](#footnote-ref-4)
5. V případě MVC architektury má prezentační vrstva View aktivnější roli. Může přímo komunikovat s Modelem, což vede k těsnějšímu provázání těchto komponent. [↑](#footnote-ref-5)
6. 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. [↑](#footnote-ref-6)
7. Slovo asynchronní v tomto kontextu znamená, že producent po odeslání zprávy nečeká na reakci spotřebitele, ale pokračuje dál ve své činnosti. [↑](#footnote-ref-7)
8. GOF (Gang of Four) je skupina autorů, kteří v roce 1994 publikovali knihu „Design Patterns: Elements fo Reussable Object-Oriented Software [4]. [↑](#footnote-ref-8)
9. Za všechny lze jmenovat např. návrhové vzory Factory, Singleton nebo Decorator. [↑](#footnote-ref-9)
10. Jedná se o akronym čtyř základních operací pro manipulaci s daty. Těmi jsou Create, Read, Update a Delete. [↑](#footnote-ref-10)
11. Za další implementace IoC můžeme považovat např. návrhové vzory Service Locator nebo Factory, představená v kap. 1.3.4. [↑](#footnote-ref-11)