Ekonomická fakulta
Faculty of Economics

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

Jihočeská univerzita v Českých Budějovicích
Ekonomická fakulta
Katedra aplikované matematiky a informatiky

Bakalářská práce

Příklady dobré praxe při vývoji aplikací na platformě .NET

Vypracovala: Hana Kamarádová
Vedoucí práce: Mgr. Radim Remeš

České Budějovice 2016

JIHOČESKÁ UNIVERZITA V ČESKÝCH BUDĚJOVICÍCH
Fakulta ekonomická
Akademický rok: **2012/2013**

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Hana KAMARÁDOVÁ**
Osobní číslo:	**E12263**
Studijní program:	**B6209 Systémové inženýrství a informatika**
Studijní obor:	**Ekonomická informatika**
Název tématu:	**Příklady dobré praxe při vývoji aplikací na platformě .NET**
Zadávající katedra:	**Katedra aplikované matematiky a informatiky**

Zásady pro vypracování:

Předmětem práce je rozbor a popis vhodných postupů při vývoji aplikací na platformě .NET. Jednotlivé případy budou demonstrovány na vytvořené aplikaci.

Metodický postup:

1. Studium odborné literatury.
2. Obecný popis metodik používaných při vývoji aplikací.
3. Analýza situací, případové studie, popis vhodných řešení demonstrovaných na vytvořené aplikaci.

Rozsah grafických prací: **dle potřeby**

Rozsah pracovní zprávy: **40 - 50 stran**

Forma zpracování bakalářské práce: **tištěná**

Seznam odborné literatury:

1. **BISHOP, J.** *C#: návrhové vzory.* **Vyd. 1. Brno: Zoner Press, 2010, 323 s. ISBN 978-80-7413-076-2.**
2. **KNUTH, Donald Ervin.** *Umění programování.* **Vyd. 1. Brno: Computer Press, 2008, xix, 648 s. ISBN 978-80-251-2025-5.**
3. **MARTIN, Robert C.** *Čistý kód.* **Vyd. 1. Brno: Computer Press, 2009, 423 s. ISBN 978-80-251-2285-3.**
4. **PECINOVSKÝ, Rudolf.** *Návrhové vzory.* **Vyd. 1. Brno: Computer Press, 2007, 527 s. ISBN 978-80-251-1582-4.**
5. *Pro C# and the .NET 4.5 framework.* **6. edition. Berkeley, Calif: APress. ISBN 14-302-4233-7.**
6. **RITCHIE, Stephen D.** *Pro .NET best practices.* **New York, NY: Apress, c2011, xx, 350 p. ISBN 978-143-0240-242.**

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

Datum zadání bakalářské práce: **2. ledna 2013**

Termín odevzdání bakalářské práce: **15. dubna 2014**

doc. Ing. Ladislav Rolínek, Ph.D.
děkan

L.S.

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

prof. RNDr. Pavel Tlustý, CSc.
vedoucí katedry

V Českých Budějovicích dne 29. března 2013

Abstrakt

Náplní bakalářské práce je poukázat na zásady a pravidla, jak navrhovat a vyvíjet aplikace v objektově orientovaném jazyce co nejlépe. Práce je zaměřena na vybrané principy návrhu softwaru (software design principles), které dávají rady a doporučení, jak předcházet problémům souvisejících s návrhem aplikace. A dále na některé důležité pojmy související s objektově orientovaným vývojem. Uplatněním těchto přístupů na konkrétní funkční aplikaci prakticky demonstruji jejich použití v jazyku C# v prostředí Microsoft Visual Studio 2015 tak, abych představila dobrou praxi.

Klíčová slova

metodika vývoje softwaru, UML, návrhové principy, návrhové vzory

Prohlášení

Prohlašuji, že svoji bakalářskou práci jsem vypracovala samostatně pouze s použitím pramenů a literatury uvedených v seznamu citované literatury.

Prohlašuji, že v souladu s § 47 zákona č. 111/1998 Sb. v platném znění souhlasím se zveřejněním své bakalářské práce, a to v nezkrácené podobě elektronickou cestou ve veřejně přístupné části databáze STAG provozované Jihočeskou univerzitou v Českých Budějovicích na jejích internetových stránkách, a to se zachováním mého autorského práva k odevzdanému textu této kvalifikační práce. Souhlasím dále s tím, aby toutéž elektronickou cestou byly v souladu s uvedeným ustanovením zákona č. 111/1998 Sb. zveřejněny posudky školitele a oponentů práce i záznam o průběhu a výsledku obhajoby kvalifikační práce. Rovněž souhlasím s porovnáním textu mé kvalifikační práce s databází kvalifikačních prací Theses.cz provozovanou Národním registrem vysokoškolských kvalifikačních prací a systémem na odhalování plagiátů.

15. dubna 2016 ……

Hana Kamarádová

Obsah

1. Úvod

Obsahem této bakalářské práce je uvedení některých příkladů, které patří mezi základy dobré praxe při programování v objektově orientovaných jazycích. Pro ukázku využití těchto principů jsem si vybrala jeden z nejpoužívanějších jazyků, jenž je součástí platformy .NET.

C# je moderní programovací jazyk, který má mnohočetné uplatnění od vývoje desktopových aplikací, vývoje softwaru pro mobilní zařízení, přes databázové programy až třeba k využití při tvorbě webových aplikací.

Tato oblast je velice obsáhlá, a možností, kterými by se tato práce mohla zabývat, je nepřeberně. Zmínit mohu například projekt platformy MONO, kde její .NET nástroje běží na různých operačních systémech, dále .NET pro mobilní zařízení jako MonoTouch, MonoDroid, pak webový framework ASP.NET MVC či SignalR, a to jsem obsahově jen u začátku výčtu různých možností. Pokud bych se ale chtěla zaobírat trochu blíže všemi tématy, přesáhla bych nejen rozsah této práce, ale snad i více než tisíc stran.

Protože nebylo obsahově možné věnovat se všem tématům, zaměřila jsem se především na principy SOLID, KISS, DRY a dále vzor MVC, jež patří k jedněm z předních konceptů, které dávají doporučení, jak předcházet problémům jako je špatná udržovatelnost zdrojového kódu, množství chyb, rychlost vývoje a v neposlední řadě výkonu aplikace. Důvod, proč se zabývat tímto tématem, byl jasný. Je vhodné (nejen z uvedených problémů) používat doporučené principy a praktiky, které snižují problémy tzv. „špinavého“ kódu.

Uplatnění těchto přístupů je prakticky předvedeno na aplikaci Pexeso napsané v jazyce C#. Nejdříve bylo nutné původní verzi aplikace zdokumentovat a vytvořit její model. Na základě analýzy návrhu a implementace jsem mohla s ohledem na výše sledované aspekty navrhnout změny. Tyto jsem uplatnila v nové verzi aplikace, na které tak demonstruji jejich využití prakticky.

Na závěr provedu zhodnocení, jak se mi uplatnění těchto přístupů a doporučení podařilo, zejména s ohledem na zvýšení kvality návrhu, jeho budoucí rozšíření a opakovatelnou využitelnost kódu.

2. Teoretická část

Příkladů dobré praxe existuje při vývoji aplikace nespočetné množství, a také se na ně můžeme dívat z mnoha úhlů pohledu. Moje práce je zaměřena především na principy vývoje při návrhu aplikace. Nejdříve si ale shrňme, kde všude se s těmi příklady můžeme zaobírat.

2.1 Disciplíny z oblasti objektově orientovaného vývoje

Dávno jsou pryč doby, kdy se jeden programátor staral o jeden program, který většinou sloužil pouze k vyřešení určitého problému nebo k automatizaci postupu řešení. Dnes jsou již informační systémy natolik rozsáhlé a složité, že se o jejich vývoj a údržbu starají celé týmy datových analytiků, softwarových architektů, programátorů, testerů a mnoho dalších uživatelů, aby dokázali vyprodukovat obrovské množství kódu.

Přestože je každý softwarový projekt unikátní a jednotlivé etapy se liší, v zájmu zlepšeného řízení bylo vytvořeno několik modelů životního cyklu vývoje softwaru (Kay, 2012).

Cílem modelu životního cyklu je definovat jednotlivé etapy vývoje softwaru. Dále potom pro každou etapu nutné činnosti, které by v ní měly proběhnout. Tyto činnosti dělíme přibližně do pěti etap následovně:

- **Analýza a specifikace požadavků** - první etapa při vývoji softwaru. Zabýváme se požadavky zákazníka - analyzujeme, definujeme, specifikujeme. Součástí by měla být i studie proveditelnosti, zda je vhodné a možné se do projektu vůbec pouštět a zanalyzování možných rizik. Nezbytností je též naplánování akceptačního testování.
- **Architektonický a podrobný návrh** - V architektonickém návrhu dochází k dekompozici systému a vymezení funkcionality jednotlivým podsystémům včetně vztahů mezi nimi. Je důležité naplánovat, jak bude vypadat testování celého systému.

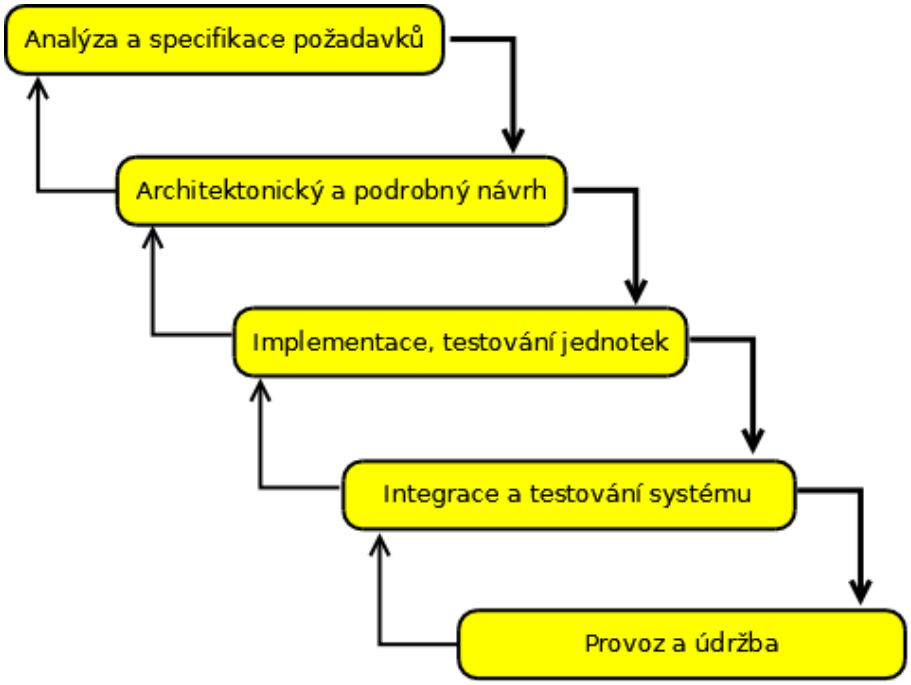
  V podrobném návrhu se zabýváme specifikací jednotlivých podsystémů a plánují se jednotlivé práce na implementaci.
- **Implementace** - programování, realizace, dokumentace a testování implementovaných součástí.
- **Integrace a testování** - spojení veškerých částí implementací dohromady a následné otestování celého systému. Dále akceptační testování uživatelem,

na jehož základě zákazník rozhodne, zda systém převzít či nikoliv. V kladném případě následuje instalace u zákazníka a školení uživatelů.

- **Provoz a údržba** - řešení problémů, které vznikají následkem používání systému. Zahrnuje opravu chyb a přizpůsobování softwaru měnícím se požadavkům (Křena & Kočí, 2006).

2.1.1 Vodopádový model životního cyklu

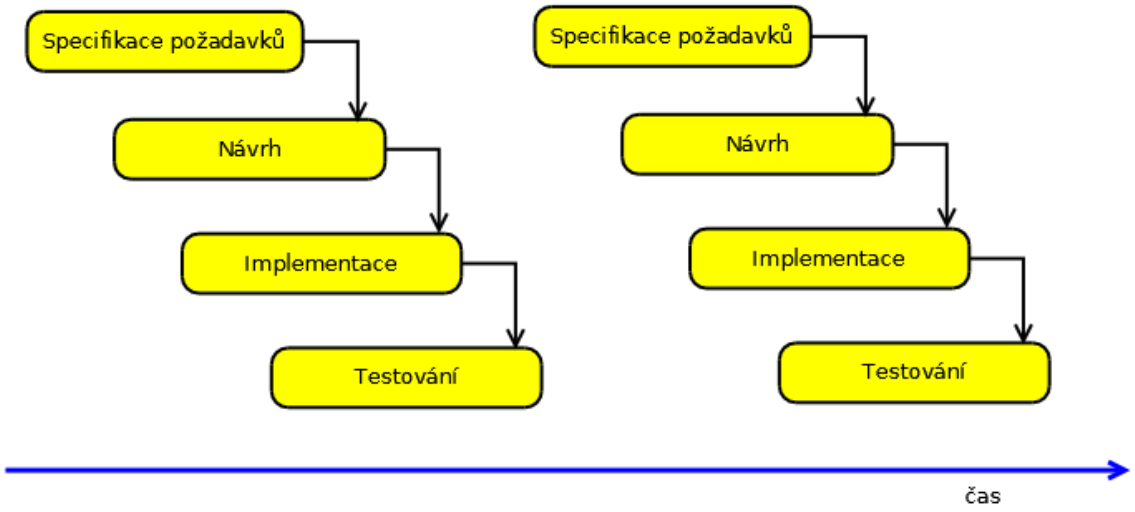
Vodopádový model (Obrázek 1) je základním modelem životního cyklu softwaru. Je to nejpřirozenější a pravděpodobně i nejstarší model životního cyklu vývoje softwaru. Etapy jsou seřazeny za sebou, a každá následující začíná, až skončí etapa předchozí (Křena & Kočí, 2006).



Obrázek 1: Vodopádový model životního cyklu softwaru (Křena & Kočí, 2006)

2.1.2 Iterativní model životního cyklu

Hlavním problém vodopádového modelu je, že uživatel vidí spustitelnou verzi příliš pozdě. Iterativní model (Obrázek 2) se snaží tento problém odstranit rozdělením procesu do iterací. Každou iteraci můžeme chápat jako instanci vodopádového modelu. Výhodou tohoto řešení je, že uživatel může vidět po každé iteraci prozatímní verzi a upřesnit tak své požadavky. Nevýhodou ovšem je, že nemusí být úplně jednoduché celý proces rozdělit do iterací. Může dojít ke zhoršení struktury systému v porovnání s tím, kdyby byly veškeré požadavky dostupné na začátku vývoje (Křena & Kočí, 2006).



Obrázek 2: Iterativní model životního cyklu softwaru (Křena & Kočí, 2006)

2.1.3 Inkrementální model životního cyklu

Jedná se o model, který se velmi podobá předchozímu iterativnímu modelu. Struktura systému je navržena tak, aby šlo pracovat na sobě nezávislých částech, které se postupně zpřístupňují uživateli. Na rozdíl od iterativního modelu může být struktura systému navržena lépe bez větších negativních dopadů při pozdějším upřesnění požadavků uživatelem (Křena & Kočí, 2006).

2.1.4 Spirálový model životního cyklu

Spirálový model připomíná iterativní a inkrementální model životního cyklu, na rozdíl od nich uživatel netestuje model s omezenou funkcionalitou, ale prototyp, který je následně zahozen. Tím se zásadně liší od předchozích modelů, jelikož nový software se vytváří od začátku. Při tomto vývoji se jednotlivé etapy stále opakují, avšak vždy na vyšší úrovni problematiky. V každém cyklu se stanovují podmínky pro zvládnutí iterace. Značný důraz je kladen na analýzu rizik (Křena & Kočí, 2006).

2.1.5 UP (Unified process)

Proces iterace a přírůstku je podstatou metody UP. Primární myšlenka UP je velmi jednoduchá: člověk je schopen lépe řešit malé problémy než velké. Cílem je rozložení velkého projektu do více menších. Každý z těchto menších projektů je považován za iteraci. Základním předpokladem je skutečnost, že každá iterace obsahuje veškeré prvky jako normální softwarový projekt (specifikaci požadavků, architektonický a podrobný návrh, implementace, integrace a testování, interní nebo externí uvedení). Finální verze

systému se vytváří postupně z jednotlivých iterací, které se na sebe nabalují (Arlow & Neustadt, 2007).

2.1.6 RUP (Rational Unified Process)

Nejznámější použitím Unified process (UP) je komerční verze Rational Unified Process (RUP). Jedná se o metodiku software vytvořenou společností Rational Software Corporation, kterou převzala společnost IBM. Na rozdíl od UP již obsahuje použitelné nástroje, veškeré standardy a další nezbytnosti. Rozsáhlé uživatelské prostředí, stejně jako podrobná dokumentace k jednotlivým nástrojům je, samozřejmostí. Hlavní rozdíl mezi UP a RUP je, že v komerční verzi je již mnohé řešeno do detailu, jinak tyto dvě metodiky převážná část podstatných věcí spojuje. RUP je vhodná spíše pro větší vývojové týmy a rozsáhlejší projekty. Důraz je kladen na vizualizaci systému pomocí jazyka UML (Arlow & Neustadt, 2007).

2.1.7 Agilní metodiky

Dlouhá doba vývoje klasických modelů životního cyklu přispěla k tomu, že se stávaly čím dál více komplexnějšími. Avšak při vytváření menších projektů tyto metodiky kvalitu výsledného produktu spíše snížily. Vývoj se značně prodražoval bez většího efektu. Tyto problémy řeší agilní metodiky, které kladou hlavní důraz na člověka. Umožňují rychlejší vývoj a zároveň jsou schopny reagovat na změny požadavků měnících se v průběhu vývoje softwarového produktu (Křena & Kočí, 2006).

Extrémní programování

Jedna z agilních metodologií, která je postavena na tradičních činnostech, jež jsou dovedeny do extrému. Vyzdvihuje týmovou práci. Manažer, zákazník a vývojáři systému jsou rovnocenní partneři, proto není kladen důraz na podrobnou specifikaci, kterou má možnost uživatel průběžně upřesňovat. Klade důraz na zákazníkovu spokojenost. Dává přednost fungujícímu softwaru před komplexní dokumentací. Hlavními zásadami jsou jednoduchost návrhu, neustálá kontrola, jenž vede k refaktorizaci. Častá integrace nových funkcí systému a testování, kdy se testy píší před implementací vlastního kódu (Wells, 2013).

SCRUM

Metodika založená na krátkých vývojových cyklech, kterým říkáme „sprinty“. Každý sprint trvá asi 2 - 4 týdny. Na začátku jsou stanoveny cíle, které se vyhodnotí na konci sprintu, kdy je funkční aplikace předvedena zákazníkovi. V tuto chvíli může klient

zahrnout svoje připomínky. Sprint se vyhodnotí a naplánují se úkoly do dalšího období. Klient má po každém sprintu k dispozici funkční verzi projektu. Vývoj je možné kdykoliv, kdy zákazník uzná za vhodné, dokončit.

Výběr vhodné metodiky je jednou ze zásadních věcí, která zajistí úspěšné dokončení projektu. „Dobrá metodika předpokládá průběžné ověřování kvality jednotlivých výstupů při práci na projektu. Předejde se tak nepříjemnému překvapení v posledních fázích, kdy již většinou není jednoduché předělávat“ (Mádl, 2003).

2.2 Principy návrhu softwaru

Návrhových vzorů, principů a doporučení pro programování v objektově orientovaném jazyce existuje hned několik. Většina z nich se vzájemně prolíná a doplňuje. Jmenujme třeba GRASP, jenž se zaměřuje na rozdělení zodpovědností jednotlivým třídám a objektům. Návrhový princip POGE nám zase říká, že dostatečně dobré řešení obsahuje pouze to, co je pro řešení nezbytně nutné, a žádnou další zbytečnou funkcionalitu. Toto se podobá principu YAGNI, jenž doporučuje se zamyslet nad tím, jestli se funkcionalita navíc v rozumném horizontu využije, či nám bude jen komplikací. Princip SSOT radí, aby v systému existoval právě jeden centrální, objektivní zdroj informací. „Demétřino pravidlo“, známé též jako „princip minimální znalosti“, vysvětluje, že objekt by měl vědět co nejméně o vlastnostech a struktuře čehokoliv jiného.

V další části práce se zaměříme zejména na principy SOLID, KISS a DRY. Vybrala jsem si právě tyto, jelikož vzhledem k rozsahu práce se není možné zaměřit na úplnou řadu vhodných doporučení a rad. Některé z nich není možné ukázat na menších projektech, u jiných záleží především na zadání a zhodnocení využitelnosti v budoucnu.

2.2.1 Základní principy SOLID

Solid (Single responsibility, **O**pen-closed, **L**iskov substitution, **I**nterface segregation and **D**ependency inversion**)** je zkratka počátečních písmen pěti principů, rad a doporučení, které popsal R. C. Martin (Martin & Martin, 2006).

Jedná se o sadu doporučení, které mají především předcházet vzniku špatného návrhu kódu, ke kterým patří především:

- Rigidity (ztuhlost) - tendence softwaru být obtížně měnitelný, a to dokonce i jednoduchými způsoby. SW je ztuhlý, pokud jediná změna způsobí kaskádu následných změn v závislých modulech. Čím více modulů je třeba změnit, tím větší je ztuhlost konstrukce.
- Fragility (křehkost) - tendence programu přestat na mnoha místech pracovat správně, pokud je udělána jednoduchá změna.
- Immobility (imobilita) - program obsahuje části, které mohou být užitečné i v jiných systémech, ale pracnost a riziko spojené s oddělením této části jsou příliš vysoké. Tento nešťastný jev se vyskytuje velmi často.
- Viscosity (viskozita) - vysoká viskozita SW znamená, že je navržený tak, že změny, které v něm musí někdo provést, vedou ke změně návrhu, se kterým byl původně vytvořen.
- Needless complexity (zbytečná složitost) - byly přidány části, které se v současnosti nepoužívají. Byly přidány pro případné lehčí rozšíření v budoucnu, ale pravděpodobně nebudou využity a nyní jen zesložiťují a zhoršují čitelnost kódu.
- Needless repetition (zbytečné opakování) - kód obsahuje několik stejných nebo podobných částí na různých místech, jež je problematické udržovat při změně.
- Opacity (nejasnost) - nerefaktorovaný kód, který je později nebo pro jiné jen těžko srozumitelný.

The Single Responsibility Principle (SRP)

„Princip jediné odpovědnosti (SRP - Single Responsibility Principle) říká, že třída nebo modul by měly mít jeden a jen jeden důvod ke změně“ (Martin, 2009, s. 154).

Nejzákladnější z principů SOLID říká, že pokud má třída více než jednu logickou zodpovědnost, zodpovědnosti se spojí dohromady. Změny jedné zodpovědnosti mohou narušit či potlačit schopnost třídy vyhovovat ostatním úkolům. Tento druh spojení vede ke křehkému návrhu, kdy program skončí neočekávaně i při sebemenší změně (Martin & Martin, 2006).

SRP instruuje vývojáře psát kód tak, že má jeden a jen jeden důvod ke změně. Pokud třída má více než jeden důvod ke změně, má více než jednu zodpovědnost. Třídy s více než jednou zodpovědností by měly být rozloženy do menších tříd. Každá třída by měla mít jednu zodpovědnost a jeden důvod ke změně (Hall, 2014).

Důvody pro dodržování tohoto principu (Dresler, 2011b):

- Zvýšení čitelnosti kódu - Pokud má třída pouze jednu zodpovědnost, bude snadnější porozumět systému jako celku. Třídy jsou mnohem efektivnější, jelikož dělají to, co jejich název deklaruje, a při čtení či úpravách se nemusíme zaobírat funkcionalitou, která nás aktuálně nezajímá.
- Lepší robustnost kódu - Pokud bude v budoucnu požadována jakákoliv změna, ovlivní pouze bázovou třídu, nikoliv podtřídy.
- Napomáhá maximalizovat soudržnost - Třída dělá právě takové operace, které od ní očekáváme. Nedělá operace, které bychom od ní neočekávali, a předpokládali bychom, že se nachází v jiné třídě. S kódem se potom lépe pracuje i lidem, kteří danou aplikaci nevytvářeli.
- Testovatelnost - Třída je lépe testovatelná, pokud má právě jednu zodpovědnost, jelikož pro její funkcionalitu budeme potřebovat právě jeden test.
- Snížení vzájemné provázatelnosti tříd - Každá třída ví jen o několika málo třídách, které spolu mají něco společného.

The Open-Closed Principle (OCP)

Princip uzavřenosti a otevřenosti (OCP) říká, že softwarové třídy, moduly či funkce by měly být otevřené pro rozšíření, ale uzavřené pro změny.

Pokud malá změna programu vyvolá spoustu změn závislých modulů, je znát, že je kód obtížné změnit (je rigidní). OCP doporučuje refaktorovat systém tak, že další změny tohoto druhu nebudou příčinou pro více modifikací.

Když je OCP optimalizovaný dobře, při potřebné změně je pouze přidán nový kus kódu, bez toho aniž by se musel změnit starý kód, který funguje.

Dvě základní vlastnosti OCP:

- Otevřený pro rozšíření - Chování modulu může být rozšířeno, pokud potřebujeme přidat novou funkcionalitu.
- Uzavření pro modifikaci - Rozšíření o nové chování modulu se nijak neprojeví na zdrojovém ani binárním kódu.

Těchto dvou zvláštních pravidel můžeme dosáhnout pomoci abstrakce. Míru uplatnění principu uzavřenosti a otevřenosti zvyšuje dodržování architektury MVC (Martin & Martin, 2006), viz. kapitola 2.3.4 MVC přístup při tvorbě aplikací.

The Liskov Substitution Principle (LSP)

Substituční princip Barbary Liskovové, který napsala již v roce 1988 říká, že „funkce (metody), které používají ukazatele nebo reference na bázovou třídu, musí být schopny použít objekty jejich podtříd bez jejich znalostí“ (Dresler, 2011a). Při využívání dědičnosti se tedy musí vytvářet pouze takové třídy, jejichž instance mohou stát na místě instancí kterékoli jejich nadtřídy.

The Interface Segregation Principle (ISP)

„Klienti by neměli být nuceni do závislosti na rozhraních, která nepoužívají“ (Dresler, 2011c).

Tento princip se zabývá nevýhodou příliš „tlustého“ rozhraní. Třídy, jejichž rozhraní nejsou soudržná, mají „tlustá“ rozhraní.

The Dependency-Inversion Principle (DIP)

Princip obrácené závislosti (ISP) popsal Martin takto:

- Modely vyšší úrovně (vrstvy) by neměly být závislé na modulech nižší úrovně (vrstvy). Oboje by mělo být závislé na abstrakci.
- Abstrakce by neměla záviset na detailech. Detaily by měly záviset na abstrakci (Martin & Martin, 2006).

Uplatnění tohoto principu ukazuje například návrhový vzor abstraktní továrna nebo fasáda, které jsou blíže popsány v kapitole 2.3.1 Vzory vztahující se k vytváření objektů a 2.3.2 Strukturální vzory.

2.2.2 Seznámení s principem KISS

KISS popisuje jako nebýt složitý nebo komplikovaný. Absence zbytečných prvků. Častou chybou je vysvětlovat si pojem jednoduchý jako zredukování již hotových částí. Ba naopak, jednoduchost vyžaduje dobrý nápad a snahu něco snadno pochopitelného a upravitelného vytvořit. To, co je jednoduché, nemusí být ovšem snadné. Patří ovšem mezi nejlepší a nejvíce použitelné, ať již se jedná v podstatě o cokoliv (De Keyser & Springael, 2010).

„Všechno by mělo být uděláno tak jednoduše, jak jen to jde. Ne však jednodušeji.“

Albert Einstein

KISS je anglický akronym, jehož tradiční rozšíření zní: „Keep It Simple, Stupid!“, jenž znamená: „Zachovej to jednoduché, hlupáku!“. Pravděpodobně z důvodů neúcty či urážky je častěji vysvětlován jako např.: „Zachovej to jednoduché a bezpečné!“, „Zachovej to jednoduché a systematické!“ nebo jako „Zachovej to jednoduché a elegantní!“

Základní princip říká, že pokud teorie, konstruktory, modely, návrh tříd, atd. jsou komplikované, jsou pravděpodobně špatně navrženy. Vysvětluje, že jednoduchost by měla být hlavním cílem návrhu.

2.2.3 DRY aneb neopakuj se!

Nejznámější a jedno z nejdůležitějších pravidel softwarového návrhu zní: Neopakuj se! V jakémkoliv systému by v ideálním případě každý kus kódu měl existovat pouze jednou.

Jedním z dobrých důvodů, proč se držet tohoto pravidla je, že pokud se nám podaří použít starý kód, nemusíme psát nebo měnit tolik kódu, když přidáme nové funkce. To znamená, že v projektu budeme mít méně míst náchylných k chybám.

Též nám pomáhá s flexibilitou našich návrhů. Pokud potřebujeme měnit způsob, jakým náš program funguje, změníme pouze nějaký kód v jednom místě, místo toho, abychom museli projít celý program a provést více změn (Kanat-Alexander, 2012).

Mnoho dobrých návrhů je založených na dodržování tohoto pravidla. Někteří se ale občas drží spíše přísloví:

„O mnohé věci se nepokusíme nikoli proto, že jsou obtížné, ale obtížné jsou proto, že se o ně nepokusíme.“

Seneca

Příčiny duplicitního kódu

- Pocit, že musíme opakovat. Že dané prostředí, zadání projektu, dokumentace toto vyžaduje.
- Informaci duplikujeme, aniž bychom si to uvědomovali.
- Jsme líní a často nám přijde jednodušší danou věc napsat znovu. Teprve, když již něco opakujeme potřetí, se často začínáme zamýšlet, zda by se nenašlo vhodnější řešení daného problému. Duplikaci z nepozornosti si většinou dobře

uvědomujeme. Často však nejsme ochotni strávit více času nad daným problémem, který by nám později například urychlil běh aplikace.

- Poslední důvod vyvstává při řešení týmového projektu. Tento problém se z daných čtyř kategorií nejhůř odhaluje. Lze částečně řešit lepší komunikací v týmu (Hunt & Thomas, 2007).

2.2.4 Ostatní principy návrhu softwaru

Pro úplnost doplním, že existují ještě další principy návrhu. K nim patří GRASP - General Responsibility Assignment Software Patterns (or Principles). Jedná se o sadu návrhových principů a doporučení, jež doporučují, jak rozdělit zodpovědnost mezi třídami a objekty.

Dále bych neměla opomenout YAGNI - You Aren´t Gonna Need It. Tento princip nám říká, že je vhodné se dívat do budoucnosti. Zamyslet se nad tím, jestli opravdu bude daná věc potřebná. Velice často se stává, že ji nikdy nevyužijeme a zbytečná funkcionalita nám akorát přináší komplikace (Miguel, 2014).

Zmiňme ještě CoC - Convention over Configuration, jenž říká, že dodržování konvencí usnadňuje orientaci v kódu. Tzn. kód by měl dělat to, co od něj většina programátorů očekává.

2.3 Návrhové vzory, jež by bylo vhodné zmínit

Návrhový vzor popisuje podstatu řešení problému, který nastává opakovaně. Mezi hlavní cíle návrhových vzorů patří znovupoužitelnost, udržitelnost a flexibilita, které pomáhají programátorům k elegantnějšímu řešení.

Bishopová (2010) ve své knize považuje za hlavní 23 vzorů, které jsou rozděleny do tří skupin. Dnes již však existují další vzory, které jsou aplikovatelné na určitou oblast, jako je třeba softwarová architektura či bezpečnost.

Rozdělení:

- Vytvářecí vzory (Prototyp, Tovární metoda, Jedináček, Abstraktní továrna, Stavitel)
- Strukturální vzory (Dekorátor, Proxy, Most, Kompozit, Muší váha, Adaptér, Fasáda)

- Vzory vztahující se k chování (Strategie, Stav, Šablonová metoda, Zřetězení odpovědností, Příkaz, Iterátor, Prostředník, Pozorovatel, Návštěvník, Interpret, Memento)

V následujících třech podkapitolách představuji vybrané vzory blíže.

2.3.1 Vzory vztahující se k vytváření objektů

Jedináček (Singleton)

„Jedináček specifikuje, jak vytvořit třídu, která bude mít nejvýše jednu instanci. Tato instance přitom nemusí být vlastní instancí dané třídy“ (Pecinovský, 2007, s. 107).

Tovární metoda (Factory Method)

„Definuje rozhraní pro vytváření objektu. Rozhodnutí, u které třídy se má spustit její instance, ale přenechává podtřídám. Tovární metoda umožňuje třídě, aby odložila rozhodnutí o vytvoření instance na své podtřídy“ (Gamma et al. 2003, s. 116).

Abstraktní továrna (Abstract Factory)

„Poskytuje rozhraní pro vytváření řad příbuzných nebo závislých objektů, aniž by se musely specifikovat konkrétní třídy“ (Gamma et al. 2003, s. 100).

2.3.2. Strukturální vzory

Zástupce (Proxy)

„Zavádí zástupce, který odstiňuje objekt od jeho uživatelů a sám řídí přístup uživatelů k danému objektu“ (Pecinovský, 2007, s. 189).

Fasáda (Facade)

„Účelem vzoru je poskytnout různé vysokoúrovňové pohledy na subsystémy, jejichž detaily jsou schovány před uživateli. Operace, které mohou být žádoucí z perspektivy uživatele, mohou být sestaveny z odlišných částí jednotlivých subsystémů“ (Bishopová, 2010, s. 111).

Dekorátor (Decorator)

„Dynamicky připojí k objektu další povinnosti. Dekorátoři poskytují při rozšiřování funkcí tvárnou alternativu k tvorbě podtříd“ (Gamma et al. 2003, s. 172).

2.3.3 Behaviorální vzory

Příkaz (Command)

„Zabalí metodu do objektu, takže s ní pak lze pracovat jako s běžným objektem. To umožňuje dynamickou výměnu používaných metod za běhu programu a optimalizaci přizpůsobení programu požadavkům uživatele“ (Pecinovský, 2007, s. 195).

Iterátor (Iterator)

„Zprostředkuje jednoduchý a přehledný způsob sekvenčního přístupu k objektům uloženým v nějaké složité struktuře (většinou v kontejneru), přičemž implementace této struktury zůstane klientovi skryta“ (Pecinovský, 2007, s. 203).

Stav (State)

„Řeší výrazný rozdíl mezi chováním objektu v různých stavech zavedením vnitřního stavu jako objektu reprezentovaného instancí některé ze stavových tříd. Změnou stavu objektu pak řeší záměnou objektu reprezentujícího stav“ (Pecinovský, 2007, s. 221).

Pozorovatel (Observer)

„Vzor Observer definuje vztah mezi objekty tak, že když jeden změní svůj stav, je to oznámeno všem ostatním podle potřeby. Obvykle existuje jediný pozorovaný objekt měnící svůj stav a mnoho pozorovatelů, kteří si přejí být informování při změně“ (Bishopová, 2010, s. 236).

2.3.4 MVC přístup při tvorbě aplikací

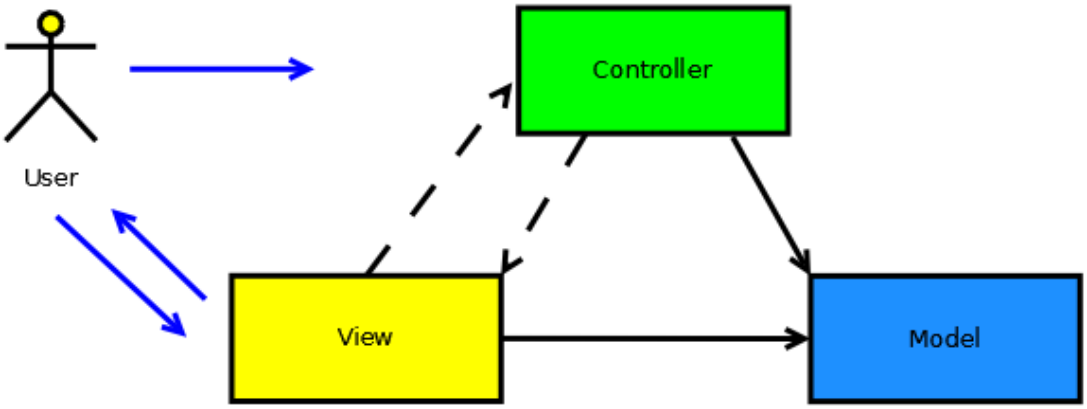
Zřejmě nejrozšířenější návrhový vzor používaný při objektově orientovaném návrhu je Model-View-Controller. MVC je akronymem tří částí aplikace, Model, View a Controller. Hlavním cílem MVC je oddělení logiky systému od jeho zobrazení na výstupu uživateli.

- Model - zaštiťuje celou logiku systému. Představuje data, která aplikace zpracovává. Může se jednat o tabulku, textový soubor nebo třeba obrázek. Neví, odkud se vzala ani v jaké formě budou zobrazena uživateli.

  Datový model je nezávislý na logice konkrétní aplikace, proto jej je též možné využít i v jiné aplikaci, která pracuje se stejnými daty, aniž by bylo třeba jakékoli změny. To, že je můžeme použít vícenásobně, je jednou z primárních vlastností objektově orientovaného programování.

- Controller - propojuje a řídí celý systém. Měl by obsahovat funkční logiku celé aplikace. Toto je část, která by se měla podívat do modelu, s jakými daty bude pracovat, zpracovat je a na základě akce uživatele zobrazit pomoci View do grafického výstupu.
- View - jeho úkolem je zobrazení dat na výstup. Dostane data a nestará se odkud. Výsledek od Controlleru zobrazí uživateli. Pro uživatele aplikace je tato podstatná část jediná, se kterou komunikuje. GUI nesmí být nijak závislé. Pokud například na žádost uživatele změníme zelená tlačítka za modrá, nebude to mít žádný vliv na žádnou jinou část z těchto tří objektů, jelikož objekty jsou nezávislé (Čada, 2009).

Na následujícím obrázku (Obrázek 3) uvádím architekturu MVC v obvyklém znázornění v interakci s uživatelem.



Obrázek 3: Návaznost MVC v interakci s uživatelem (Bernard, 2009)

2.4 Klíčové koncepty objektově orientovaného návrhu

Dobrý návrh je klíčem k úspěšnému vývoji a realizaci softwaru. Řada hlavních zásad pomáhá dosáhnout kvalitního návrhu. Mezi klíčové koncepty návrhu patří abstrakce, dekompozice, modularita, skrývání informací a oddělovací politika a postupy.

2.4.1 Abstrakce

Řešený problém analyzujeme a klasifikujeme do abstraktních datových struktur, tj. v objektově orientovaném programování do objektů. Objekt je abstrakcí části řešeného problému. Ten má potom zodpovědnost za řešení určité konkrétní části

z celého problému. Cílem abstrakce je bez ztráty významu zjednodušit pohled na celý systém.

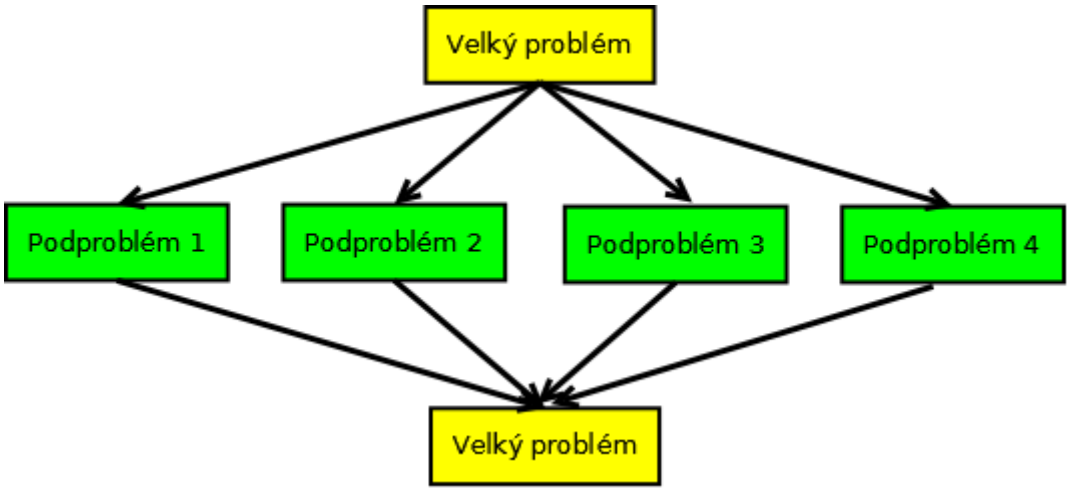
2.4.2 Dekompozice

Dekompozice znamená rozložení problému na menší podproblémy. Rozklad často tvoří základ pro organizaci projektu. Dobrá dekompozice minimalizuje závislost mezi komponentami.

Mezi přední výhody dekompozice patří, že různí lidé mohou pracovat nezávisle na odlišných podproblémech. Projekt je možno paralelizovat. Umožňuje snadnější údržbu projektu.

Avšak špatně pochopený problém se těžko rozkládá. Každý podproblém by měl být ve stejné úrovni detailu (Vliet, 2007; Levine & Gill, 2000).

Následující obrázek (Obrázek 4) ilustruje příklad dekompozice většího problému na menší.



Obrázek 4: Příklad dekompozice velkého problému na podproblémy

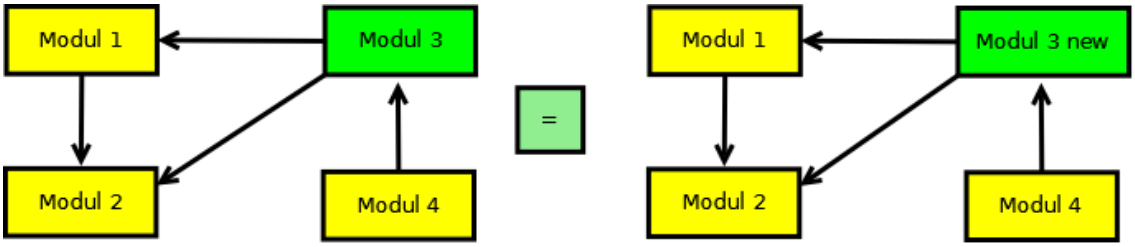
2.4.3 Skrývání informací

Skrývání informací je proces, při kterém jsou skryty detaily objektů nebo funkcí. Je to technika, která snižuje složitost problému. Je příkladem, jak použít abstrakci v návrhu softwaru. Moduly by spolu měly komunikovat pouze přes dobře definované rozhraní. Jedním z hlavních mechanismů je zapouzdření. Programátor se může soustředit na nový objekt, aniž by se musel obávat skrytých detailů (Vliet, 2007; Levine & Gill, 2000).

2.4.4 Modularita

Když je systém složený z modulů, tak by mělo být možno jeden modul odstranit a nahradit jiným, aniž by se to dotklo jiných modulů (Levine & Gill, 2000).

Na obrázku (Obrázek 5) níže představuji situaci, kdy zaměněním „Modulu 3“ za „Modul 3 new“, nejsou okolní moduly nijak ovlivněny.



Obrázek 5: Příklad záměny modulů bez dopadu na okolí

2.4.5 Oddělovací politiky a postupy

Více postupů může být implementováno stejným mechanismem. Mějme různé činnosti, např. plánování procesů a stránkování paměti. K obojímu může být použit stejný mechanismus nahrazovací politiky.

Naopak různé mechanismy jako je vázaný seznam či pole můžu použít k jedné činnosti a přitom dosáhnout stejného výsledku (Levine & Gill, 2000).

2.5 Ostatní vybrané pojmy

2.5.1 Metriky měření kvality kódu

Metriky kódu poskytují vývojářům lepší přehled o vyvíjeném kódu. Využití metrik napomáhá lépe zjistit, které metody by se měly přepracovat nebo lépe otestovat. S jejich pomoci lze více porozumět aktuálnímu stavu projektu, identifikovat potenciální rizika či sledovat pokrok během vývoje (MSDN - the microsoft developer network, 2016a).

Maintainability index (index udržovatelnosti)

Udává index hodnoty mezi 0 a 100, kde 100 reprezentuje relativně snadnou údržbu kódu. Čím vyšší hodnota, tím lepší udržovatelnost. Barevně odlišené hodnocení může být použito k rychlé identifikaci problémových míst v kódu.

Cyclomatic complexity

Měří strukturální složitost kódu. Je určena na základě výpočtu počtu různých cest kódu v toku programu. Program, který má složitý tok řízení, bude vyžadovat další testy, aby se dosáhlo podobného pokrytí kódu, a bude hůře udržovatelný.

Depth of Inheritance (hloubka dědičnosti)

Udává počet definic tříd, která sahají až ke kořeni hierarchie dané třídy. Čím větší hloubka hierarchie, tím obtížnější může být pochopit, kde jsou konkrétní metody nebo pole definovány nebo předefinovány.

Class Coupling (párování tříd)

Měří párování do unikátních tříd prostřednictvím parametrů, lokálních proměnných, návratových typů, volání metod, generického nebo šablonového vytváření instancí, základních tříd, implementace rozhraní, polí definovaných na externích typech a dekorace atributů. Dobrý design softwaru stanovuje, že druhy a metody by měly mít vysokou soudržnost a nízké párování. Vysoké párování indikuje design, který je obtížné znovu použít a udržovat, kvůli množství vzájemných závislostí na jiných typech.

Lines of Code

Ukazuje přibližný počet řádků kódu. Je založený na IL kódu, a proto není přesný. Velmi vysoký počet řádků může znamenat, že metoda se snaží dělat moc práce a je vhodné ji rozdělit. Může to též znamenat, že metoda bude hůře udržovatelná.

2.5.2 Reverse engineering a Refactoring

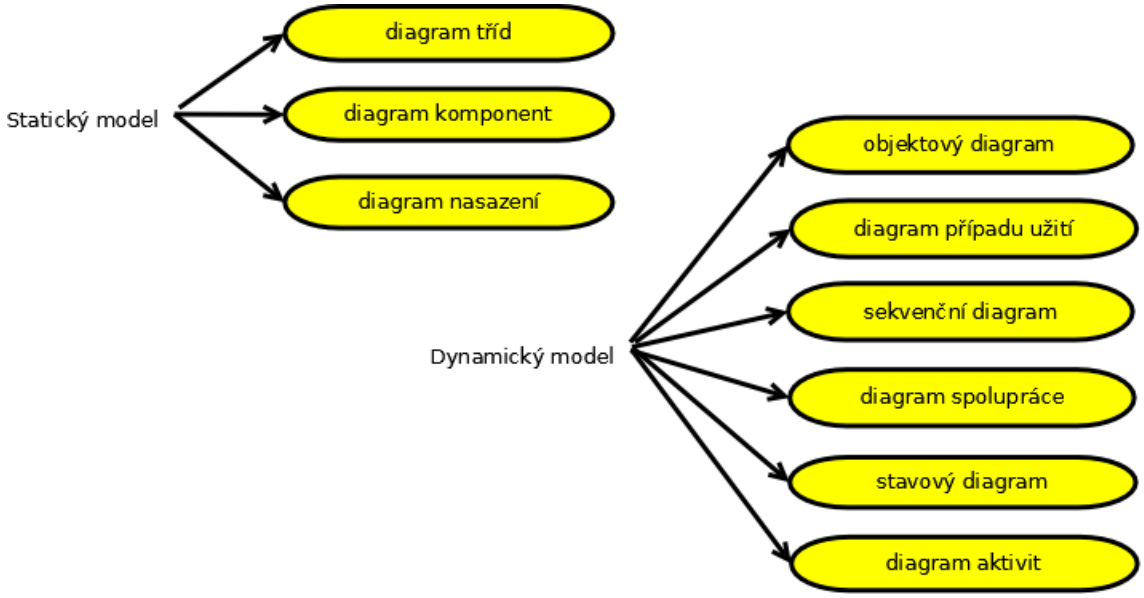
“Zpětné inženýrství je definováno jako proces analýzy předmětného systému s cílem identifikovat komponenty systému a jejich vzájemné vazby a / nebo vytvořit reprezentaci systému v jiné formě” (Sochor, 1996).

Využívá se především v případě, kdy není dostupná dokumentace nebo nikdy nebyla. Reverzní inženýrství je v podstatě obrácený postup činnosti vývoje. Je to proces, při kterém zpětně analyzujeme, jak byl počítačový program napsán. Cílem je zjistit co nejvíce o problému, který neznáme, nic o něm nevíme, zanalyzovat či zdokumentovat. Důvodem může být rozšíření, oprava chyby či zlepšení výkonnosti.

2.5.3 Návrh v UML

Jazyk UML je univerzální jazyk pro vizuální modelování systémů. Diagramy, které jsou vytvořené v jazyku UML se mohou snadno implementovat a přesto jsou srozumitelné pro lidi (Arlow & Neustadt, 2003).

V tomto jazyce používáme pojem **pohled**. Nad jedním systémem můžeme vytvářet několik druhů pohledů a přitom každý může být z jiného úhlu. UML nabízí několik diagramů (Obrázek 6), které rozdělujeme dle dvou aspektů. První druh diagramů se týká struktury, kterou zobrazují statické modely a druhý z nich řeší chování a je zobrazován v dynamických modelech (Křena & Kočí, 2006).



Obrázek 6: Rozdělení UML diagramů (Křena & Kočí, 2006)

Cílem statických modelů je ukázat jaké objekty jsou pro modelování daného systému důležité a vazby mezi nimi.

Hlavní metou dynamických modelů je popsat chování životního cyklu těchto zmiňovaných objektů a jakým způsobem spolu spolupracují, aby dosáhly požadované funkce navrhovaného systému (Arlow & Neustadt, 2003).

Diagram tříd

Mezi nejpoužívanější UML diagram popisující strukturu patří „class diagram“. Diagramy tříd nám umožňují si představit statický obsah a vztahy mezi třídami. Můžeme v nich ukázat proměnné a metody. Je zde také znázorněna dědičnost, tzn. že zde vidíme, která třída ze které dědí. Stručně řečeno, můžeme v něm popsat všechny závislosti mezi třídami.

Tento pohled nám umožňuje mnohem jednodušší vyhodnocení struktury závislosti systému, než kdybychom toto hledali ve zdrojovém kódu. Vidíme, když abstraktní třídy jsou závislé na konkrétních třídách a můžeme lépe stanovit strategii, jak to napravit (Martin & Martin, 2006).

Diagram případu užití (use case diagram)

Diagram případu užití shrnuje, kteří uživatelé používají aplikaci nebo systém a co s ním mohou dělat. Neukazuje jeho detailní použití v jednotlivých případech, ale pouze stručně popisuje interakce mezi jednotlivými aktéry a případy užití. Tento diagram neukazuje krok po kroku, jaké funkce se musí vykonat k dosažení cílů jednotlivých případů užití. Detaily můžeme popisovat v jiných diagramech a dokumentech. K tomuto účelu slouží především detaily případů užití (MSDN - the microsoft developer network, 2016b).

3. Metodika

V praktické části se nejdříve zabývám tvorbou dokumentace a analýzou implementace funkční aplikace. Jako praktický příklad jsem si zvolila vlastní aplikaci - síťovou verzi deskové hry Pexeso, na jejíž realizaci chci demonstrovat míru zohlednění vybraných principů návrhu softwaru. Na daném příkladě dále modelovat změny tak, abych míru uplatnění těchto principů zvýšila. Dokumentace existujícího řešení s analýzou návrhu je nezbytný předpoklad pro to, abych se mohla při zhodnocení zaměřit nejen na návrh aplikace, ale abych mohla z konkrétní implementace vyvozovat přesnější závěry. Součástí mé metody, jak uvést příklady dobré praxe, je dále modelování změn s ohledem na lepší míru využitelnosti kódu i míru shody s návrhovými principy.

Metodu jsem si stanovila následovně:

1) Vybrala jsem si **funkční aplikaci** pro desktop, přiměřeného rozsahu, realizovanou v objektově orientovaném jazyce.
2) Vytvořila jsem chybějící dokumentaci a provedla **analýzu řešení** a **návrhu architektury**.
3) Použila jsem **metriky** kódu nabízející vývojové prostředí a **zhodnotila** je.
4) Z dostupných výsledků jsem provedla **vyhodnocení dodržování vybraných principů** návrhu softwaru.
5) Na základě zjištěných výsledků jsem navrhla a implementovala **redesign** architektury aplikace.

4. Praktická část

4.1 Vývojové prostředí a vybraná aplikace

Z rodiny nástrojů pro .NET jsem si vybrala programovací jazyk C#. Jako demonstrační příklad jsem si vybrala svoji aplikaci síťové hry Pexeso. Skutečnost, že součástí nebyla ani žádná dokumentace, ba ani komentáře ve zdrojovém kódu, a bylo ji třeba dodělat, již značilo, že se nebude moc jednat o příklad dobré praxe.

Pro analýzu a vývoj aplikace bylo použito Microsoft Visual Studio Community 2015 (verze 14), které je bezplatné pro vývojáře aplikací. Jedná se o vývojové prostředí pro vytváření aplikací pro Windows, IOS, Android, webové aplikace i cloudové služby. Nabízí možnost pro programování, ladění či testování v mnoha jazycích jako je např. C#, C++, Visual Basic či Python. Možnost analýzy modelu v této komunitní edici je poměrně omezená. Prakticky jsem použila jen funkci pro generování diagramu tříd. Tyto diagramy tříd původního a nového návrhu uvádím v příloze na konci této práce (Příloha A - F). Nicméně nezobrazují vazby mezi třídami. Proto jsem pro detailnější analýzu modelu použila modelovací nástroj firmy Sybase, a to PowerDesigner (verze 16, trial).

V úvodu praktické části blíže seznamuji čtenáře s původním řešením zejména pomocí UML diagramů a vytvořené dokumentace. Následuje detailnější analýza návrhu spolu s hodnocením za pomoci metrik kódu a hodnocení dle vybraných principů návrhu softwaru. To byl důležitý předpoklad pro návrh požadovaných změn, redesign architektury a implementaci nové verze aplikace, čemuž je věnována závěrečná část.

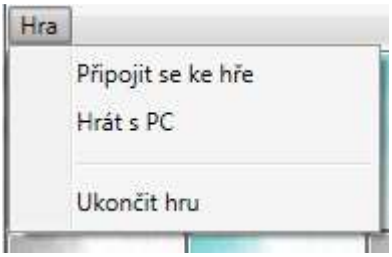
Zdrojové kódy původní a nové aplikace, spolu s modelem vytvořeným v PowerDesigner modeleru, jsou součástí CD přílohy bakalářské práce.

4.2 Dokumentace a analýza návrhu původní aplikace

4.2.1 Uživatelský popis karetní hry Pexeso

Následující popis uživatelských vlastností je společný pro původní i novou verzi aplikace. Jednotlivé obrázky jsou pro obě verze shodné.

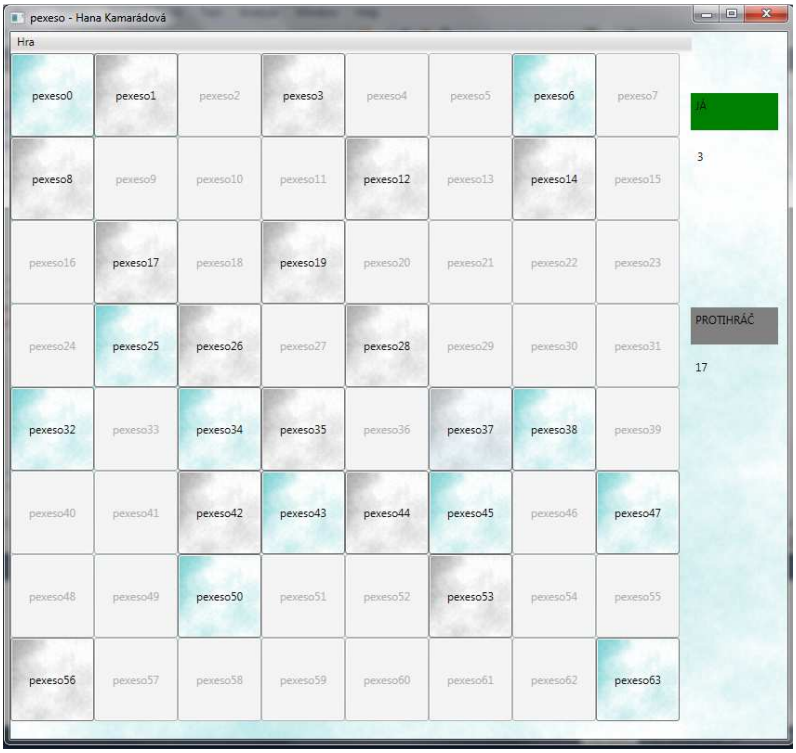
Na následujícím obrázku (Obrázek 7) jsou vidět základní uživatelské funkce klientské části.



Obrázek 7: Menu aplikace se základními uživatelskými funkcemi (aplikace Pexeso, klientská část)

Pexeso se skládá ze 64 kartiček, na kterých je 32 různých obrázků. Hra je určena právě pro dva hráče. Je možné hrát buď hráč proti počítači, nebo dva hráči proti sobě. Hru spouštíme vybráním způsobu hry a to buď klikem na “Připojit se ke hře” nebo “Hrát s PC”. Pokud se chceme připojit ke hře, je nutné, aby byl nejdříve spuštěn server, který čeká na připojení právě dvou klientských aplikací. Každý hráč otáčí vždy dvě kartičky s cílem otočit dvě se stejným obrázkem. Pokud se obrázky neshodují, kartičky se otočí zpět, a hraje protihráč. Pokud hráč najde dvě stejné, kartičky zůstanou otočeny obrázkem nahoru, přičte se mu bod a hráč pokračuje otočením dalších dvou kartiček. Otočené kartičky se stejným obrázkem, z hry dále mizí (zšednou a nejde na ně kliknout). Cílem hry je získat nejvíce bodů.

Na následujícím obrázku (Obrázek 8) je příklad uživatelského rozhraní rozehrané hry.



Obrázek 8: Uživatelské rozhraní rozehrané hry (aplikace Pexeso, klientská část)

4.2.2 Architektura aplikace

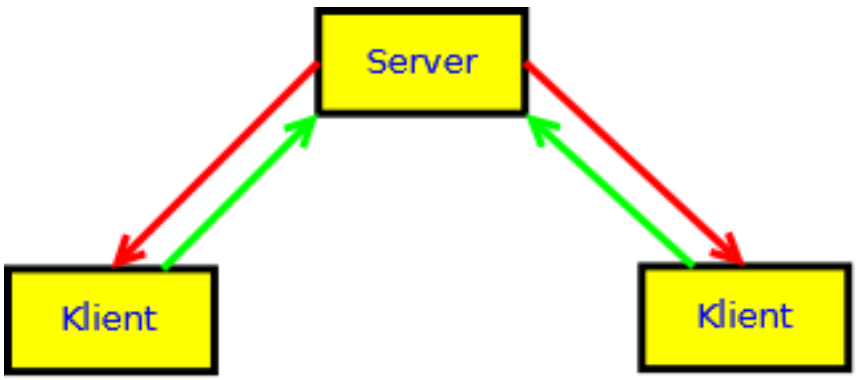
Klientskou část aplikace představuje grafické uživatelské rozhraní (GUI). Toto GUI je hlavní částí aplikace. Přes toto rozhraní uživatel za pomoci myši aplikaci ovládá. Volí, jaká hra má být započata, i provádí samotné tahy. Takovéto akce (kliknutí) vyvolávají volání jednotlivých událostí, které má aplikace nadefinována ke sledování (např. klik v menu, klik na kartičku, a podobně). V případě, že je spuštěna hra s počítačem, hra ke svému chodu nepotřebuje síťovou část, kterou představuje server. Ten je nezbytný k tomu, aby bylo možné započít síťovou komunikaci mezi klienty připojenými ke hře, a jednak pomocí konzole informuje správce serveru o potřebných událostech. Pro síťovou komunikaci serveru s klienty je použit framework WCF.

WCF je definováno kolekcí koncových bodů, kde každý z nich má tyto tři vlastnosti:

- Spojení (způsob jakým probíhá komunikace bodu s okolím) - pro ni byly použity síťové pojmenované roury
- Adresa koncového bodu (kde se nachází brána, která bude komunikovat se službou) - v pexesu byla na serveru i klientech použita síťová pojmenovaná roura „net.pipe://localhost/PipePexeso“
- Popis co služba přenáší - zde byla zvolena jednosměrná (one-way) komunikace

Po definici všech potřebných koncových bodů služba na nich naslouchá, zda některý z klientů nevyžaduje komunikaci. V kladném případě je na serveru vyvolána událost, která odpovídá události zavolané z klienta pro daný kanál. Poté server informaci nějakým způsobem vyhodnotí, zpracuje, a vyvolá další činnost, např. přeposlání tahu jinému klientovi, přihlášení ke hře, apod.

Na následujícím obrázku (Obrázek 9) ilustruji základní schéma komunikace mezi klientem a serverem aplikace Pexeso.



Obrázek 9: Základní schéma komunikace klient - server

4.2.3 Slovní popis tříd, metod a knihoven

Klientská část aplikace

Class MainWindow

Jedna velká třída, jež zajišťuje v podstatě veškerou funkčnost celé aplikace. Dle návrhového vzoru MVC je typu „kontroler“. Obsahuje jak funkční logiku, tak metody pro GUI aplikace.

`MainWindow()` - konstruktor třídy okna aplikace, deaktivuje tlačítka pexesa, nastavuje časovače aplikace, inicializuje hru voláním *InitGame()*.

`void Button_Click(object sender, RoutedEventArgs e)` - obsluha události po kliknutí na hrací políčko Pexesa. Otáčí kartičky (*hrejPexeso()*), a v případě síťové hry také zasílá tah protihráči.

`void HrajePocitac()` - metoda obsluhuje část, která představuje tahy počítače.

`void hratSPc_Click(object sender, RoutedEventArgs e)` - metoda pro obsluhu kliknutí na položku menu „Hrát s PC“. Provede inicializaci hry na logické i vizuální úrovni - *InitGame()* a *nastavMujTah()*.

`void hrejPexeso(string nazevTlacitka)` - metoda otáčí kartičky, na které bylo kliknuto, a zároveň vyhodnocuje shodu obrázků.

`void InitGame()` - metoda zavolá metody pro logickou inicializaci hracího pole, zamíchání karet, deaktivaci tlačítek hracího pole.

`void KdyzJsouTlacitkaJine(object source, EventArgs e)` - metoda řídí chování aplikace pro případ, že otočené kartičky nejsou stejné.

`void KdyzJsouTlacitkaStejne(object source, EventArgs e)` - metoda řídí chování aplikace pro případ, že otočené kartičky jsou stejné. Vyhodnocuje také, zda je ještě možné provést další tah a případně i to, kdo je vítězem hry.

`void nastavMujTah()` - metoda nastaví prostředí hracího pole tak, aby hráč věděl, že nehraje on, ale protihráč / PC.

`void nastavProtihraceTah()` - metoda nastaví prostředí hracího pole tak, aby hráč věděl, že nehraje on, ale protihráč / PC

void ObsluhaUdalostiPrijetiHracihoPole(List<string> hraciPole, List<int> odpovidajiciSiKarticky) - metoda používá v síťové hře pro obsluhu události přijetí hracího pole ze serveru od hráče, který hru inicioval.

void ObsluhaUdalostiStartHry() - metoda používaná v síťové hře pro obsluhu události zaslané serverem, a informuje o tom, že byli přihlášeni oba hráči a hru je možno zahájit.

void povolTlacitka() - metoda provede zpřístupnění tlačítek/kartiček uživateli pro kliknutí

void propojitKeHre_Click(object sender, RoutedEventArgs e) - metoda pro obsluhu kliknutí na položku menu „Připojit se ke hře“. Tímto hráč buď iniciuje hru novou, nebo se připojí ke stávající.

void SeberObrazek() - metoda načte z předdefinované složky na disku seznam obrázků pro kartičky Pexesa

void ukončitHru_Click(object sender, RoutedEventArgs e) - metoda pro obsluhu kliknutí na položku menu „Ukončit hru“.

void zakazTlacitka() - metoda provede znepřístupnění tlačítek/kartiček uživateli pro kliknutí

void ZamichejKarty() - metoda provede zamíchání seznamu karet

Class client

Instanciovaná třída klient organizuje komunikaci se serverem. Objekt je inicializován na základě uživatelského požadavku „připojit se ke hře“ pro vytvoření síťové hry. Datová komunikace je řízena událostmi.

~client() - destruktor třídy.

client(DuplexServiceCallback tr) - konstruktor třídy s parametrem síťového spojení.

OdeslaniKarticek(List<string> seznamKdeJsouObrazky, List<int> zdaJsouStejne) - metoda provede zaslání hracího pole na server, odkud je pak zasláno druhému hráči.

PosliTah(int a) - metoda provede zaslání tahu hráče na server, odkud je pak zaslán druhému hráči.

`PrihlasSeKeHre(string JmenoHrace)` - inicializace/přihlášení hráče do síťové hry.

Interface IDuplexServiceCallback

Interface definuje metody, které bude přijímat klient

`void HraciPole(List<string> hraciPole, List<int> odpovidajiciSiKarticky)`

`void ObaPripraveniKeHre()`

`void SendMessageForClient(string message)`

`void TahProKlienta(int a)`

Class DuplexServiceCallback

Třída implementující rozhraní *IDuplexServiceCallback*. Obsahuje také definice událostí (*StartHry(), TahOdProtihrace(), UdalostHraciPole()*, které jsou volány z některých metod pro vyvolání veřejných událostí jmenného prostoru Pexeso: *UdalostZacatekHry*, *UdalostPrijetiHracihoPole* a *UdalostPrijetiTahuOdProtihrace*).

`void HraciPole(List<string> hraciPole, List<int> odpovidajiciSiKarticky)` - metoda ze serveru přijme hrací pole od protihráče, a přes událost *UdalostHraciPole()* vyvolá veřejnou událost *UdalostPrijetiHracihoPole().*

`void ObaPripraveniKeHre()` - metoda vyhodnotí, zda jsou oba hráči přihlášeni ke hře a pokud ano, vyvolá přes událost *StartHry()* veřejnou událost *UdalostZacatekHry().*

`void SendMessageForClient(string message)` - metoda přijme ze serveru zprávu a zobrazí ji v konzoli

`void TahProKlienta(int a)` - metoda přijme ze serveru tah od protihráče a vyvolá pomocí události *TahOdProtihrace()* veřejnou událost *UdalostPrijetiTahuOdProtihrace().*

Interface IDuplexService

Interface definuje metody, které bude přijímat server.

`void OdeslaniHracihoPole(int id, List<string> hraciPole, List<int> odpovidajiciSiKarticky)`

`void OdeslaniTahu(int id, int a)`

`string Prihlaseni(string message)`

Delegate UdalostPrijetiHracihoPole

Třídy delegátů představují události pro komunikaci po síti.

hraciPole:List<string>

odpovidajiciSiKarticky:List<int>

Delegate UdalostPrijetiTahuOdProtihrace

string tlacitko

Delegate UdalostZacatekHry

Serverová část aplikace

Tato část aplikace slouží ke komunikaci klientů (2 hráčů) pro komunikaci po síti:

Class DuplexService

Třída implementuje rozhraní IDuplexService().

void FunkceCasovace(object hbghb) - metoda slouží k informování obou hráčů o vzájemné připravenosti ke hře.

void OdeslaniHracihoPole(int d, List<string> hraciPole, List<int> odpovidajiciSiKarticky) - metoda slouží k přijetí hracího pole od jednoho z hráčů/klientů a k následnému poslání tomu druhému.

void OdeslaniTahu(int id, int tah) - metoda slouží k přijetí tahu od jednoho z hráčů/klientů a k následnému poslání tomu druhému.

void Prihlaseni(string message) - metoda slouží k přijímání informací o pokusu přihlášení hráčů ke hře, následnému rozhodnutí, zda lze hráče do hry vpustit a zda může být hra započata.

Class Program

void Main(string[] args)

Interface IDuplexService

Interface definuje metody, které bude přijímat server.

void OdeslanihracihoPole(int id, List<string> hraciPole, List<int> odpovidajiciSiKarticky)

void OdeslaniTahu(int id, int a)

string Prihlaseni(string message)

Interface IDuplexServiceCallback

Interface definuje metody, které bude přijímat klient.

```
void HraciPole(List<string> hraciPole, List<int>
odpovidajiciSiKarticky)

void ObaPripraveniKeHre()

void TahProKlienta(int a)

void ZpravaProKlienta(string message)
```

Použité knihovny

System - Obsahuje základní třídy, které definují běžně používané hodnoty a datové typy, události, obslužné rutiny událostí, rozhraní, atributy a zpracování výjimek. Umožňuje práci s poli, řetězci nebo konzolí.

System.Collections.Generic - Obsahuje třídy kolekcí a rozhraní umožňující definovat generické kolekce, které dovolí uživatelům vytvořit silně typové kolekce, jež poskytují lepší bezpečnost než negenerické kolekce.

System.Linq - Tento jmenný prostor poskytuje třídy a rozhraní podporující dotazování pomocí LINQ (Language Integrated Query).

System.ServiceModel - Základní jmenný prostor pro Windows Communication Foundation (WCF). Obsahuje typy, které jsou v rámci této technologie potřebné k vývoji služeb a klientských aplikací.

System.Threading - Obsahuje typy umožňující tvorbu aplikací využívajících multithreading.

System.Windows - Obsahuje několik základních důležitých prvků používaných v aplikacích založených na rozhraní WPF, včetně animačních klientů, ovládacích prvků uživatelského rozhraní, datových vazeb a převodů mezi typy.

System.Windows.Controls - Obsahuje třídy k tvorbě ovládacích prvků, které umožňují uživateli ovládat aplikaci. Třídy ovládacích prvků jsou základem pro práci s libovolnou aplikací, protože uživateli umožní zobrazit, zadat či vybrat údaje či jiné informace (reprezentuje bázové třídy pro rozhraní uživatele tak, že používá ControlTemplate pro definování jeho vzhledu).

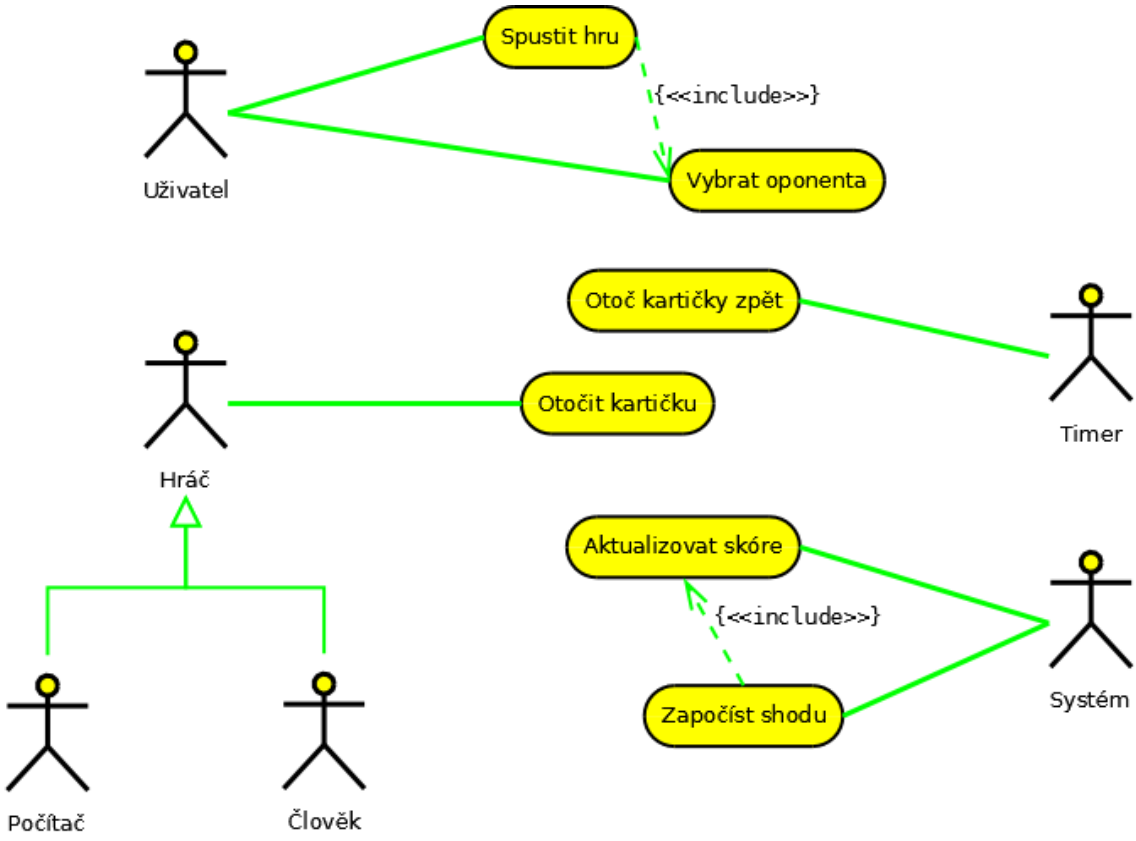
System.Windows.Media - Obsahuje typy, které podporují integraci multimédií včetně nákresů, textů, zvuku a videa ve WPF aplikacích.

System.Windows.Media.Imaging - Obsahuje typy, které jsou používány k zakódování a dekódování bitmapových obrazů.

System.Windows.Threading - Podpora vláken pro Windows Presentation Foundation (WPF).

4.2.4 Diagram případu užití

Základní interakci uživatele s aplikací popisuje následující diagram případů užití (Obrázek 10).



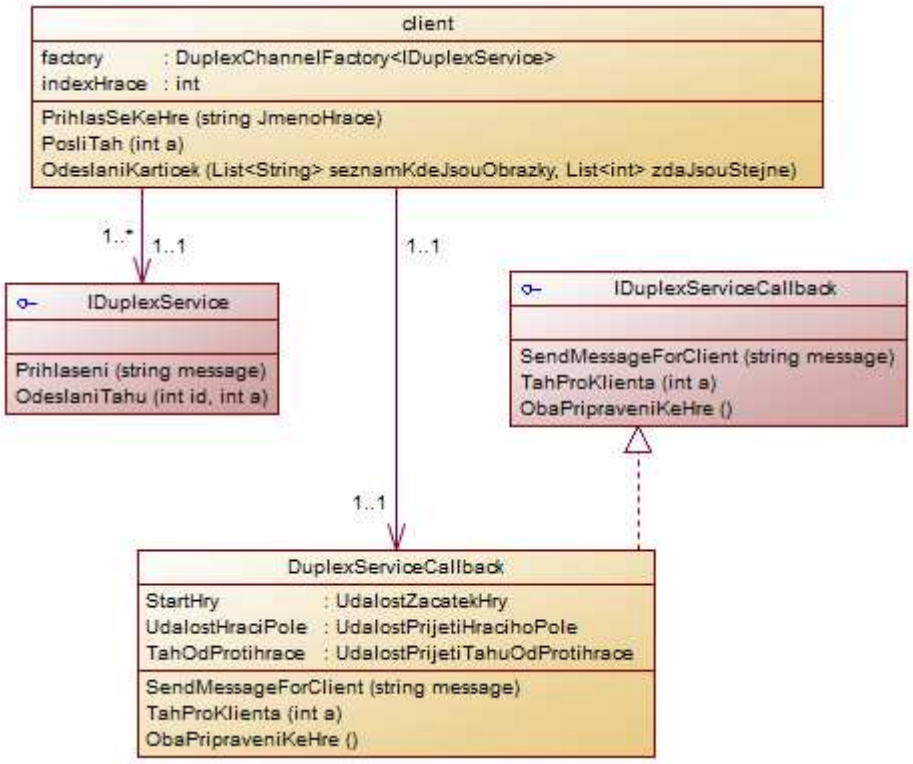
Obrázek 10: Diagram případu užití (původní verze Pexesa)

4.2.5 Diagram tříd

V následujících diagramech je představena architektura původního řešení. Hlavní funkce jsou implementovány ve třídě MainWindow (Obrázek 11), která tak zajišťuje obsluhu grafického rozhraní a správu potřebných dat pro chod aplikace.

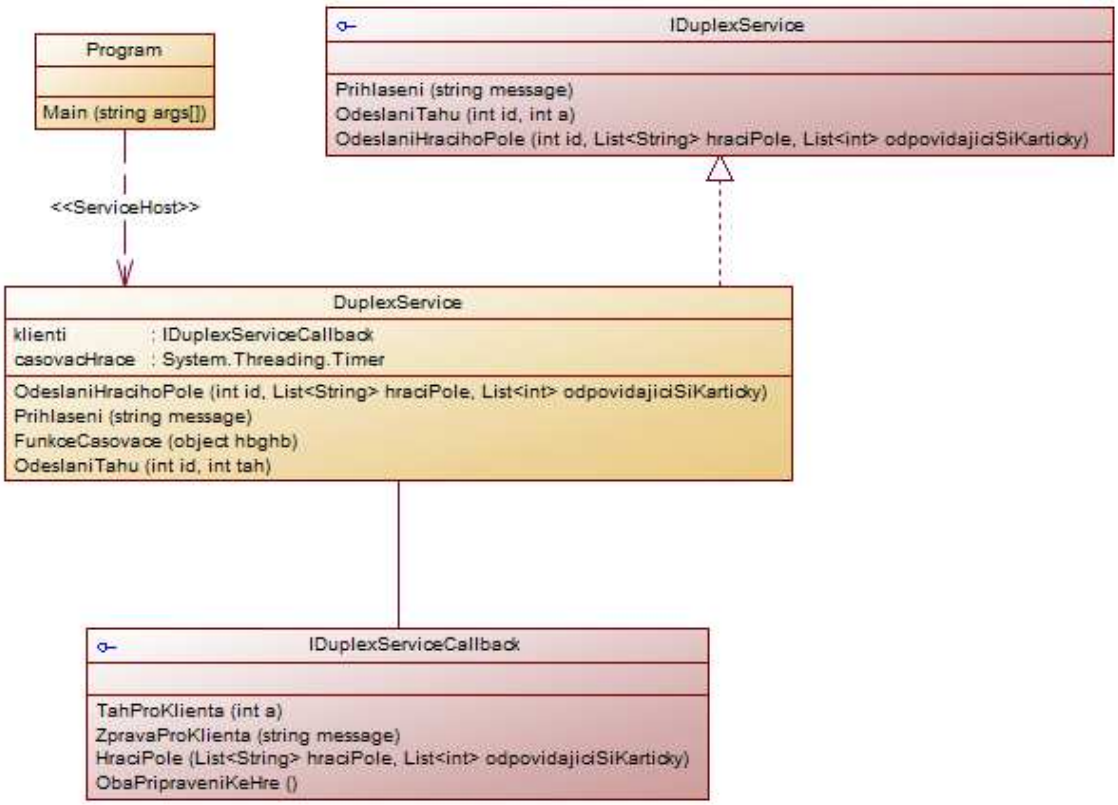
Obrázek 11: Diagram tříd původního řešení klientské části hry Pexeso, hlavní třídy

Třídy zajišťující komunikaci na straně klienta jsou představeny v následujícím diagramu (Obrázek 12).



Obrázek 12: Diagram tříd původního řešení klientské části hry Pexeso, detail tříd zajišťující síťovou komunikaci

Architektura serverové části aplikace je představena na diagramu následujícím (Obrázek 13). I přes shodu názvů některých tříd s klientskou částí se jedná o jejich různé implementace.



Obrázek 13: Diagram tříd původního řešení síťové komunikace na straně serveru

4.3 Výpočet metriky kódu a hodnocení výsledků

Aktuální verze Visual Studia nám nabízí možnost testování aplikace z hlediska indexu udržovatelnosti, cyklomatické složitosti, hloubky dědičnosti, provázanosti tříd a přibližného počtu řádků kódu (viz. kapitola 2.5.1).

Nad původní aplikací Pexeso jsem spustila “calculate code metrics for pexeso” s následujícími výsledky.

Maintainability index spočítá hodnoty v rozmezí 0 - 100, které představují relativní jednoduchost udržitelnosti kódu. Čím je hodnota vyšší, tím by měl být kód snadněji udržovatelný.

Na základě vyhodnocení jsem zjistila, že moje třídy dosahují indexu udržovatelnosti mezi hodnotou 56 - 100 se zeleným indikátorem. Čím vyššího indexu dosahují, tím by měl být kód lépe udržitelný. Dle manuálu Visual Studia jsou však všechny hodnoty optimální. Potenciální riziko hrozí až u žlutého indikátoru, který značí hodnoty mezi 10 a 19. Nízkou udržovatelnost indikuje hodnota mezi 0 a 9 s červeným čtverečkem. Nejhorší hodnoty této metriky dosahovala třída MainWindow, průměrně 56. Nejhorší

metodou dané třídy je *void ZamichejKarty()* s hodnotou 40. I tato nejnižší hodnota však ještě svítí zeleně.

Nejvyšší cyklomatickou složitost 11 má metoda *hrejPexeso()* třídy MainWindow, která obsadila i nejvyšší hodnotu 19 u nevhodného jevu hloubce dědičnosti

Metrika Lines of Code říká, že celá aplikace má přibližně 393 řádků kódu. Z toho 255 řádků zabírá opět hlavní třída MainWindow.

Z daných výsledků lze vyvodit, že nejhůře napsanou třídou je MainWindow, ve které může při potřebné změně nebo rozšíření dojít snáze k neočekávanému chování.

Na následujícím obrázku (Obrázek 14) uvádím příklad měření původní verze aplikace Pexeso, klientské části. Další výstupy měření jsou součástí přílohy (Příloha G - I).

Hierarchy	Maintainability Index	Cyclomatic Complexity	Class Coupling	Lines of Code
pexeso (Debug)	88	131	59	343
pexeso	88	131	59	343
MainWindow	56	72	44	255
client	62	19	12	49
DuplexServiceCallback	78	20	8	38
IDuplexService	100	3	3	0
IDuplexServiceCallback	100	4	2	0
UdalostZacatekHry	100	4	2	0
UdalostPrijetiHracihoPole	100	4	3	0
UdalostPrijetiTahuOdProtihrace	100	4	2	0
App	100	1	1	1

Obrázek 14: Výsledky metriky kódu původní verze aplikace Pexeso (Visual Studio 2015)

4.4 Hodnocení dle vybraných principů návrhu softwaru

O třídě MainWindow můžeme prohlásit, že nesplňuje mnoho příkladů dobré praxe. Třída obsahuje směsici metod i proměnných, které se neorganizovaně a nestrukturovaně váží víceméně současně ke všem částem aplikace najednou - ke vzhledu, k funkčnosti i k datové složce.

Metody samotné nesplňují pravidlo jedné zodpovědnosti. Například metoda *KdyzJsouTlacitkaStejne()*, která je volána pokud dvě otočené kartičky byly stejné, provádí najednou toto vše:

- nastavuje GUI (povoluje tlačítka)
- nastavuje datovou stránku hry (co už bylo uhádnuto)
- kontroluje, zda hra už neskončila (pak tedy vyhodnocuje, kdo vyhrál), apod.

Metody a proměnné nebývají pojmenovány vždy úplně srozumitelně podle toho, co mají dělat nebo dělají: *SeberObrazek()*, *Button_Click()*, *img*, atd. Dále nejsou proměnné a metody pojmenovány konzistentně, tj. liší se způsob jejich pojmenování z pohledu velikosti písmen na počátku jména: např. *zakazTlacitka()*, *ZamichejKarty()*. Oba z uvedených příkladů nedodržení zásad použití vhodných jmenných konvencí snižují schopnost vývojáře dobře se orientovat nejen v objektech samotných, ale i v tom, co dané metody provádějí a k čemu jsou určeny.

Bývá porušován princip DRY, opakují se části, které by mohly být nebo dokonce již jsou zapouzdřeny do funkcí (např. v konstruktoru třídy MainWindow se opakuje kód, který je totožný s obsahem funkce *zakazTlacitka()*, podobně se lze dívat i na obsah metod *zakazTlacitka()* a *povolTlacitka()*, jejichž obsah by stačilo shrnout do jedné metody s parametrem, který určuje povolení či zakázání).

Nejen s principem DRY, ale i s dalšími je těsně provázán KISS, který poukazuje na to, že všechny proměnné, metody či třídy by měly být nejjednodušší, jak jen je to možné. Jak již jsem zmínila výše, metody *zakazTlacitka()* a *povolTlacitka()* byly napsány tak, aby jejich implementace byla co nejlehčí. Ovšem podstata KISS tkví v tom, že je to co nejvíce jednoduché pro běh aplikace, rozšíření či udržovatelnost, proto zmíněné metody i mnohé další nesplňují ani zdaleka princip KISS.

4.4.1 Hodnocení třídy klient

Třída client definuje metody pro síťovou komunikaci hry se serverem. Tato komunikace využívá framework WCF. Třída ve svém konstruktoru vytváří pomocí třídy DuplexChannelFactory (ze jmenného prostoru System.ServiceModel) komunikační kanál, přes který bude:

- posílat zprávy na server (využívá rozhraní IDuplexService)
- přijímat zprávy ze serveru, k čemuž používá instanci třídy DuplexServiceCallback, která implementuje rozhraní IDuplexServiceCallback

V této třídě by měly od sebe být odděleny obecná komunikační část od té, která provádí operace spojené s hrou. Zjednodušení a zpřehlednění kódu by mohlo být provedeno kupříkladu novou funkcí určenou pro vytvoření komunikačního kanálu, v kombinaci s ošetřením, že se tato operace zdařila. Takovýto kód je ve funkcích třídy client použit duplicitně celkem 3x. Vytvořením nové funkce by se nejen zkrátil kód třídy samotné, ale také zlepšila jeho udržovatelnost.

Implementace řízení síťové komunikace je ve třídě client zpřehledněna použitím návrhové vzoru Abstraktní továrna (viz. kapitola 2.3.1). Pro bližší ilustraci uvádím následující část kódu z implementace (Zdrojový kód 1), jak je v konstruktoru třídy uplatněna.

```
factory = new DuplexChannelFactory<IDuplexService>(
        context,
        new NetNamedPipeBinding(),
        new EndpointAddress("net.pipe://localhost/PipePexeso")
);
```

Zdrojový kód 1 - Použití návrhového vzoru Abstraktní továrna v klientské části aplikace původní verze

4.4.2 Hodnocení serverové části

Kód serverové části byl v rámci vývoje pexesa vyvíjen v jiné „solution“ a také oddělen do jiného jmenného prostoru (PexesoServer). Nevýhodou těchto oddělení je to, že kromě horší spravovatelnosti obou celků došlo také k duplicitě definic obou rozhraní (interface IDuplexService a IDuplexServiceCallback).

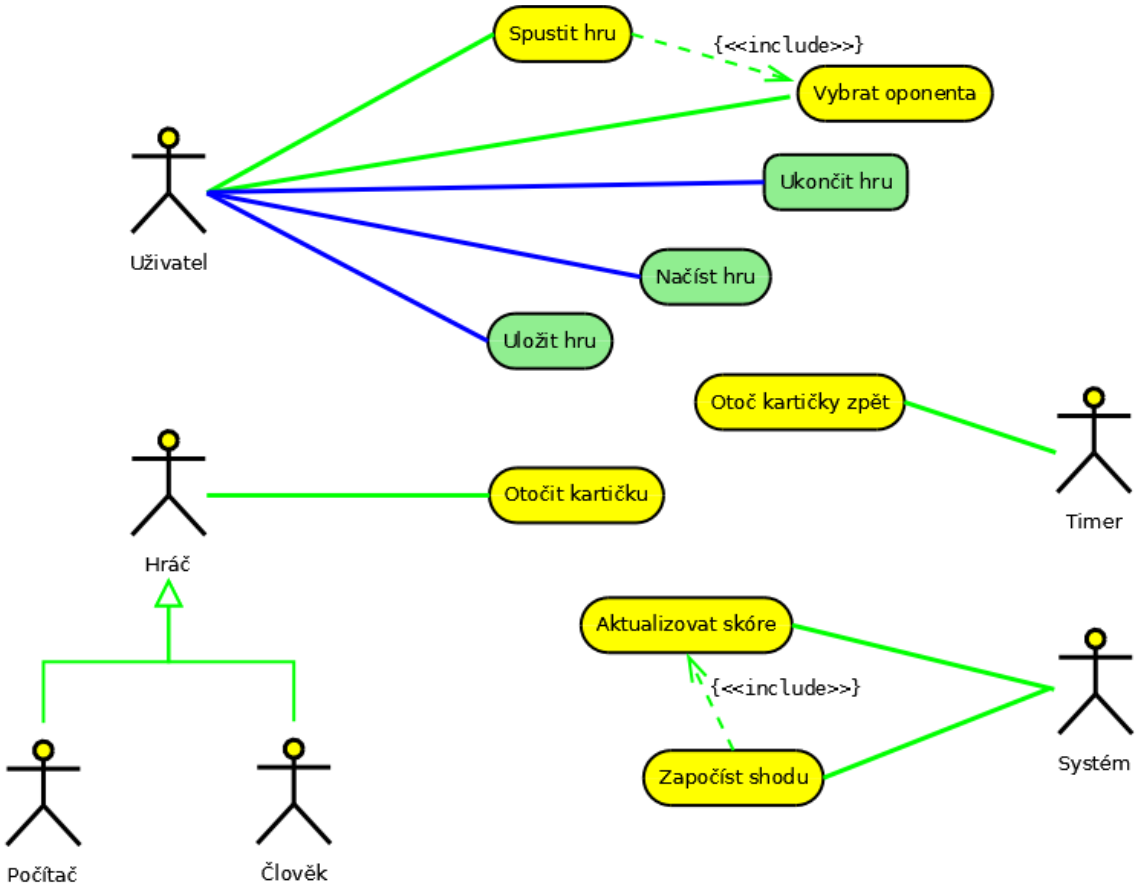
4.5 Redesign architektury a návrh dalších změn

V následujících podkapitolách se blíže věnuji popisu změn vyplývajících ze zhodnocení původní verze. Nejvýznamnější změna se odehrála ve třídě MainWindow s ohledem na požadovanou dekompozici a širší uplatnění návrhového vzoru MVC. Úpravami tak prošel nejen diagram tříd, ale i např. diagram případu užití, kde jsem se pokusila zohlednit možné rozšíření aplikace a tím zvýšit robustnost návrhu.

Při změnách v návrhu a následné implementaci se snažím v co nejširší míře respektovat důsledky vyplývající z hodnocení (viz. kapitola 4.4 Hodnocení dle vybraných principů návrhu softwaru), na které se v průběhu odkazuji.

4.5.1 Diagram případu užití

Optimalizací prošly tedy nejprve případy užití funkcí aplikace (Obrázek 15). Modrou barvou jsou naznačeny vazby, které by bylo možné očekávat s ohledem na případné další rozšíření funkcí aplikace.



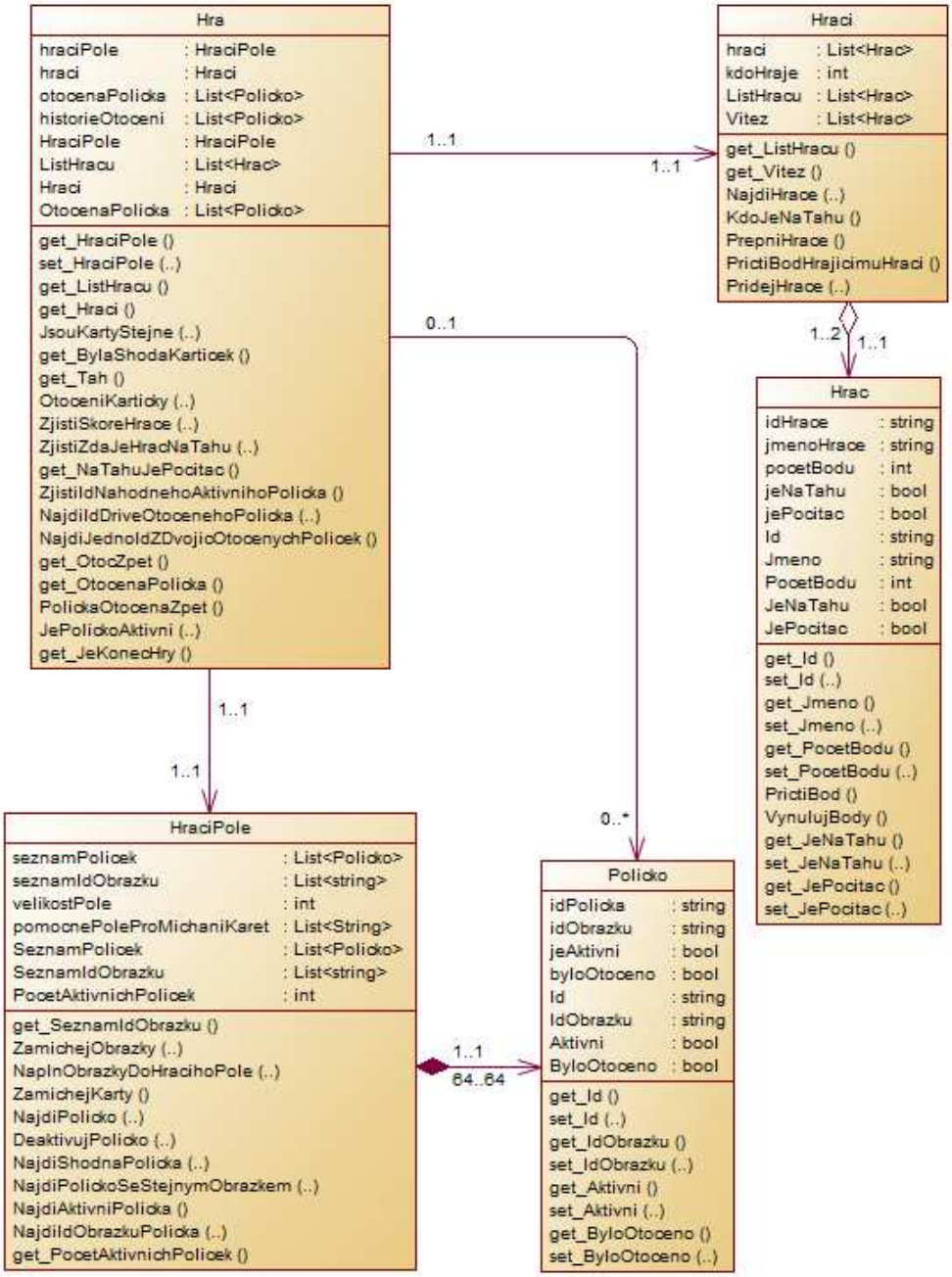
Obrázek 15: Diagram případu užití s navrhovaným rozšířením (aplikace Pexeso)

Na modelovaném příkladě se jedná o funkce, které umožňují hru přerušit a poté uvést do původního stavu. Toto očekávatelné rozšíření funkčních požadavků a jejich implementace by měla být usnadněna při uplatnění MVC, jak si ukážeme v následující kapitole.

4.5.2 Architektura dle MVC

Zásadní změnou, která byla provedena, bylo oddělení datové, nebo-li modelové části (z pohledu MVC) do samostatných tříd (Model). Tyto třídy obsahují informace o hře, hráčích, hracím poli, atd. Oddělením této datové části je umožněno tuto část aplikace např. nejen opakovaně použít v jiné aplikaci s jiným vzhledem (View) či ovládáním (Controllerem), ale zároveň snadněji oddělit data hry pro její uložení a obnovu, jak bylo naznačeno v rozšířeních diagramu případu užití (Obrázek 15).

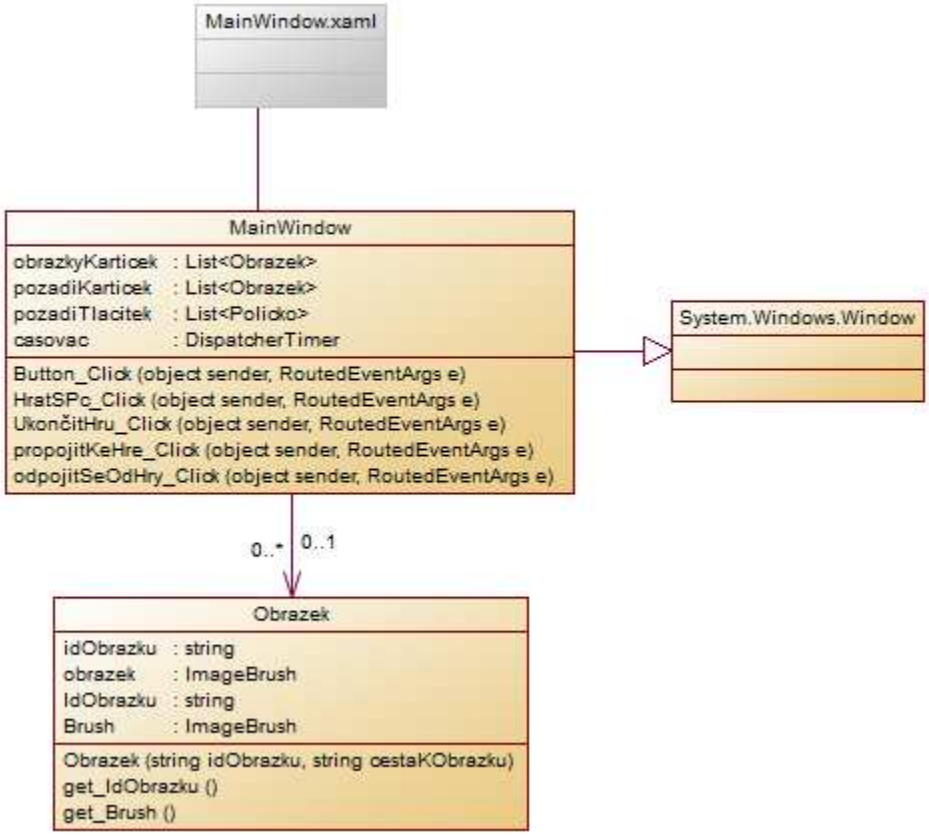
Model redesignované aplikace je představen diagramem tříd na následujícím obrázku (Obrázek 16).



Obrázek 16: Diagram tříd pro klientskou část upravené aplikace Pexeso - Model

U kompozice tříd jsem se snažila dodržet princip DRY (například vhodné opakované využívání metod *NajdiPolicko()*, *NajdiHrace()* namísto opakování kódu).

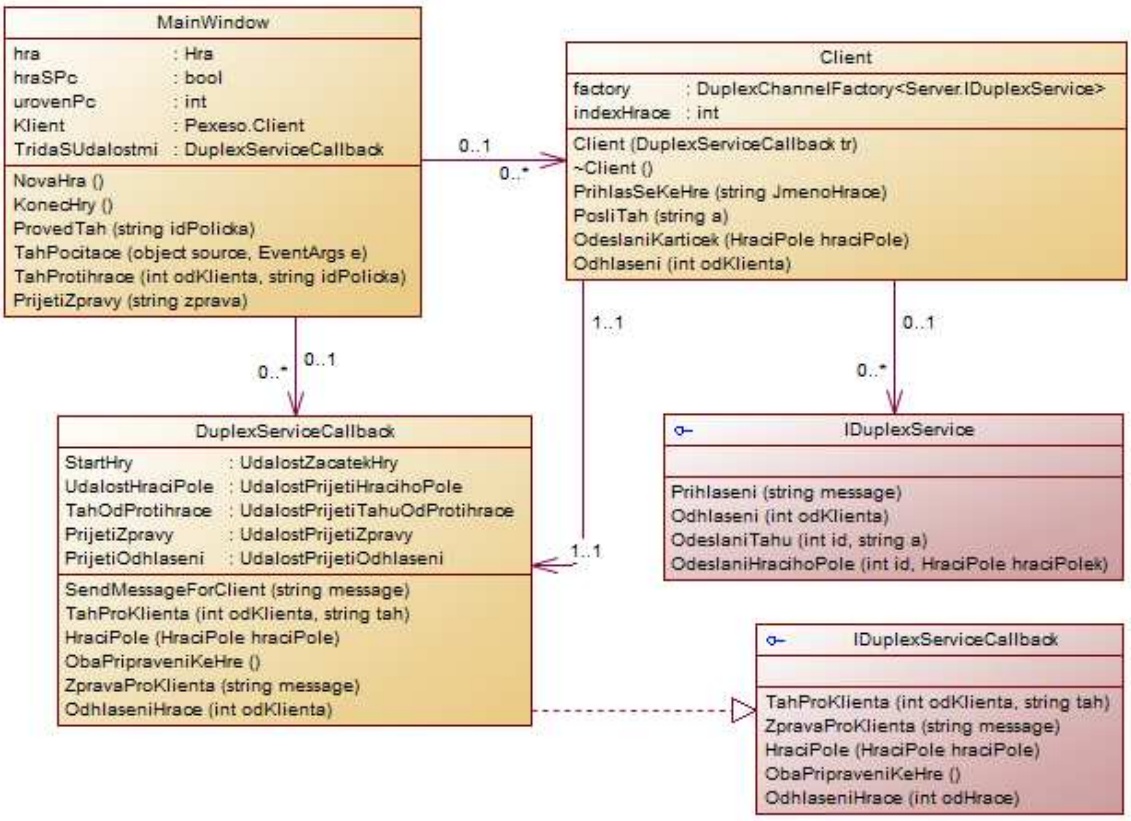
Na následujícím diagramu (Obrázek 17) jsou představeny třídy zastupující část View. Při dekompozici MainWindow byl zohledněn návrhový princip OCP (viz. kapitola 2.2.1 Základní principy SOLID), aby byly metody daných tříd více uzavřené pro změny a více otevřené pro rozšíření.



Obrázek 17: Diagram tříd pro klientskou část upravené aplikace Pexeso - View

Zavedením nové třídy Obrazek se tak snažím o snížení závislosti této datové části View na Controlleru. Nicméně užší propojení řídící třídy s prezentační vrstvou (specifikace uživatelského rozhraní je spoludefinována v souboru MainWindow.xaml) důslednější oddělení dat pro jednotlivé logické části dle MVC znesnadňuje. Ale to je ostatně vlastnost frameworku pro desktopové aplikace.

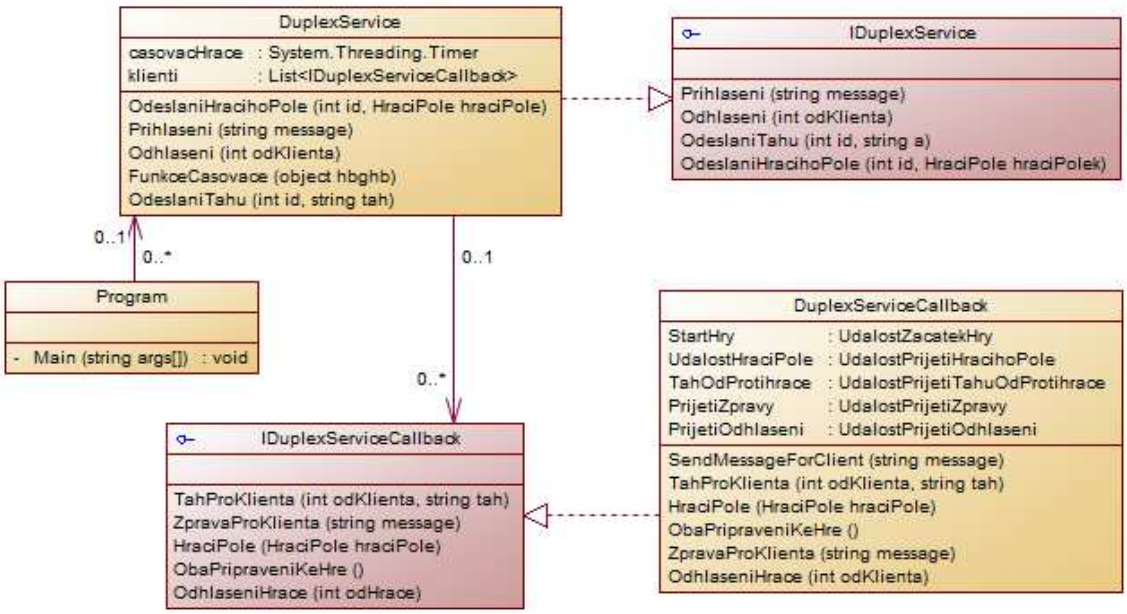
Následující diagram (Obrázek 18) znázorňuje síťové řešení na straně klienta. Třída MainWindow je společná pro část View a Controller. Část metod je využívána pro obsluhu uživatelského rozhraní a ostatní pro řízení aplikace.



Obrázek 18: Diagram tříd pro klientskou část upravené aplikace Pexeso - Controller

Třídy pro řízení síťové komunikace tak logicky spadají do řídící části aplikace, a proto jsou reprezentovány v tomto diagramu.

Na následujícím diagramu serverové části (Obrázek 19) se tyto třídy pro řízení síťové komunikace objevují také, ale na rozdíl od původního řešení se již jedná o stejnou sdílenou část implementace, čímž byla odstraněna duplicita kódu, aby byl respektován zejména princip DRY.



Obrázek 19: Diagram tříd síťové komunikace na straně serveru upravené aplikace Pexeso

4.5.3 Dodržení konvencí

Ke zlepšení kvality kódu, k jeho zpřehlednění a zvýšení udržovatelnosti, přispívá i dodržování doporučených konvencí zápisu.

Ve zdrojovém kódu redesignované verze jsem se, až na výjimky, zaměřila konkrétně na následující:

- názvy tříd, rozhraní, atributů, metod, vlastností, událostí i jmenné prostory používají zápis Pascal case (použití velkého písmena v prvním a každém dalším slově názvu), např. HraciPole, *Hra.NaTahuJePocitac()*, apod.)
- název rozhraní navíc začíná „I“
- v názvech nejsou použita podtržítka
- až na výjimky jsou názvy kratší než 20 znaků
- názvy se snaží dodržet účel (např. *Hrac.PrictiBod()*, Hrac.PocetBodu, Hrac.Jmeno, atd.)
- názvy lokálních proměnných a názvy parametrů metod používají velbloudí notaci (malé písmeno v prvním, a velké v každém dalším slově - idHrace, jmenoHrace)

4.5.4 Další navrhované změny a hodnocení

Dále je uveden výčet možných změn, které by bylo vhodné zvážit při příští implementaci:

1. Delegace tříd UdalostPrijetiHracihoPole, UdalostZacatekHry a UdalostPrijetiTahuOdProtihrace v části klient by měly být součástí nové samostatné třídy SitovaHra.
2. Širší uplatnění principů MVC. Dekompozice MainWindow na část, která se věnuje grafickému uživatelskému prostředí, a část, která řídí aplikaci, aby se zvýšila např. možnost migrace do jiného běhového prostředí.
3. V části PexesoServer rozdělit třídu DuplexService na dvě. DuplexService by měla zajišťovat pouze síťovou komunikaci. V této třídě by zůstala pouze metoda *OdeslaniHracihoPole()* a *OdeslaniTahu()*.
4. Aplikace je závislá na operačním systému. Síťový framework WCF je součástí knihovny *System.ServiceModel* Visual Studia, obdobně jako grafická knihovny *System.Windows*, ze kterých dědí třída MainWindow.
5. K zamyšlení by dále bylo použití dalších návrhových vzorů jako je Příkaz, Abstraktní továrna a Pozorovatel. Vzor chování Příkaz by se mohl použít na otáčení kartiček či na uživatelské rozhraní, vytvářecí vzor Abstraktní továrna na vytváření objektů, jako např. hračů. Pozorovatel, vzor vztahující se k chování by mohl být zodpovědný za měnící se strukturu, změny na pexesu jako třeba aktualizace skóre hry. Za zvážení však stojí, zda by tyhle změny na aplikaci takto malého rozsahu skutečně vedly jak ke zlepšení struktury návrhu, tak ke zkvalitnění funkčnosti aplikace.

Sledovanými změnami v návrhu se mi podařilo implementovat třídy a metody, tak aby splňovaly podmínku jedné zodpovědnosti, a celkově kód zpřehlednit. Uplatněním architektury MVC se mi ve větší míře podařilo oddělit modelová data od zbytku celku. Důsledné oddělení View od Controlleru však implementováno nebylo (zejména s ohledem na body 2. a 4.).

5. Závěr

Cílem mé práce byl rozbor a popis vhodných postupů při vývoji aplikací na platformě .NET a jejich demonstrace na vhodném příkladě.

V teoretické části jsem se zabývala přehledem metodik při vývoji aplikací. Představila jsem jak klasické modely životního cyklu, tak metodiky s agilními přístupy, jejichž stěžejním rozdílem je, že se místo na samotný proces zaměřují na člověka. Dále jsem představila vybrané principy návrhu, zejména některé principy ze SOLID, KISS a DRY, o nichž se domnívám, že jejich dodržování patří mezi základní principy dobré praxe při návrhu a vývoji aplikací. Představila jsem návrhové vzory včetně architektury MVC. Úspěšnému vývoji a realizaci softwaru pomocí objektově orientovaných programovacích jazyků pomáhá dodržování klíčových konceptů návrhu, mezi které patří abstrakce, dekompozice či modularita. V další části je seznámení s metrikami kódu, které můžeme prakticky využít ve Visual Studiu 2015. Dále je představeno základní rozdělení UML diagramů, diagram tříd a diagram případu užití, na kterých jsou četně ukazovány řešené příklady.

V praktické části jsem sledované zásady z teoretické části demonstrovala na vlastních vzorových aplikacích s využitím moderních vývojářských nástrojů (jejichž výstupy jsou přílohou bakalářské práce), čímž jsem se snažila zvýšit užitnou hodnotu práce pro čtenáře se zájmem o sledovanou problematiku. Původní vzorovou aplikaci jsem podrobně zdokumentovala, architekturu aplikace jsem hodnotila z hlediska dodržování sledovaných návrhových principů a uplatnění návrhových vzorů. Dále jsem využila možnosti Visual Studia 2015, které nabízí testování ohledně metrik kódu. Na základě zjištěných výsledků jsem navrhla redesign původního návrhu a ve snaze dodržet příklady dobré praxe, jsem navrhla a implementovala aplikaci novou.

Vzhledem k tomu, že původní aplikace nesplňovala mnohá doporučení, a přesto byly metriky kódu vyhodnoceny jako úspěšné, odhaduji, že dané hodnocení není úplně vhodné pro využití na aplikaci tak malého rozsahu. Což ovšem bude pravděpodobně jinak při vývoji projektu většího rozsahu.

Domnívám se, že se mi splnit cíle mé bakalářské práce podařilo. Na základě pojmů z teoretické části čtenáře prakticky provázím všemi fázemi životního cyklu vývoje softwaru (analýza, návrh, implementace a nasazení). Na modelovaném příkladě problematiku nejprve analyzuji, následně se pokouším navrhnout vhodné změny, které

následně implementuji. Úspěšnost nasazení změn podrobuji bližší analýze a ze zhodnocení ověřuji úspěšnost změn v návrhu. Na základě zdokumentovaní daného procesu tak čtenáři demonstruji vhodné postupy při vývoji.

I přes snahu uplatnit některé postupy dobré praxe se najdou další, nad kterými by bylo vhodné se zamyslet a zakomponovat je. Sledované aspekty totiž není možné vždy úplně aplikovat, ale už jenom snaha o jejich dodržování vede k výraznému zkvalitnění softwarového návrhu a posléze i programového kódu. Umět prakticky správně využívat vhodné zásady a doporučení při vývoji není vždy snadné, i přes jejich zdánlivou jednoduchost. Protože jsem chtěla využít možnosti s danou problematikou blíže teoreticky a prakticky pracovat, byla pro mne volba tématu bakalářské práce jasná.

I Summary and keywords

Summary:

The goal of the bachelor´s thesis is to mention the keystones and best practices of an object-oriented design and development of software. The thesis focuses on selected software design principles, which serve as a guide to those who want to avoid problems caused by a poor design of code. In addition, the thesis explains important nomenclature related to the subject matter.

To give an example of how to use the OOP design patterns and best practices, the thesis also includes one use case - a working C# application developed in the Microsoft Visual Studio 2015 environment.

Keywords:

Software development process, UML, software design principles, software design pattern

II Seznam použitých zdrojů

Arlow, J.; Neustadt, I. (2003). UML a unikovaný proces vývoje aplikací. Brno: Computer Press.

Arlow, J., Neustadt, I. (2007). UML 2 a unifikovaný proces vývoje aplikací. Brno: Computer Press.

Bernard, B. (2009). Prezentační vzory z rodiny MVC. Dostupné z: https://www.zdrojak.cz/clanky/prezentacni-vzory-zrodiny-mvc

Bishopová, J. (2010). C#: návrhové vzory. Brno: Zoner Press.

Čada, O. (2009). Objektové programování: naučte se pravidla objektového myšlení. Praha: Grada Publishing.

De Keyser, W., Springael, J. (2010). Why Don´t We Kiss!?: A Contribution to Close the Gap Between Real-world Decision Makers and Theoretical Decision-model Builders. Brussels: UPA.

Dresler, R. (2011a). Liskovové princip zaměnitelnosti - podmínky kladené na dědičnost. Dostupné z: http://www.robertdresler.cz/2011/02/liskov-substitution-principle.html

Dresler, R. (2011b). Princip jedné zodpovědnosti. Dostupné z: http://www.robertdresler.cz/2011/01/princip-jedne-zodpovednosti.html

Dresler, R. (2011c). Princip oddělení rozhraní - méně je více. Dostupné z: http://www.robertdresler.cz/2011/02/princip-oddeleni-rozhrani-mene-je-vice.html

Gamma, E., Helm, R., Johnson, R., Vlissider, J. (2003). Návrh programů pomocí vzorů: stavební kameny objektově orientovaných programů. Praha: Grada Publishing.

Hall McLean Gary (2014). Adaptive Code via C#: Agile coding with design patterns and SOLID principles. United States of America: Microsoft Press.

Hunt, A., Thomas, D. (2007). Programátor pragmatik: Jak se stát lepším programátorem a vytvářet kvalitní software. Brno: Computer Press.

Kay, R. (2002). Životní cyklus vývoje systému. Dostupné z: http://computerworld.cz/archiv/zivotni-cyklus-vyvoje-systemu-18709

Kanat-Alexander, M. (2012). Code Simplicity: The Fundamentals of Software. United States of America: O'Reilly Media.

Křena, B., Kočí, R. (2006). Úvod do softwarového inženýrství: IUS (Studijní opora). Brno: FIT VUT.

Mádl, V. (2003). Kvalitní metodika je předpokladem úspěchu při řízení projektů. Dostupné z: http://www.systemonline.cz/clanky/kvalitni-metodika.htm

Martin, R. C., Martin, M. (2007). Agile Principles, Patterns, and Practices in C#. United States of America: Prentice Hall.

Martin, R. C. (2009). Čistý kód: Návrhové vzory, redaktorování, testování a další techniky agilního programování. Brno: Computer Press.

Miguel, S. J. (2014). KISS, YAGNI & DRY, 3 Principles to Simplify Your Life as Developer. Dostupné z: http://www.itexico.com/blog/bid/99765/Software-Development-KISS-YAGNI-DRY-3-Principles-to-simplify-your-life

MSDN - the microsoft developer network (2016a). Code Metrics Values. Dostupné z: https://msdn.microsoft.com/en-us/library/bb385914.aspx

MSDN - the microsoft developer network (2016b). UML Use Case Diagrams: Guidelines. Dostupné z: https://msdn.microsoft.com/en-us/library/dd409432.aspx

Levine, L. D., Gill, D. Ch. (2000). Software Design Principles and Guidelines. Dostupné z: http://www.cs.wustl.edu/~levine/courses/cs342/c++/design-principles_4.pdf

Pecinovský, R. (2007). Návrhové vzory. Brno: Computer Press.

Sochor, J. (1996). Údržba softwaru. Zpravodaj ÚVT MU. Roč. VI, č. 3, s. 15-20. Dostupné z: http://webserver.ics.muni.cz/zpravodaj/articles/61.html

van Vliet, H. (2007). Software Engineering: Principles and Practice. Wiley

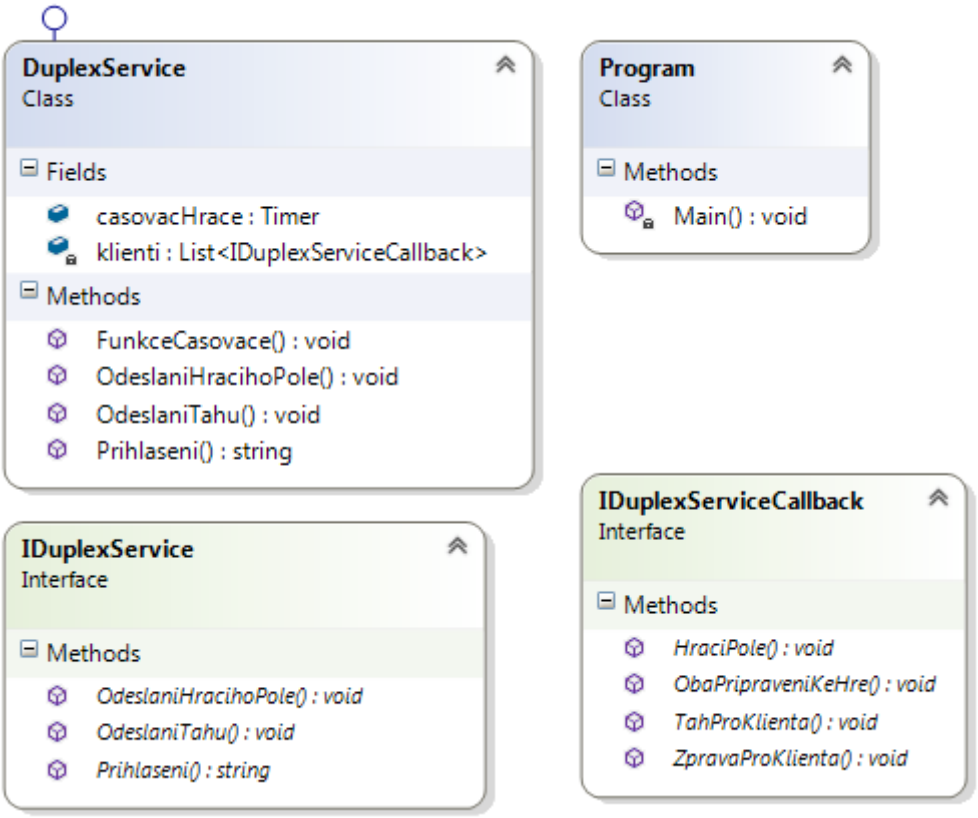
Wells, D. (2013). Extreme Programming: A gentle introduction. Dostupné z: http://www.extremeprogramming.org

III Seznam obrázků

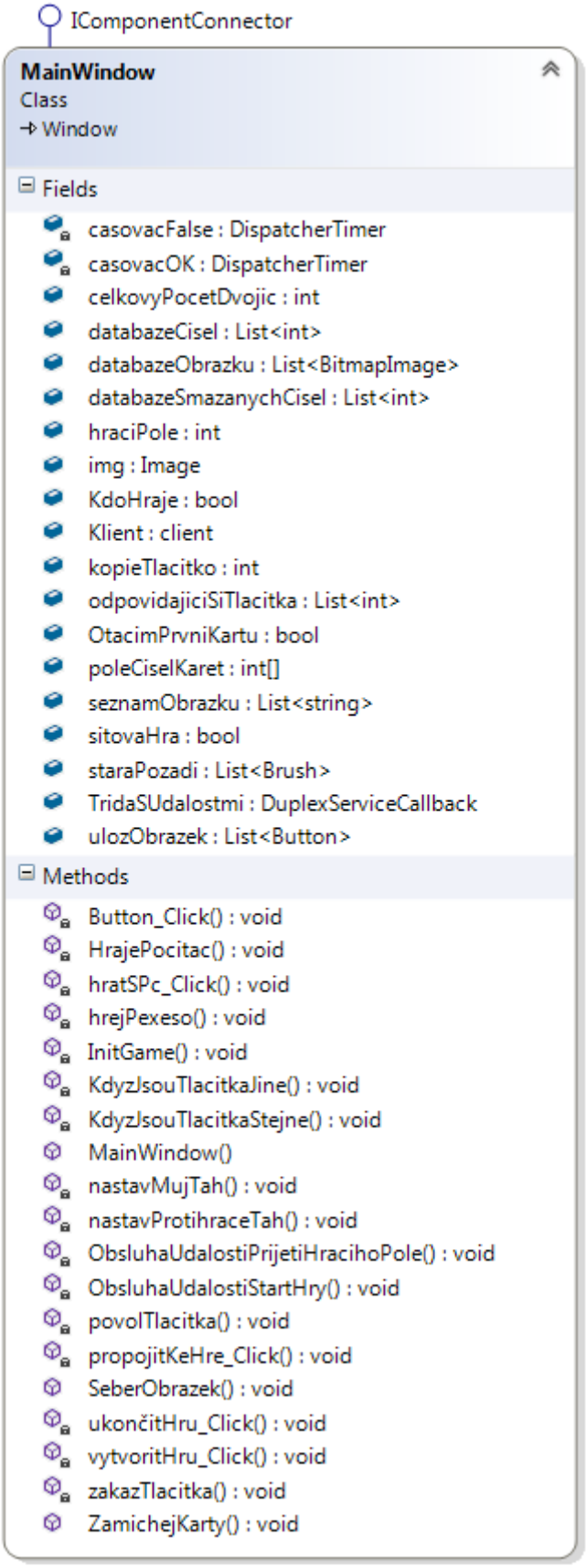
IV Seznam příloh

Visual Studio 2015 (reverse engineering diagramy):

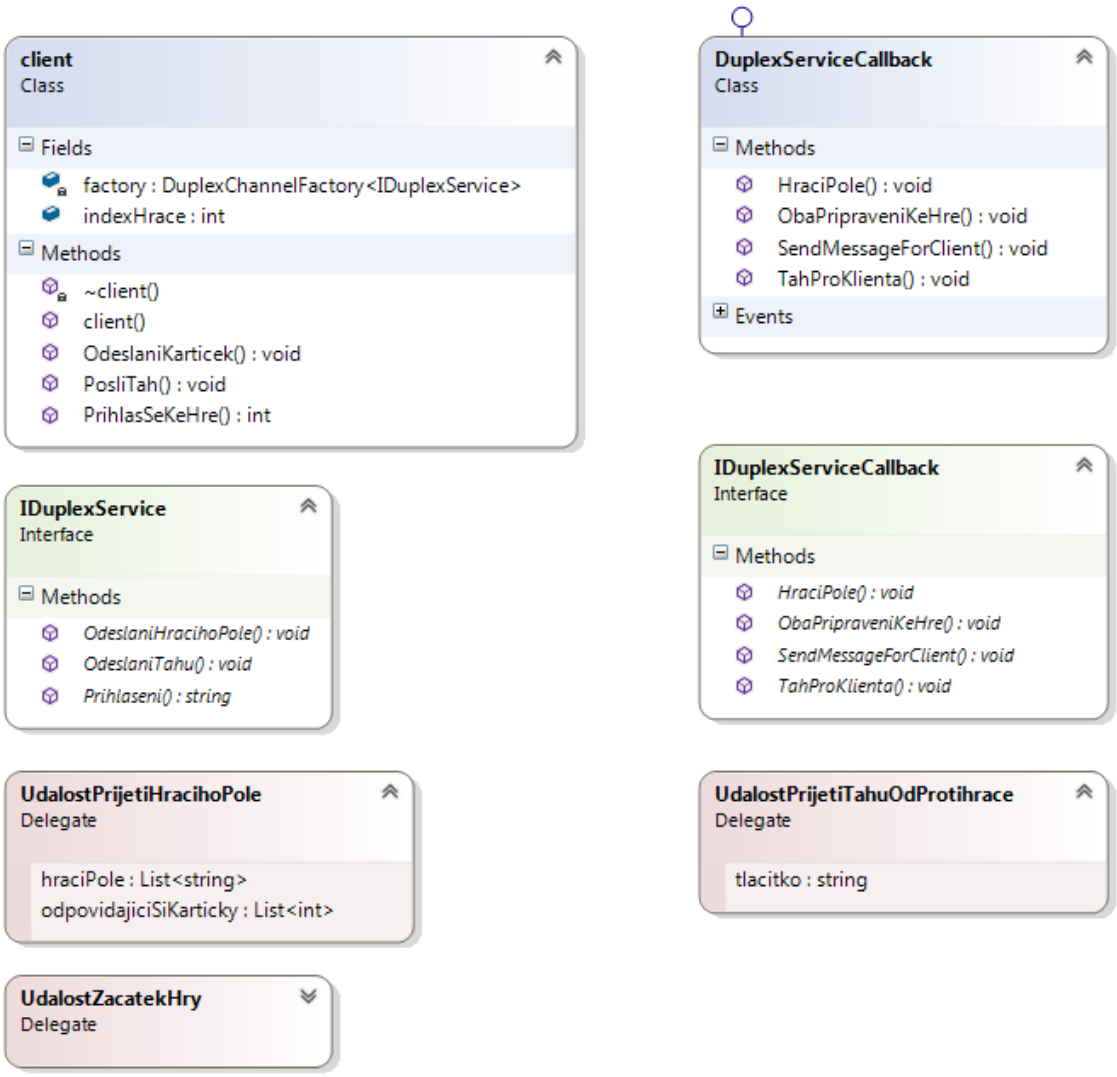
V Přílohy



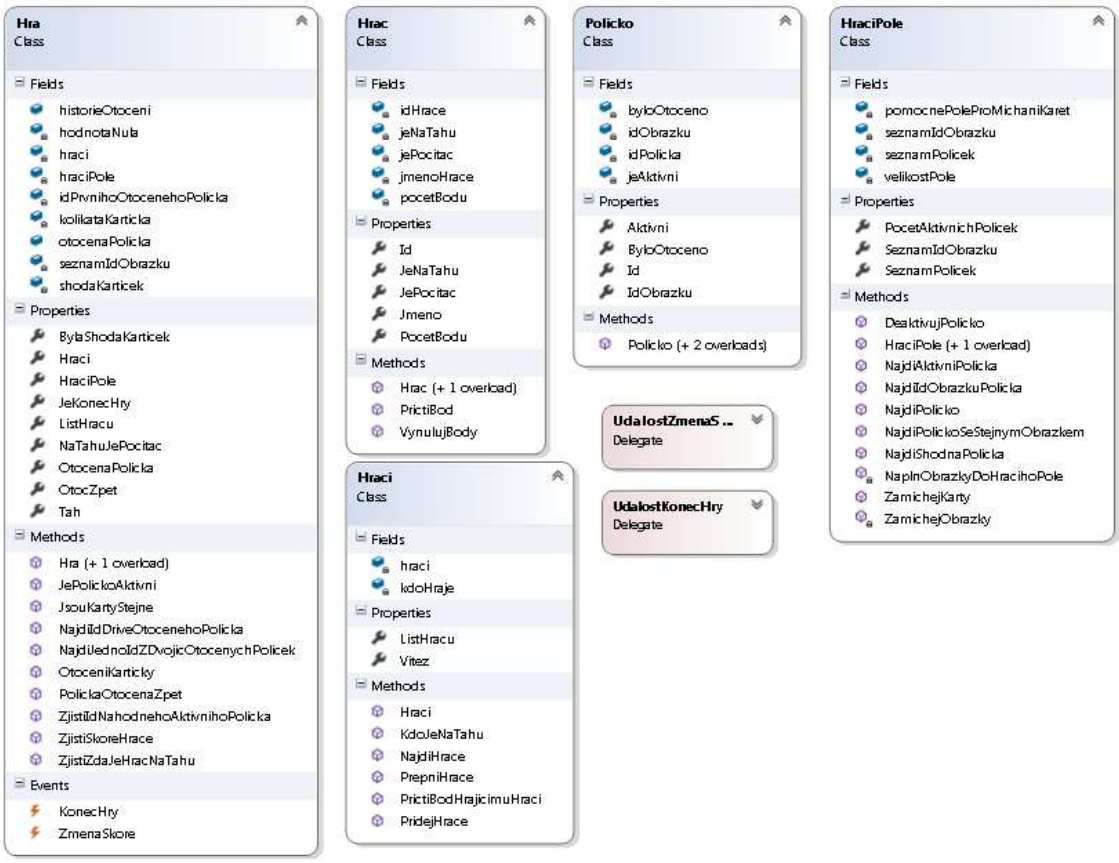
Příloha A: Diagram tříd zajišťujících síťovou komunikaci na straně serveru - původní aplikace



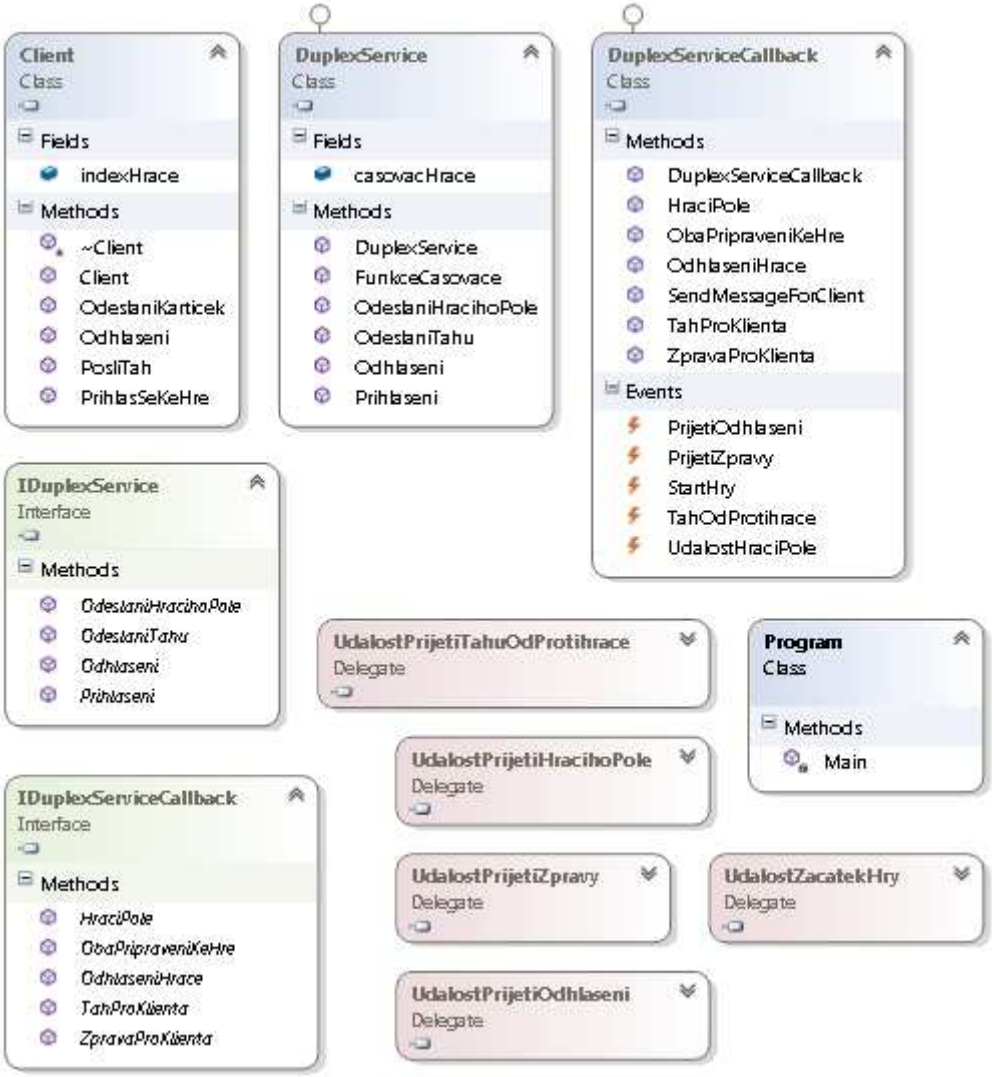
Příloha B: Diagram tříd MainWindow - původní aplikace



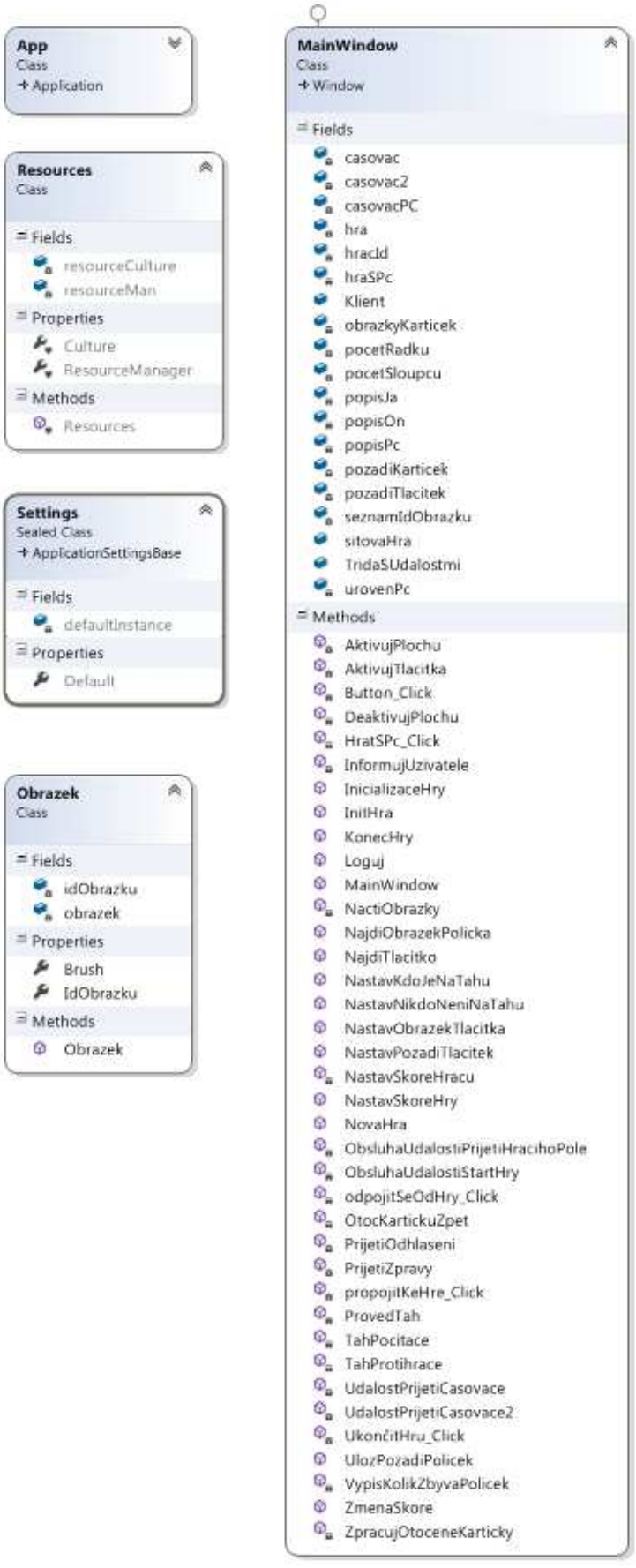
Příloha C: Diagram tříd zajišťujících síťovou komunikaci na straně klienta - původní aplikace



Příloha D: Diagram tříd části model - upravená aplikace



Příloha E: Diagram tříd - upravená aplikace



Příloha F: Diagram tříd - upravená aplikace