Česká zemědělská univerzita v Praze

Provozně ekonomická fakulta

Katedra informačního inženýrství

Bakalářská práce

Vývoj SW s využitím vkládání závislostí a IoC/DI kontejnerů v prostředí .NET

Rostislav Jirásek

ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE

Provozně ekonomická fakulta

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Rostislav Jirásek

Informatika

Název práce

Vývoj SW s využitím vkládání závislostí a IoC/DI kontejnerů v prostředí .NET

Název anglicky

SW development using dependency injection and IoC/DI containers in .NET

Cíle práce

Práce je zaměřena na problematiku vývoje aplikací v prostředí .NET, přičemž hlavním tématem je využití návrhového vzoru "vkládání závislostí". Hlavním cílem práce je popsat výhody, nevýhody a na praktických příkladech ukázat využití tohoto návrhového vzoru při vývoji moderních aplikací v prostředí .NET. Dílčím cílem je pak provést srovnání vybraných IoC/DI kontejnerů.

Metodika

Práce sestává ze dvou částí, teoretické a praktické.

Teoretická východiska pro zpracování praktické části, představující první část práce, budou popsána na základě syntézy poznatků získaných studiem odborné literatury.

Praktická část práce spočívá v návrhu a implementaci ukázkových aplikací využívajících vzor "vkládání závislostí" s využitím různých IoC/DI kontejnerů.

Jednotlivé implementace budou vzájemně porovnány a budou shrnuty zásadní implementační rozdíly. Poznatky získané během zpracování práce budou shrnuty a zhodnoceny.

Doporučený rozsah práce

35-40 stran

Klíčová slova

vývoj, software, C#, .NET, IoC, DI

Doporučené zdroje informací

BAHARESTANI, D. Mastering Ninject for Dependency Injection. Packt Publishing, 2013 142 s. ISBN: 978-1-7821-6620-7

BETTS, D., G. Melnik, F. Simonazzi a M. Subramanian. Dependency Injection with Unity. Microsoft patterns & practices, 2013 142 s. ISBN: 978-1-6211-4028-3

PRASANNA, D. R. Dependency Injection:Design patterns using Spring and Guice. Manning Publications, 2009 352 s. ISBN: 978-1-9339-8855-9

SEEMANN, M. Dependency Injection in .NET. Manning Publications, 2011 584 s. ISBN: 978-1-9351-8250

Předběžný termín obhajoby

2018/19 LS – PEF

Vedoucí práce

Ing. Jiří Brožek, Ph.D.

Garantující pracoviště

Katedra informačního inženýrství

Elektronicky schváleno dne 24. 1. 2019

Ing. Martin Pelikán, Ph.D.

Vedoucí katedry

Elektronicky schváleno dne 24. 1. 2019

Ing. Martin Pelikán, Ph.D.

Děkan

V Praze dne 19. 02. 2019

Čestné prohlášení

Prohlašuji, že svou bakalářskou práci " Vývoj SW s využitím vkládání závislostí a IoC/DI kontejnerů v prostředí .NET" jsem vypracoval samostatně pod vedením vedoucího bakalářské práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou citovány v práci a uvedeny v seznamu použitých zdrojů na konci práce. Jako autor uvedené bakalářské práce dále prohlašuji, že jsem v souvislosti s jejím vytvořením neporušil autorská práva třetích osob.

V Praze dne 15.3.2019 _____

Poděkování

Rád(a) bych touto cestou poděkoval Ing. Jiřímu Brožkovi Ph.D. za odborné vedení práce, trpělivost a za cenné rady, které mi pomohly k vypracování této práce. Dále bych chtěl poděkovat své rodině za velkou míru trpělivosti a podpory.

Vývoj SW s využitím vkládání závislostí a IoC/DI kontejnerů v prostředí .NET

Abstrakt

Práce se zaměřuje na vývoj software za použití návrhového vzoru vkládání závislostí v prostředí .NET. Součástí práce je návrh a implementace knihovny. Tato knihovna tvoří základ pro implementaci aplikací, které využívají různých IoC/DI kontejnerů a v kterých jsou demonstrovány možnosti vybraných kontejnerů. Jednotlivé implementace jsou v práci vzájemně porovnány. Další součástí práce je benchmark vybraných IoC/DI kontejnerů.

Klíčová slova: C#, .NET, vývoj, software, IoC, DI, kontejnery, OOP

SW development using dependency injection and IOC/DI containers in .NET

Abstract

Thesis focuses on development of software with usage of dependency injection design pattern in .NET. Part of this thesis is design and implementation of library. This library forms a base for implementation of applications, which are using different IoC/DI containers and in these applications are demonstrated features of chosen containers. Implementations are then compared to each other. Other part of this thesis is benchmark of chosen IoC/DI containers.

Keywords: C#, .NET, development, software, IoC, DI, containers, OOP

Obsah

Seznam obrázků

Seznam tabulek

Seznam použitých zkratek

API – soubor funkcí a procedur umožňující využití knihovny, služby, operačního systém atd. k vývoji software
DI – vkládání závislostí
GC – součást prostředí .NET obstarávající uvolňování prostředků z paměti
IoC – návrhový princip, při kterém komponenta nevytváří služby, které potřebuje ke své funkci, ale požaduje, aby jí tato služba byla poskytnuta externě
LINQ – soubor technologií .NET frameworku umožňující tvorbu dotazů přímo z programovacích jazyků
MVVM – třívrstvá architektura návrhu aplikace
OOD – objektově orientovaný návrh
OOP – objektově orientované programování
XML – značkovací jazyk

1 Úvod

Doba se neustále zrychluje. Mnoho technologií, které v minulosti lidstvo mohlo znát jen z filmů, dnes lidstvo již využívá nebo začne využívat v blízké budoucnosti. Technologický rozvoj stále více vyžaduje rozvoj sektoru informačních technologií. Výrobci hardwarových komponent neustále přicházejí s novými, rychlejšími a efektivnějšími komponentami, aby bylo možné uspokojit neustále se zvyšující nároky na výkon.

V softwarovém odvětví je rozvoj nových technologií ještě rychlejší, stále více se např. začínají využívat cloudové technologie. Tento trend rozvoje vyžaduje po vývojářích software, aby vyvíjeli aplikace s ohledem na možnost přizpůsobení změnám, které dnešní doba přináší.

Návrhový vzor vkládání závislostí vývojářům umožňuje vyvíjet aplikace, které se skládají s mnoha volně propojených komponent. Použití tohoto návrhového vzoru značně usnadňuje práci vývojářům, pokud jsou nuceni aplikaci rozšiřovat nebo upravovat, vzhledem ke změnám nároků na aplikaci.

V této práci je tento návrhový vzor podrobněji popsán a jeho použití je ukázáno na praktických příkladech.

2 Cíl práce a metodika

2.1 Cíl práce

Práce je zaměřená na problematiku vývoje software v prostředí .NET s použitím návrhového vzoru vkládání závislostí a IoC/DI kontejnerů.

Hlavním cílem práce je demonstrovat, jakým způsobem se vkládání závislostí dá využít, pro tvorbu moderních aplikací a jaké výhody nebo případné nevýhody s sebou tento přístup přináší. Demonstrace využití tohoto návrhového vzoru bude provedena na praktických příkladech využívajících IoC/DI kontejnerů.

Dílčím cílem práce je porovnání vybraných IoC/DI kontejnerů.

2.2 Metodika

Metodika práce spočívá v syntéze poznatků z odborných zdrojů, zabývajících se vývojem software s použitím návrhového vzoru vkládání závislostí.

Tyto poznatky následně budou využity, k návrhu a implementaci knihovny, která bude následně využita k implementaci jednotlivých aplikací, u kterých budou použity různé IoC/DI kontejnery.

Jednotlivé implementace pak budou autorem popsány a bude provedeno srovnání těchto implementací.

Následně bude vytvořen benchmark vybraných kontejnerů, na kterém autor otestuje výkonnostní a paměťové nároky jednotlivých vybraných kontejnerů. Výsledky tohoto benchmarku budou shrnuty.

Nakonec autor provede srovnání jednotlivých kontejnerů.

3 Teoretická východiska

3.1 .NET Framework a jazyk C#

3.1.1 .NET Framework

.NET Framework je vývojová platforma, představená firmou Microsoft v roce 2002, která je určena k vývoji moderního software pro operační systém Windows. Poskytuje programátorům služby usnadňující vývoj software a API, která abstrahují služby operačního systému Windows. (Microsoft, 2018)

Mezi klíčové vlastnosti .NET patří:

- **Správa paměti.** .NET obsahuje mechanizmus pro alokaci a uvolňování paměti bez nutnosti zásahu vývojářů.
- **Interoperabilita programovacích jazyků.** .NET umožňuje programátorům psát jednotlivé komponenty aplikací v různých programovacích jazycích.
- **Vývojové frameworky.** .NET obsahuje celou řadu vývojových frameworků pro různé oblasti vývoje aplikací. Ať už se jedná o webové aplikace, aplikace s přístupem k databázím nebo grafické aplikace pro operační systém Windows.
- **Zpětná kompatibilita.** Většina aplikací napsaných s použitím starší verze .NET Frameworku lze jednoduše převést na novější verzi bez nutnosti modifikací zdrojového kódu. (Microsoft, 2018)

3.1.2 Jazyk C#

Jazyk C# je programovacím jazykem, který patří spolu s Javou nebo C++ do skupiny programovacích jazyků, vycházejících z jazyka C. Světu byl představen firmou Microsoft v roce 2002, tedy ve stejném roce jako .NET framework. Síla jazyka C# spočívá v tom, že je založen na užitečných syntaktický konstruktech různých programovacích jazyků, k tomu přidává i vlastní unikátní konstrukty, to vše z možností využití výhod .NET frameworku, který poskytuje řadu užitečných knihoven dále rozšiřující možnosti tohoto jazyka.

„Protože jazyk C# je hybrid více programovacích jazyků, z něj činí jazyk, který je syntakticky čistý (ne-li čistší) jako Java, je podobně jednoduchý jako Visual Basic a téměř stejně mocný a flexibilní jako C++“ (Troelsen, a další, 2017 str. 6)

Stejně jako např. Java je jazyk C# plně objektově orientovaný, to znamená, že veškeré typy jsou objekty, ať už se jedná o enumeraci, číslo, textový řetězec nebo instance tříd, ale poskytuje i možnosti procedurálního programování. Poskytuje všechny aspekty OOP, tedy zapouzdření, polymorfismus, dědičnost a abstrakci.

Mezi užitečné funkce jazyka C# patří:

- Podpora LINQ
- Podpora generických, anonymních a dynamických typů
- Možnost vytvářet rozšiřující metody, které umožňují rozšířit třídu bez nutnosti vytváření potomka této třídy
- Možnost snadného asynchronního programování
- Podpora pro atributy tříd a jejích členů atd. (Troelsen, a další, 2017)

3.2 Návrhové vzory

„Návrhové vzory jsou praxí ověřené postupy v objektově orientovaném programování, které se zaměřují na řešení problémů v návrhu objektově orientovaných aplikací."

Jednotlivé návrhy se řadí do skupin dle účelu jejich použití. A to na:

- **Vytvářející**, které se zabývají problémem vytváření objektů. Např. Návrhové vzory Factory nebo Singleton
- **Strukturální**, které řeší problémy spojené se strukturou objektů a tříd v aplikaci. Např Návrhové vzory Adapter nebo Facade.
- **Chování**, obsahující návrhové vzory, které řeší interakci a předávání zodpovědností mezi objekty. Do této skupiny patří např. návrhové vzory Observer nebo Iterator. (Gamma, a další, 1995)

3.3 SOLID zásady objektově orientovaného designu

„SOLID je zkratka pro soubor pěti zásad v objektově orientovaném designu aplikací, které usnadňují vývojářům vyvíjet software, který se snadno udržuje a je snadné ho rozšířit." (Oloruntoba, 2015) Jedná se o:

- **Single-responsibility principle** – Třída by měla mít jeden a pouze jeden důvod ke změně, což znamená, že by každá třída měla v programu plnit pouze jednu funkci. Opakem této zásady je tzv. "božský objekt"[1]
- **Open-closed principle** – Třída by měla být otevřena rozšířením, ale uzavřena změnám. Mělo by tedy být umožněno rozšířit třídu bez nutnosti modifikace třídy samotné.
- **Liskov Substitution principle –** Každá třída, která je předkem jiných tříd, může být nahrazena svým potomkem, aniž by bylo nutné modifikovat předka.
- **Interface segregation principle –** Třída by neměla být závislá na interface, který nepoužívá.
- **Dependency Inversion principle** – Modul na vysoké úrovni by neměl záviset na modulu na nízké úrovni, ale měla by být mezi nimi vložena úroveň abstrakce. (Oloruntoba, 2015)

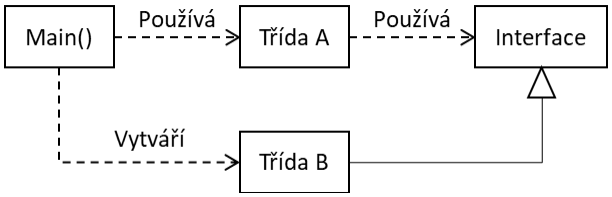
[1] Objekt, který je v programu zodpovědný za vše.

3.4 Závislosti mezi třídami

Jestliže třída A, potřebuje ke své funkci třídu B, vzniká mezi nimi závislost. Třída A se stává závislou na třídě B a zároveň třída B se stává závislostí třídy A. Takovéto třídy se též nazývají propojené (coupled).

Míra závislosti mezi třídami se dá rozdělit na dva protipóly, tedy třídy mohou být silně propojené (tightly coupled) jako na obrázku č. 1, volně propojené (loose coupled) jako na obrázku č.2, anebo někde mezi těmito protipóly. (Seeman, 2011)

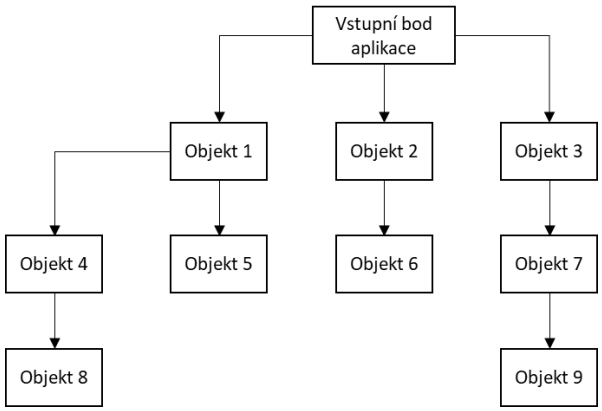
Obrázek č. 1 – Silně propojené třídy[2]



Obrázek č. 2 – Volně propojené třídy[3]

„Problém silně propojených tříd spočívá v nemožnosti vyměnit závislost bez nutné modifikace závislé třídy, což odporuje SOLID zásadám v objektově orientovaném designu.“ (Seeman, 2011)

Podle definice Hordějčuka: „Program složený z objektů je možno abstrahovat jako orientovaný graf, kde objekty jsou vrcholy grafu a závislosti představují hrany grafu.“ Tedy v případě sestavení diagramu, propojením všech tříd komponenty/aplikace jejich závislostmi vzniká tzv. graf objektů komponenty/aplikace, jak je možno vidět na obrázku č. 3. (Hordějčuk)



Obrázek č. 3 – Graf závislostí aplikace[4]

V zásadě existují dva přístupy v určení, kdo zodpovídá a vytváří závislosti objektu:

- Objekt vytváří a zodpovídá za své závislosti
- Závislost je do objektu vložena zvenčí

[2] Vlastní zdroj
[3] Vlastní zdroj
[4] Vlastní zdroj

Druhému přístupu se říká vkládání závislostí.

3.5 Vkládání závislostí

„Vkládání závislostí je soubor principů a návrhových vzorů, který umožňuje vytvářet volně propojený kód.“ (Seeman, 2011)

To znamená, že vkládání závislostí umožňuje snížit míru závislosti mezi jednotlivými komponenty aplikace a je tedy možné v závislé třídě vyměnit jednu komponentu, která je závislostí, za jinou, což je hlavním benefitem volně propojeného kódu, který se pomocí vkládání závislostí vytváří. (Baharestani, 2013)

Vkládání závislostí využívá principu Inversion of Control, který spočívá v tom, že třída nevytváří svou závislost, pouze signalizuje např. parametrem konstruktoru, že tuto závislost potřebuje, aby mohla vykonávat svou funkci a čeká, že ji tato závislost bude dodána. (Fowler, 2004) (Betts, a další, 2013)

3.5.1 Výhody vkládání závislostí

Vkládání závislostí s sebou přináší benefity. Mezi nejdůležitější benefity podle Seemana (2011) patří:

- **Late binding** – Určení třídy, která se stává závislosti je možno provést až při vytvoření instance závislé třídy. V některých případech je dokonce možné vyměnit jednu závislost za jinou bez nutnosti rekompilace kódu nebo vkládat závislosti na základě kontextu.
- **Rozšiřitelnost** – Jednu komponentu aplikace je možno rozšířit, aniž by bylo nutné zasahovat do ostatních komponent.
- **Paralelní vývoj** – Usnadňuje vývoj rozsáhlých aplikací v nezávislých týmech pracujících na různých komponentách aplikace.
- **Snadná údržba** – Přidávání nových funkcionalit do aplikace a odstraňování problémů je snazší, protože zodpovědnosti jednotlivých tříd jsou jasně definované.
- **Testovatelnost jednotek** – Umožňuje psaní jednotkových testů, protože jednotkové testy by měli být prováděny na jednotce, která je izolována od svých závislostí.

3.5.2 Nevýhody vkládání závislostí

Vkládání závislostí s sebou přináší i své nevýhody, které mohou řadu vývojářů odrazovat od jeho použití, mezi největší nevýhody tohoto přístupu patří:

- **Design aplikace** – Vkládání závislostí vyžaduje, aby aplikace byla na tento přístup připravená z hlediska designu
- **Trasování programu** – Vkládání závislostí ztěžuje trasování programu tím, že odděluje sestrojení objektu od jeho následného použití
- **Mnoho malých tříd** – Rozdělením aplikace na třídy s jedinou zodpovědností, vytváří v aplikaci mnohem více tříd než při tradičních přístupech

Závisí tedy na vývojářích, zda je pro jejich aplikaci výhodné použít vkládání závislostí s ohledem na požadavky aplikce s přihlédnutím k těmto výhodám/nevýhodám. (Seeman, 2011) (What are the downsides to using Dependency Injection?, 2011)

3.6 Možnosti implementace vkládání závislostí

Existují tři základní možnosti, kterými lze implementovat vložení závislosti. Jsou to: vložení závislosti konstruktorem třídy, vložení závislosti do vlastnosti třídy a vložení závislosti do metody.

3.6.1 Vložení závislosti konstruktorem

Nejpoužívanějším způsobem vkládání závislosti je vložení závislosti do konstruktoru třídy.

```
public class SomeClass
{
    // Reference na závislost
    private readonly ISomeInterface _dependency;

    // Veřejný konstruktor třídy s vloženou závislostí v parametru
    public SomeClass(ISomeInterface dependency)
    {
        // Test nulové reference garantuje, že závislost byla vložena
        _dependency = dependency ?? throw new ArgumentNullException("dependency");
    }
}
```

Třída, vyžadující závislost na jiné třídě musí obsahovat veřejný konstruktor, který přijímá instanci závislosti jako parametr. Pokud není závislost závislé třídě dodána, kód nelze zkompilovat. Je důležité, aby třída neobsahovala výchozí konstruktor, protože by pak bylo možné vložení závislosti obejít.

Jelikož třídy se řadí mezi referenční typy, je možné dosadit za instanci závislosti nulovou referenci, což umožní zkompilování kódu. Proto je potřeba tuto skutečnost vyloučit testem na nulovou referenci.

Dobrým způsobem, jak uchovat referenci na závislost, je uložení reference do readonly pole, aby nebylo možné pole nechtěně modifikovat po sestrojení objektu.
V opačném případě by bylo možné nastavit referenci na závislost na null a obejít tím test umístěný v konstruktoru třídy. (Seeman, 2011)

3.6.2 **Vložení závislosti do vlastnosti**

Dalším způsobem, kterým je možno vložit závislost, je využití vložení do vlastnosti třídy s použitím setteru.

```
public class SomeClass
{
    // Veřejná vlastnost umožňující vložení reference na závislost
    public ISomeInterface Dependency { get; set; }
}
```

Závislá třída v tomto případě obsahuje veřejnou vlastnost, do které je možno vložit referenci na závislost.

Tento způsob vložení závislosti s sebou přináší problém. Na rozdíl od vložení závislosti v konstruktoru třídy, není možné garantovat, že závislost bude k dispozici po celou dobu životního cyklu objektu. Je to způsobeno tím, že reference závislosti nemusí být nastavena před jejím prvním použitím, a dokonce může být nastavena na nulovou referenci v běhu programu.

Řešením tohoto problém je nastavení výchozí instance závislosti v konstruktoru a přidání testu na nulovou referenci do setteru vlastnosti obsahující závislost.

Nejlepším využitím vložení závislosti do vlastnosti je v případě volitelné závislosti, kde k výše popsanému problému nemůže dojít. (Prasanna, 2009) (Seeman, 2011)

3.6.3 **Vložení závislosti s pomocí metody**

Poslední možností je vložení závislosti prostřednictvím parametru metody.

```
public class SomeClass
    {
        // Veřejná metoda s referncí na závislost v parametru
        public string SomeMethod(ISomeInterface dependency)
        {
            // pokud závislost není null vrátí hodnotu property závislosti
            return dependency?.SomeStringProperty
                    ?? throw new ArgumentNullException("dependency");
        }
    }
```

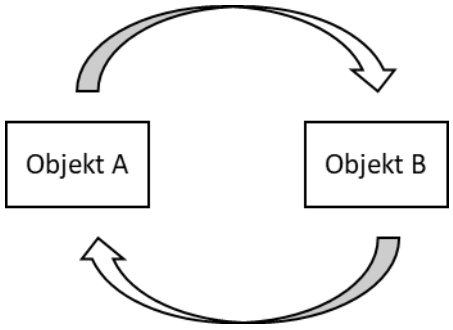
V tomto případě, závislá třída obsahuje veřejnou metodu, do které vkládáme závislost skrz parametr metody. To umožňuje měnit závislosti při každém volání metody.

„Využití této metody je například v případě potřeby informovat závislou třídu o aktuálním kontextu volání.“ (Seeman, 2011)

3.6.4 **Běžné problémy s vkládáním závislostí**

3.6.4.1 Problém cyklické závislosti

V případě vkládání závislostí konstruktorem se může stát, že dva objekty na sobě závisí navzájem. Objekt A vyžaduje ke svému sestrojení objekt B a objekt B zase vyžaduje objekt A, což je zobrazeno na obrázku č. 4. Jedná se tedy o typický příklad problému slepice-vejce.[5]



Obrázek č. 4 – Problém cyklické závislosti[6]

„Tento problém nemusí být pouze mezi dvěma objekty, ale může zahrnovat mnohem více objektů, pokud jejich postupné závislosti uzavřou kruh. Neexistuje postup, kterým by se dala definovat strategie sestavování objektů, a proto v případě cyklických závislostí nelze zkonstruovat ani jeden z objektů.“ (Prasanna, 2009)

Řešení tohoto problému je vložení závislosti do vlastnosti, alespoň u jednoho z objektů, což vede k přerušení cyklu a objekty je pak možné sestrojit. Tento postup by se neměl používat často, protože cyklické závislosti jsou pach v kódu, který upozorňuje na to, že aplikace není dobře navržená. Podle Seemana (2011) je v dobrém návrhu aplikace rozložena do vrstev a každá třída by měla komunikovat pouze v rámci své vrstvy nebo s vrstvou na nižší úrovni, která bezprostředně následuje. A cyklická závislost důkazem porušení této metodologie, ačkoliv existují i výjimky. Jedna z výjimek je použití vkládání závislostí v WPF, pokud je použita architektura Model-View-ViewModel[7]. Zde View závisí na ViewModelu a naopak. V takovýchto případech je použití vložení závislosti do vlastnosti třídy nezbytné.

3.6.4.2 Vkládání příliš mnoha závislostí

Dalším problém může být situace, že třída vyžadují příliš mnoho závislostí, které do ní musí být vloženy. Podle Seemana se jedná o ukázku toho, že třída má v aplikaci více zodpovědností, což ukazuje na porušení Single Responsibility principu a třída by měla být refaktorována, aby se tento problém v návrhu odstranil. (Seeman, 2011)

```
public class SomeClass(
       SomeDependency depedency,
       OtherDependency otherDependency,
       AnotherDependency AnotherDependency,
       SomeService service, OtherService otherService)
```

[5] Tento problém byl definován již ve starověku Řeckými filozofy. Jedná se o otázku, zda bylo dříve vejce nebo slepice.
[6] Vlastní zdroj
[7] Model-View-ViewModel je třívrstvá architektura pro návrh aplikací s grafickým uživatelským rozhraním.

3.7 IoC/DI kontejnery

„IoC/DI kontejnery jsou softwarové knihovny, které usnadňují sestrojování objektů a objektových grafů a řízení životního cyklu objektů.“ (Seeman, 2011)

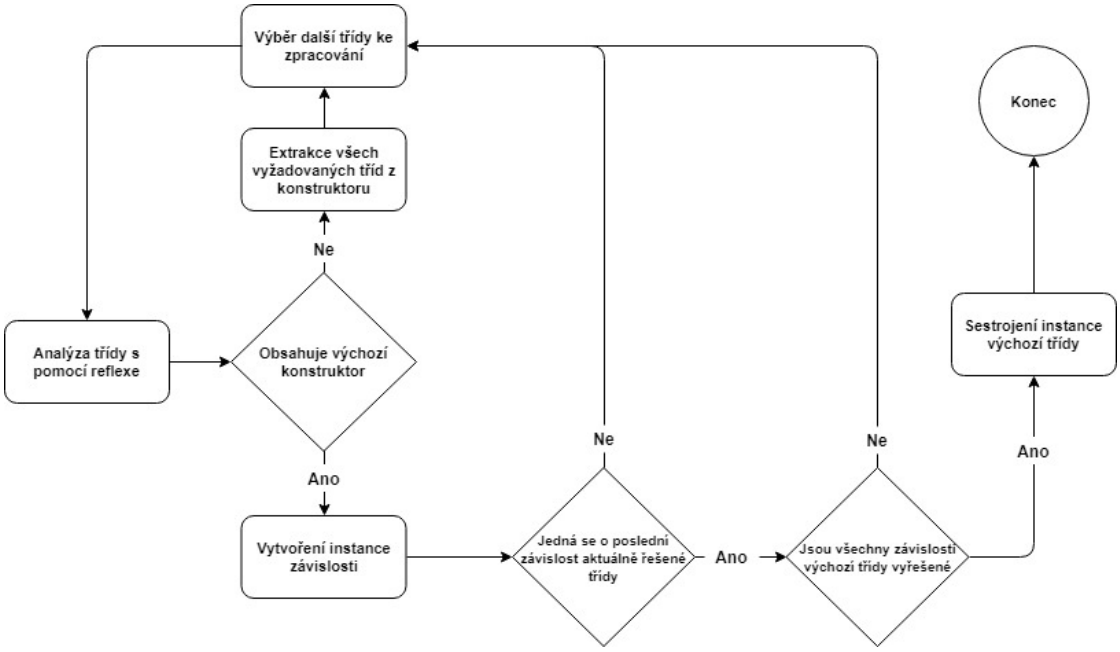
Jedná se o Frameworky, které na sebe berou zodpovědnost za sestrojování objektů závislostí a přiřazování těchto objektů jednotlivým příjemcům, kteří danou závislost vyžadují. Přístup využívající předání zodpovědnosti za vytváření objektů na jednu komponentu aplikace, v tomto případě kontejner, plně vyhovuje Single Responsibility principu v SOLID zásadách OOD. Kontejner v aplikace plní pouze jednu funkci stejně jako jiná komponenta aplikace obstarává pouze logování nebo komunikaci s databází. (Seeman, 2011)

3.7.1 Auto-Wiring

Auto-Wiring je jedna z klíčových služeb, které IoC/DI kontejnery poskytují. Jedná se o schopnost sestrojit celý objektový graf na základě informací, které kontejner získá analýzou všech tříd, které jsou potřebné pro sestrojení dané komponenty nebo i celé aplikace.

Podle Seemana (2011), algoritmus auto-wiringu spočívá v rekurzivním procházení tříd potřebných v dané komponentě, analýze konstruktorů s pomocí reflexe a následném vytváření instancí těchto tříd, pokud jsou vyřešeny veškeré závislosti potřebné pro vytvoření instance dané třídy. Algoritmus poté pokračuje, dokud není sestrojen celý graf objektů vyžadované komponenty.

Je nutné si uvědomit, že vzhledem tomu, že kontejner prochází pouze konstruktory tříd, auto-wiring funguje správně pouze při konstruktorovém vkládání závislostí a je nutné, aby třídy byly na tento způsob vkládání závislosti připraveny. Algoritmus auto-wiringu je zobrazen na obrázku č. 5.



Obrázek č. 5 – Algoritmus auto-wiringu[8]

[8] Vlastní zdroj

3.7.2 **Použití IoC/DI kontejnerů**

Použití kontejneru se zpravidla provádí ve třech fázích, jedná se o registraci komponent, vyřešení závislostí a uvolnění komponent z kontejneru.

3.7.2.1 Registrace komponent

V této fázi probíhá konfigurace kontejneru. Kontejneru je potřeba sdělit, jaké konkrétní třídy má v dané komponentě aplikace používat a jejich mapování na abstraktní typy.

Mapování tříd na abstraktní typy způsobí, že kontejner při auto-wiringu vloží do konstruktoru obsahujícím závislost, instanci třídy, která byla definována během mapování. Třídu je možno specifikovat i pomocí interface. V tomto případě se musí specifikovat i třída, která interface implementuje a instance této třídy bude následně použita ve všech závislostech

V následujícím kódu je tedy každá závislost v objektovém grafu definovaná pomocí interface ISomeInterface implementována třídou SomeClass.

```
// Registrace a mapování komponent pro Castle Windsor kontejner
container.Register(
   Component.For<SomeService>(),
   Component.For<ISomeInterface>()
          .ImplementedBy<SomeClass>()
);
```

Během registrace komponent je možno specifikovat celou řadu dalších nastavení, které dále upřesňují, jak se bude daná vložená závislost chovat, jakým způsobem bude závislost sestrojena, zdali má být vložena již sestrojená instance objektu atd. (Castle Project, 2017)

3.7.2.2 Vyřešení závislostí

Zde probíhá vyřešení všech závislostí potřebných k sestrojení grafu objektů dané komponenty aplikace.

```
// vyřešení závislostí a sestrojení grafu objektů s použitím Castle Windsor
var service = container.Resolve<SomeService>();
```

Pro většinu aplikací je třeba vyřešit závislosti pouze jednou, ale existují i výjimky. Například u aplikací, které odpovídají na požadavky (webové aplikace) je někdy žádoucí, sestrojit danou komponentu, při každém požadavku. Je potom tedy možno, sestrojit jen tu komponentu, která bude potřeba, pro vyřízení daného požadavku. (Seeman, 2011)

3.7.2.3 Uvolnění komponent

Během této fáze dochází k uvolnění všech sestrojených komponent z kontejneru.

```
// uvolnění komponenty z kontejneru s použitím Castle Windsor
container.Release(service)
```

Uvolnění sestrojených objektových grafů poté, co již nejsou potřebné pro chod aplikace, je jedním z osvědčených postupů v programování.

„Uvolnění komponent z kontejneru je speciálně důležité v případě, že komponenty implementují IDisposable interface[9].“ (Seeman, 2011)

3.7.3 **Způsoby registrací komponent IoC/DI kontejneru**

Většina kontejnerů umožňuje různé způsoby registrací komponent, které se v zásadě rozdělují do tří kategorií.

3.7.3.1 Automatická registrace

Některé kontejnery umožňují nevytvářet žádné registrace pro komponenty, místo toho kontejner automaticky doplní veškeré závislosti do aplikace na základě skenování tříd za pomocí reflexe. Jediná věc, která se u kontejneru musí specifikovat je, z které assembly má brát třídy, které budou doplněny jako závislosti. Některé kontejnery dokonce podporují automatické mapování komponent na jejich výchozí interface, které implementují. Tento přístup ale dává vývojářům menší kontrolu nad tím, jaká závislost bude do komponenty vložena. (Seeman, 2011)

3.7.3.2 Registrace pomocí kódu

Registrace pomocí kódu je nejpoužívanějším způsobem konfigurace kontejneru. Vývojář explicitně definuje, které závislosti bude kontejner doplňovat do komponent a na jaká interface se bude daná závislost mapovat. Nevýhodou tohoto přístupu je nutné rekompilace, při výměně závislosti. (Seeman, 2011)

3.7.3.3 Registrace pomocí XML souboru

Registrace pomocí XML souboru je poslední z možností registrací závislostí v kontejneru. V tomto případě se registrace umístí buď do souboru app.config nebo do jiného souboru ve formátu XML a kontejneru se předá umístění tohoto souboru. Výhodou tohoto přístupu je, že výměna jedné závislosti za jinou může být provedena bez nutnosti rekompilace kódu. Hlavní nevýhodou je, že konfigurace tímto způsobem obsahuje velké množství kódu a je mnohem hůře čitelná. (Seeman, 2011)

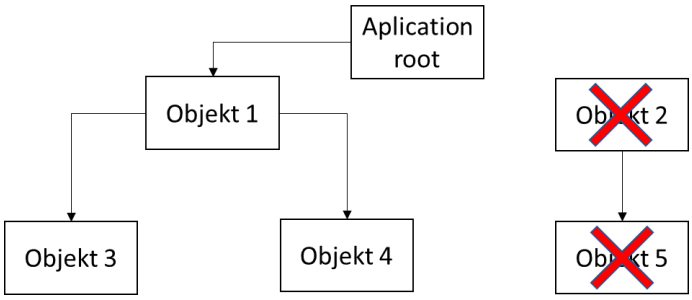
[9] Komponenty, které implementují IDisposable interface využívají zdroje, u kterých je potřeba explicitně zajistit uvolnění místa v paměti. (Troelsen, a další, 2017)

3.7.4 **Řízení životního cyklu závislostí**

3.7.4.1 Životní cyklus objektu

Životní cyklus objektu je doba od sestrojení objektu po jeho ukončení. V .NET frameworku rušení objektů obstarává tzv. Garbage Collector, který obstarává uvolnění objektů z paměti, poté co objekt již není potřeba pro běh aplikace.

„Pro identifikaci objektů, které je potřeba ukončit, GC sestrojí strom objektů, které je možno trasovat ze vstupního bodu aplikace a ukončí objekty, které se v tomto stromu nenachází.“ (Chodounský, 2017) Toto je možné vidět na obrázku č. 6.

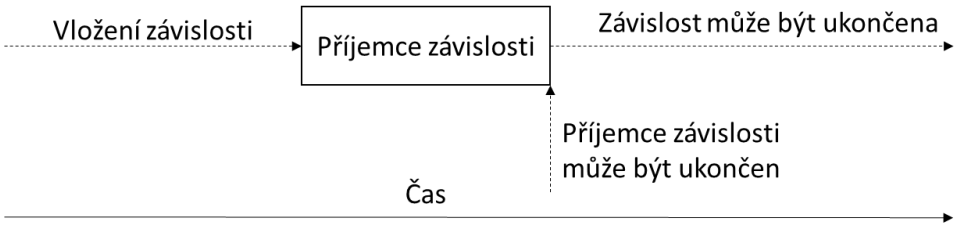


Obrázek č. 6 – Ukončování objektů pomocí GC[10]

3.7.4.2 Životní cyklus závislosti

V případě použití IoC/DI kontejneru není závislá třída zodpovědná za životního cyklus svých závislostí, protože figuruje pouze jako příjemce závislosti. Závislá třída nesestrojila objekt, na kterém závisí a v případě, že je instance závislé třídy ukončena, závislost může v programu existovat i nadále, pokud na ni jiná instance libovolné třídy drží referenci.

Ilustrace životního cyklu závislosti je na obrázku č. 7.



Obrázek č. 7 – Životní cyklus závislosti[11]

3.7.4.3 Životní styl závislosti

IoC/DI kontejnery umožňují explicitně definovat tzv. životní styl, pro každou závislost. Životní styl závislosti, kontejneru značí, jakým způsobem má tuto závislost přidělovat příjemcům závislosti. Je možné kontejneru říci, aby do každého příjemce závislosti vložil novou instanci závislosti nebo aby se odlišné závislé třídy stali příjemcem stejné instance závislosti. Nastavení životního stylu se provádí během registrace komponent, pro každou komponentu zvlášť.

Mezi hlavní životní styly závislostí patří singleton, transient a per graph.

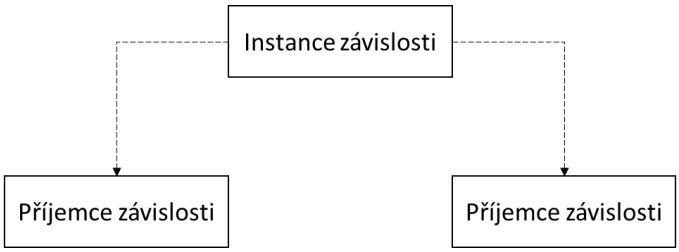
[10] Vlastní zdroj
[11] (Seeman, 2011 str. 217)

3.7.4.3.1 Singleton

Jak již název napovídá[12], životní styl singleton u závislosti způsobí, že kontejner vytvoří jedinou instanci závislosti, která je pak sdílena se všemi příjemci této závislosti, což je možné vidět na obrázku č. 8.

Nezaměňovat tento životní styl s návrhovým vzorem stejného názvu[13], protože životní styl je platný pouze pro kontejner a je tedy možné, na rozdíl od třídy aplikující návrhový vzor singleton, zkonstruovat více instancí této třídy.

Tento životní styl je většinou velmi efektivní z hlediska nároků na paměťový prostor, je proto vhodné ho používat, kdykoli je to možné. Hlavní problém znemožňující použití tohoto životního stylu je v případě, že komponenty využívající závislost nejsou thread-safe[14].

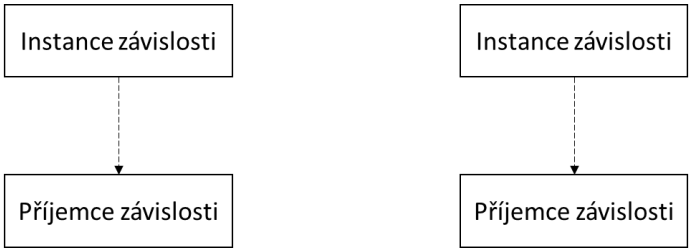


Obrázek č. 8 – Singleton lifestyle[15]

3.7.4.3.2 Transient

Životní styl transient, na rozdíl od singeltonu, vytváří novou instanci závislosti pro každého příjemce závislosti. Jak je vidět na obrázku č. 9. Toto znamená, že se jedná o nejbezpečnější životní styl pro závislosti z pohledu vláknové bezpečnosti, ale také nejméně efektivní z paměťového i výkonnostního hlediska.

Vzhledem k neefektivitě by se měl používat pouze v případech, kdy není možno použít jiný životní styl.



Obrázek č. 9 – Transient lifestyle[16]

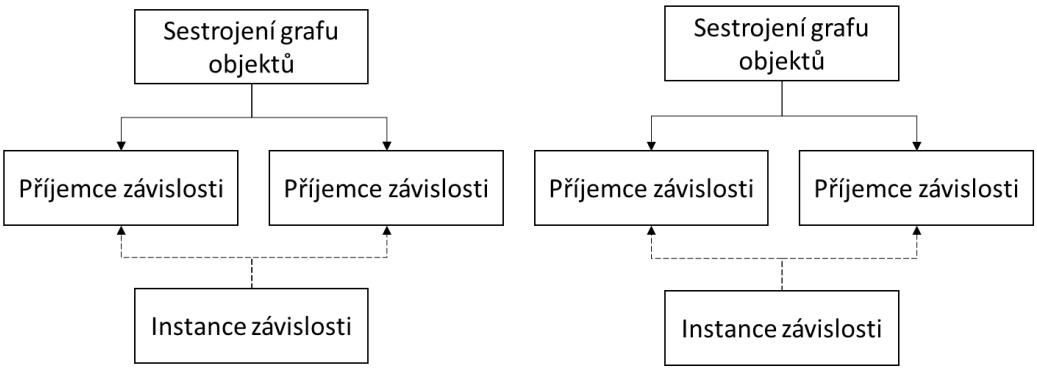
[12] Singleton znamená v anglickém jazyce „jedináček“.
[13] V objektově orientovaném programování se singleton používá i jako název návrhového vzoru. Při každém pokusu o vytvoření instance třídy aplikující tento návrhový vzor se zkontroluje, zda již v aplikaci neexistuje jiná instance této třídy a pokud ano, vrátí referenci na tuto instanci třídy. V opačném případě zkonstruuje novou instanci třídy (Gamma, a další, 1995).
[14] „Thread-safe je pojem v programování, který vyjadřuje, že danou část programu je možno zavolat z rozdílných vláken procesu, aniž by docházelo k problémům spojených s použitím sdílených dat konkurenčními vlákny.“ (Rouse, 2005)
[15] Vlastní zdroj
[16] Vlastní zdroj

3.7.4.3.3 Per Graph

Per Graph je životní styl, který kombinuje bezpečnost stylu transient a efektivitu stylu singleton. Princip tohoto životního stylu spočívá ve vytvoření jedné instance závislosti na jeden sestrojený graf objektů.

„Bezpečnost z pohledu vláken spočívá v předpokladu, že vlákno, které sestrojuje graf objektů je stejné jako vlákno, které daný objektový graf používá.“ (Seeman, 2011)

V případě použití tohoto životního stylu je nutné si uvědomit, zda neexistuje možnost, že jednotliví příjemci závislosti budou spuštěny na rozdílných vláknech.



Obrázek č. 10 – Per Graph lifestyle[17]

3.7.5 **Příklady používaných IoC/DI kontejnerů**

Na trhu je celá řada IoC/DI kontejnerů, s různými možnostmi, pro vytváření aplikací. Níže jsou příklady používaných kontejnerů:

- **AutoFac –** Autofac je vyspělý, velmi dobře zdokumentovaný kontejner, nabízející celou řadu pokročilých funkcí s velmi konzervativním typem registrací komponent. Komponenty se vkládají do objektu, pomocí kterého se poté vytvoří kontejner. Následně již není možné přidávat další závislosti do kontejneru. (Autofac Contributors, 2018)
- **Ninject** – Ninject je jedním z velmi, používaných kontejnerů, který poskytuje i pokročilé funkce. Jedná se o první kontejner, který umožňoval vkládání závislostí na základě kontextu. (Ninject, 2018)
- **Castle Windsor** – Castle Windsor (dále jen Windsor) je jedním z nejpoužívanějších kontejnerů na trhu. Jedná se o velmi dobře zdokumentovaný a podporovaný kontejner, obsahující spoustu pokročilých funkcí. (Castle Project, 2017)
- **StructureMap –** StructureMap je nejstarším nepřetržitě používaným kontejnerem na trhu. Obsahuje celou řadu pokročilých funkcí, je velmi dobře zdokumentován a patří mezi nejpoužívanější kontejnery. (StructureMap contributors, 2018)
- **Unity** – Unity kontejner byl původně vyvinut společností Microsoft, která ho dlouhou dobu podporovala. V roce 2015 byl tento kontejner předán novým vlastníkům, kteří se na jeho vývoji podíleli. (Landwerth, 2015)

[17] Vlastní zdroj

4 Vlastní práce

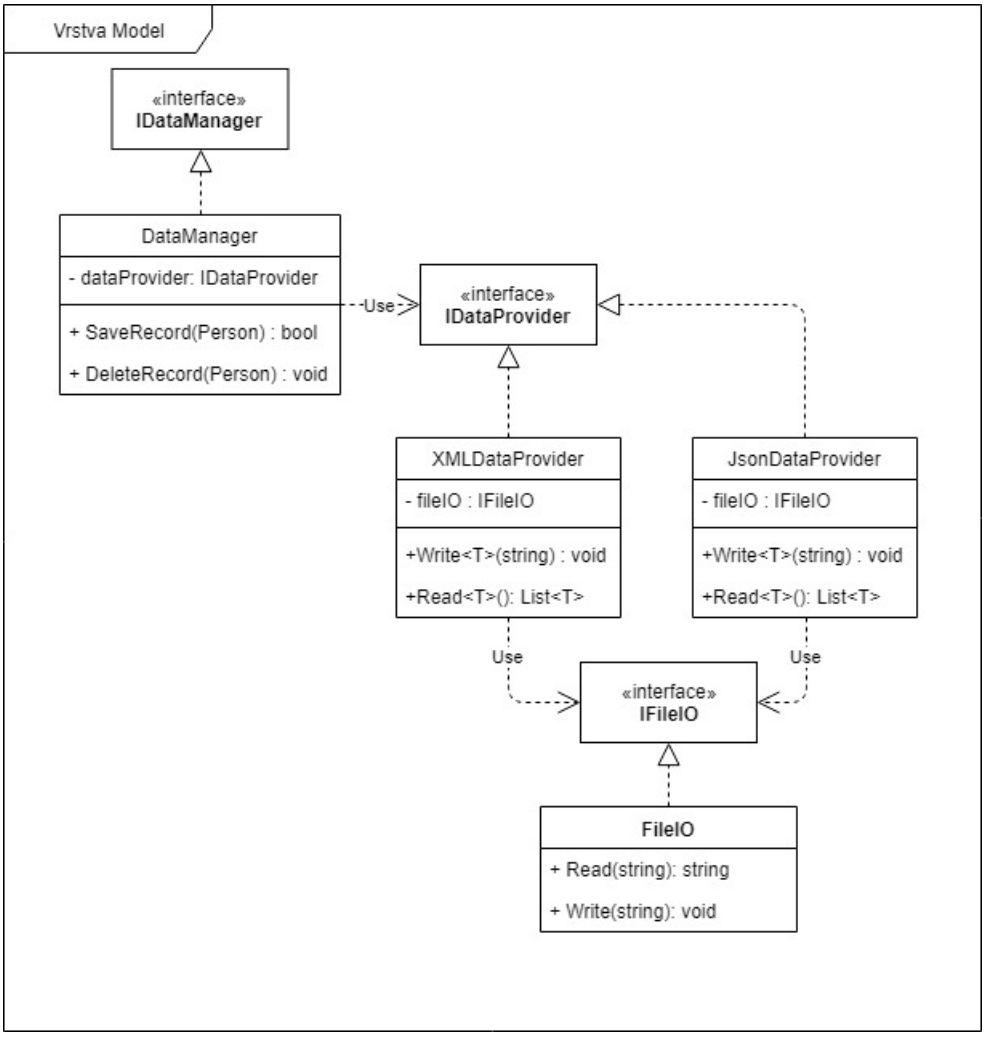
4.1 Návrh a implementace aplikace

Pro představení možností jednotlivých kontejnerů se autor rozhodl vytvořit jednoduchou aplikaci s GUI, založené na třívrstvé architektuře MVVM, která provádí serializaci a zápis kontaktů do souboru, kontakty je dále možno upravovat nebo mazat a je také možné vyhledávat kontakty na základě příjmení osoby nebo telefonního čísla.

Aplikace je implementována formou knihovny. Tato kapitola obsahuje stručný popis implementace této knihovny, která je dále využívána v rámci jednotlivých implementací za pomocí vybraných kontejnerů, které jsou popsány v kapitole 4.2.

4.1.1 Vrstva Model

Tato vrstva v aplikaci představuje datovou vrstvu aplikace a obstarává veškerou logiku aplikace na datové části. Zjednodušený UML diagram této vrstvy je na obrázku č. 11.



Obrázek č. 11 – UML diagram vrstvy "Model"[18]

[18] Vlastní zdroj

4.1.1.1 DataManager

Třída DataManager se stará o vytváření, modifikaci a mazání záznamů v soukromé kolekci a následnou synchronizaci záznamů v souboru prostřednictvím data provideru, který je do třídy vložen skrz konstruktorové vkládání závislostí. Jeho interface tvoří dvě metody, SaveRecord a DeleteRecord.

Metoda SaveRecord nejprve zkontroluje, zda jsou načtena data ze souboru, poté zda záznam neexistuje v interní kolekci, následně zjistí, zda se jedná o nový záznam nebo se jedná o modifikaci již existujícího záznamu a podle výsledku zavolá odpovídající private metodu InsertRecord nebo ModifyRecord. Tyto metody záznam vytvoří/modifikují a následně zajistí synchronizaci dat mezi aplikací a souborem pomocí privátní metody Synchronize.

Metoda DeleteRecord smaže záznam z interní kolekce na základě Id záznamu a zajistí synchronizaci dat opět pomocí metody Synchronize.

4.1.1.2 DataProvider

Třídy XmlDataProvider a JsonDataProvider se starají o serializaci dat do odpovídajícího formátu a předání výsledku třídě starající se o zápis do souboru.

Obě implementují interface IDataProvider a je tedy možné libovolně měnit, která implementace DataProvideru bude použita. Interface je tvořen metodami Write a Read, které jsou generické a je tedy možné je použít pro jakýkoliv serializovatelný objekt.

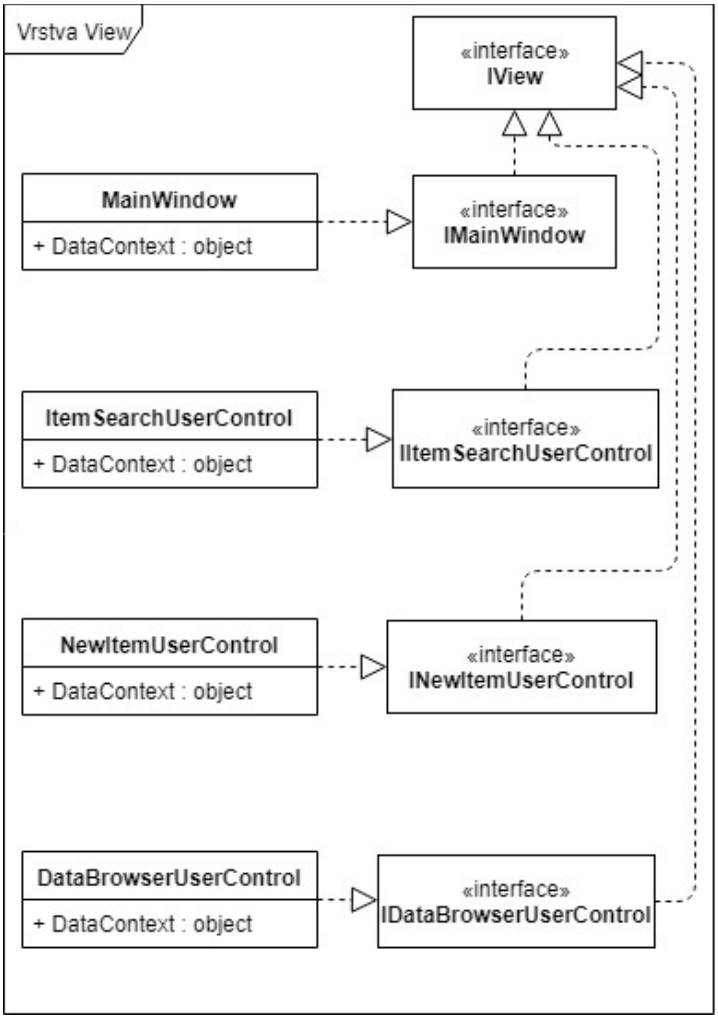
4.1.1.3 FileIO

Třída FileIO obstarává operace zápisu nebo čtení nad datovým souborem aplikace. Tato třída závisí na názvu souboru, který je do třídy vložen kontejnerem na základě konfigurace uložené v konfiguračním souboru aplikace.

4.1.2 **Vrstva View**

Vrstva View představuje jednoduché GUI aplikace vytvořené s pomocí frameworku WPF. Jelikož autor použil pro implementaci aplikace architekturu MVVM, celé grafické rozhraní je napsáno pomocí značkovacího jazyka XAML. Tento přístup umožňuje, že vrstva view neobsahuje žádný kód na pozadí[19]. Zodpovědnost za interakční logiku tedy nemá přímo vrstva view, ale s pomocí tzv. bindingu view komunikuje s odpovídajícím view modelem. Aby toto bylo možné, je potřeba vložit view model, starající se o interakční logiku daného view, do vlastnosti DataContext, kterou dané view obsahuje. V aplikaci je toto řešeno vložením závislosti z pomocí konstruktoru a modifikací chování jednotlivých kontejnerů.

V aplikaci tvoří tuto vrstvu čtyři třídy, z nichž jedna je typu Window a ostatní jsou typu UserControl. Každá třída má své vlastní interface, které implementuje a toto interface vychází z interface IView, které zaručuje, že všechny třídy této vrstvy obsahují vlastnost DataContext, pro vložení view modelu. Zjednodušený UML diagram této vrstvy je na obrázku č. 12.



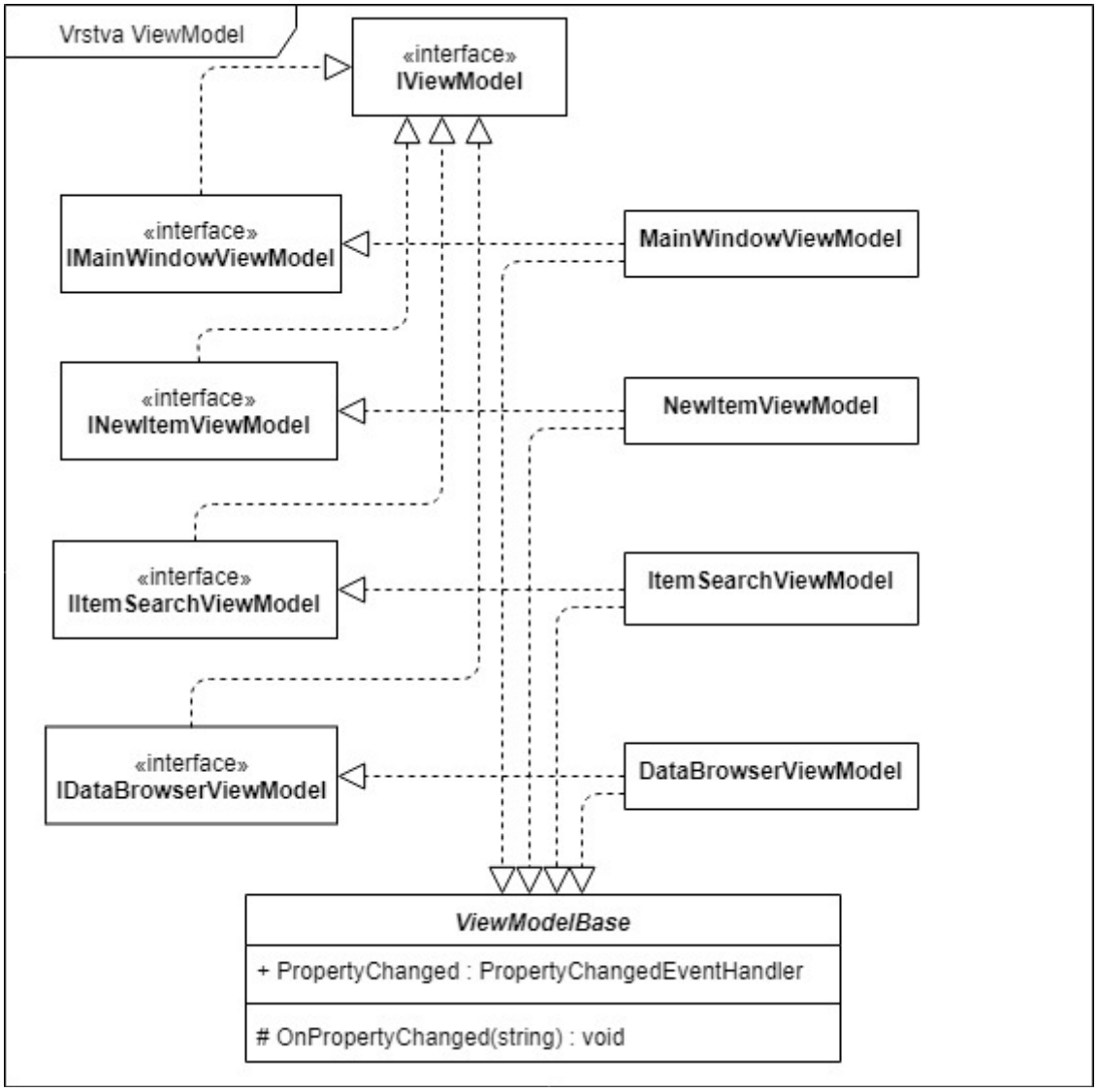
Obrázek č. 12 – Zjednodušený UML diagram vrstvy "View"[20]

[19] V WPF je termínem „kód na pozadí“ označován kód umístěný ve view, který se stará o interakční logiku view (Sexton, 2011)

[20] Vlastní zdroj

4.1.3 **Vrstva View Model**

Vrstva View Model se stará o interakční logiku GUI v architektuře MVVM. Pro každé view aplikace autor vytvořil odpovídající view model, obsahující veřejné vlastnosti, které view využívá s pomocí data bindingu, k zobrazování stavu aplikace nebo k posílání příkazů. Všechny view modely implementují abstraktní třídu ViewModelBase[21], která implementuje interface INotifyPropertyChanged a obstarává notifikaci view o změně hodnoty vlastnosti view modelu. Každý view model má vytvořen odpovídající interface a tyto interface jsou zděděné z interface IViewModel, které slouží pro jednoduchou identifikaci view modelu. Zjednodušený UML diagram této vrstvy je na obrázku č. 13.



Obrázek č. 13 – Zjednodušený UML diagram vrstvy "ViewModel"[22]

[21] Kód k implementaci této třídy byl získán z webové stránky https://stackoverflow.com/questions/36149863/how-to-write-a-viewmodelbase-in-mvvm

[22] Vlastní zdroj

4.1.4 **Komunikace mezi View Modely**

Pro komunikaci mezi jednotlivými View Modely autor zvolil přístup event agregátoru. Event agregátor je třída, která používá návrhový vzor observer pro příjem zpráv o různých událostech. Tyto zprávy předává všem odběratelům, které daná událost zajímá.

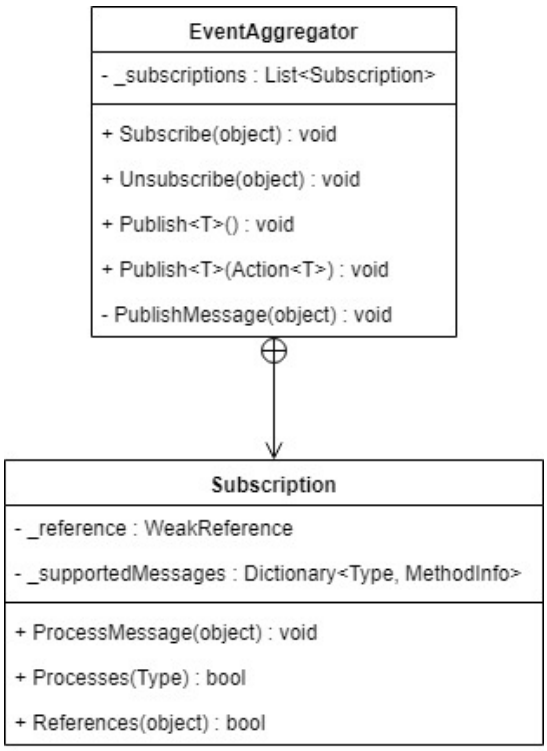
4.1.4.1 EventAggregator

EventAggregator je hlavní třídou obstaravající logiku zasílání zpráv. Obsahuje veřejné metody Subscribe a Unsubscribe, které se starají o přidávání/odebírání odběratelů. Dále obsahuje dvě polymorfní metody Publish, které se používají k vytvoření a poslání zprávy. Je možné zaslat prázdnou zprávu nebo zprávu inicializovat s pomocí Action<T> delegáta.

EventAggregator obsahuje vnořenou privátní třídu Subscription, kterou používá pro uchovávání informací o odběratelích. Tato třída si uchovává weak referenci[23] na odběratele a dále obsahuje slovník, kde klíč je typ odesílané zprávy a hodnotou je MethodInfo objekt, jež obsahuje metadata metody, která daný typ zprávy přijímá.

Třída dále obsahuje metodu ProcessMessage, která zasílá zprávu zavoláním metody Invoke nad objektem MethodInfo korespondujícím s typem zprávy, který je uložen ve slovníku _subscriptions.

Metoda Processes identifikuje, zda odběratel přijímá zprávu daného typu a metoda References vrací, zda daná instance Subscription obsahuje referenci na odběratele.



Obrázek č. 14 – UML diagram třídy EventAggregator[24]

[23] Weak reference je reference na objekt, která umožňuje, aby byl objekt uvolněn garbage collectorem z paměti, což u běžné reference na objekt není možné. (Troelsen, a další, 2017)

[24] Vlastní zdroj

4.1.4.2 Interface IListenTo

Interface IListenTo se v autorově implementaci event agregátoru používá k notifikaci, že daná třída naslouchá nějaké události. Za pomoci modifikace chování jednotlivých kontejnerů při vytváření objektů, třída implementující toto interface je automaticky zařazena na seznam odběratelů event agregátoru. Modifikace chování jednotlivých kontejnerů jsou popsány v kapitole 4.2.

```
public interface IListenTo
{
}
```

K zajištění toho, že bude přijímat zprávy pouze určitého druhu, bylo použita i generická verze tohoto interface, která zabezpečuje, že odběratel zprávy implementuje metodu ListenTo, které za parametr přijímá zprávu daného typu.

```
public interface IListenTo<TMessage> : IListenTo where TMessage : IMessage
{
    void ListenTo(TMessage msg);
}
```

4.1.4.3 Implementace zpráv

Pro zjednodušení zasílání zpráv se autor rozhodl využít tzv. proxy objektů[25], vytvořených pouze na základě interface. Vytváření těchto objektů je zajištěno z pomocí statické třídy ProxyFactory, která využívá tříd z knihovny Castle.Core.

Třída ProxyFactory poskytuje dvě polymorfní metody Generate umožňující generování proxy objektů a jejich následnou inicializaci. Je možno vygenerovat proxy objekt neobsahující žádná pole nebo obsahující pole, která jsou inicializovány s použitím Action<T> delegáta.

```
public static T Generate<T>() where T : class
{
    var p = _generator.CreateInterfaceProxyWithoutTarget<T>(new Interceptor());
    return p;
}

public static T Generate<T>(Action<T> lambda) where T : class
{
    var p = Generate<T>();
    lambda(p);
    return p;
}
```

S použitím ProxyFactory je tedy možné poslat zprávu s použitím následujícího syntaxe:

```
var proxy = ProxyFactory.Generate<IMessage>(x => x.Somevalue = value);
```

[25] Proxy objekty umožňují zachytit volání na původní objekt s tím, že mohou modifikovat nebo rozšířit funkce původního objektu. (Verhas, 2016)

4.1.5 **Abstrakní factory**

Pro demonstraci možností kontejnerů se autor rozhodl do aplikace implementovat návrhový vzor abstraktní factory, které celá řada kontejnerů dokáže automaticky vytvářet jako proxy objekty, vytvořené na základě interface.

V aplikace jsou implementovány dvě abstraktní factory, které se v aplikaci používají pro vytváření view a některých view modelů.

Pro vytvoření těchto factories autor použil interface IViewFactory a IViewModelFactory, které obsahují pouze signaturu vytvářecích metod. O samotné implementace těchto factories se starají jednotlivé kontejnery. V následujícím kódu je ukázka IViewFactory.

```
public interface IViewFactory
{
    INewItemUserControl CreateNewItemUserControl(INewItemViewModel viewModel);
    IItemSearchUserControl CreateItemSearchUserControl(
        IItemSearchViewModel viewModel);
    IDataBrowserUserControl CreateDataBrowserUserControl(
        IDataBrowserViewModel viewModel);
    IMainWindow CrateMainWindow(IMainWindowViewModel viewModel);
}
```

Pro vytvoření view poté stačí zavolat odpovídající metodu na proxy objektu vytvořeném kontejnerem, což je možno vidět na ukázce kódu z třídy MainWindow.

```
CurrentView = _viewFactory.CreateNewItemUserControl(_newItemViewModel);
```

Tento přístup nevyžaduje vytvářet implementace těchto factories a v třídě MainWindow umožňuje dynamicky měnit zobrazení user controlů použitých hlavním okně aplikace, aniž by bylo třeba mít všechny použité user controly uloženy v proměnné třídy.

4.1.5.1 MainWindowCreator

Třída MainWindowCreator se v aplikaci používá k vytvoření a zobrazení hlavního okna aplikace. Autorovým záměrem u této třídy bylo ukázat jakým způsobem je možné vytvářet okna prakticky z jakéhokoliv místa v aplikaci.

V konstruktoru jsou třídě předány proxy objekty obou abstraktních factories. Metoda GetMainWindow se prostřednictvím těchto factories postará o vytvoření samotného okna.

```
public MainWindowCreator(IViewFactory viewFactory,
                         IViewModelFactory viewModelFactory)
{
    _viewModelFactory = viewModelFactory ??
           throw new ArgumentNullException(nameof(viewModelFactory));
    _viewFactory = viewFactory ??
           throw new ArgumentNullException(nameof(viewFactory));
}

public Window GetMainWindow()
{
    var vm = _viewModelFactory.CreateMainWindowViewModel();
    return _viewFactory.CreateMainWindow(vm);
}
```

4.2 Implementace aplikace za použití různých IoC/DI kontejnerů

V této kapitole autor popisuje, jakým způsobem implementoval aplikaci s použitím různých IoC/DI kontejnerů. Pro implementace aplikací autor vybral kontejnery Ninject, Windsor a Unity. Původním záměrem autora bylo použít i kontejner AutoFac, který bohužel, vzhledem k nedostatečným možnostem pro rozšíření kontejneru, nebyl vhodným kontejnerem pro implementaci aplikace. V implementacích aplikací je využíváno konstruktorového vkládání závislostí, kromě několika výjimek popsaných níže.

Pro implementaci některých funkcionalit v aplikaci bylo zapotřebí rozšířit kontejnery o specifické funkce nad rámec standardních funkcionalit jednotlivých kontejnerů. Autor tím chtěl ukázat, že je možné přizpůsobit chování kontejneru pro zjednodušení práce při implementaci některých technik, které se používají při vývoji software.

První rozšíření pomáhá automaticky vložit view model předaný pomocí konstruktoru view do vlastnosti view DataContext, která se používá pro data-binding[26]. Jedná se vložení závislosti do vlastnosti třídy, které bylo za pomocí tohoto rozšíření realizováno konstruktorovým vložením závislosti, bez nutnosti použití kódu na pozadí.

Další rozšíření se týká zasílání zpráv o stavu aplikace mezi view modely pomocí event agregátoru. Toto rozšíření pomáhá v automatické registraci nebo odstranění odběratele event agregátoru, kde odběrateli, pro automatickou registraci stačí implementovat interface IListenTo<IMessage>. V tomto případě se jedná o vložení závislosti do metody Subscribe nebo Unsubscribe, které chtěl autor tímto způsobem demonstrovat.

Kapitola obsahuje rozbor těchto rozšíření, pro jednotlivé kontejnery.

4.2.1 Příprava aplikace pro použití s kontejnery

Pro jednotlivé implementace za použití vybraných kontejnerů, bylo zapotřebí upravit výchozí bod aplikace v souboru app.xaml, protože při vytvoření WPF projektu se jako výchozí bod aplikace automaticky nastaví spuštění hlavního okna aplikace. Tento přístup není žádoucí v případě použití kontejnerů, protože kontejner by měl být co nejblíže vstupnímu bodu aplikace a měl by tedy být jedním z prvních objektů, které se v aplikaci inicializují.

Změnu vstupního bodu je možno upravit změnou atributu StartupUri na atribut Startup v souboru app.xaml. Jako hodnota tohoto atributu se následně nastaví název metody ze souboru app.xaml.cs, kde se nachází vstupní bod aplikace. Pro implementaci autor vytvořil metodu ApplicationStartup v souboru app.xaml.cs.

```
private void ApplicationStartup(object sender, StartupEventArgs e)
{
}
```

Do této metody se poté umístila inicializace kontejneru a samotné vytvoření hlavního okna aplikace.

[26] Data-binding se používá v MVVM pro mapování vlastností view modelu na vlastnosti objektů umístěných ve view, aby view reflektovalo stav view modelu (Microsoft, 2017)

4.2.2 **Implementace pomocí Ninject kontejneru**

Pro správnou funkci aplikace bylo potřeba rozšířit Ninject kontejner o nové funkce. Tato kapitola se zabývá popisem těchto modifikací, konfigurací a následným použitím tohoto kontejneru. Pro implementaci autor používal dokumentaci kontejneru Ninject[27].

4.2.2.1 Rozšíření kontejneru pro event agregátor a MVVM

Rozšíření kontejneru pro event agregátor a MVVM bylo dosaženo pomocí vlastní aktivační strategie, která je aplikována po sestrojení objektu. Byla tedy vytvořena třída CustomActivatioStrategy dědící z třídy ActivationStrategy, která umožňuje překrýt metody Activate a Deactivate. Tyto metody jsou volány při aktivaci nebo deaktivaci objektu, po jeho sestrojení s aktuálním kontextem kontejneru a referencí na aktivovaný nebo deaktivovaný objekt v parametrech volání. Pro zjednodušení implementace se autor rozhodl použít jednu aktivační strategii pro implementaci obou rozšíření.

Pro MVVM autor vytvořil kód v metodě Activate, který zjistí, zda vytvořený objekt je view, poté získá parametry předané do konstruktoru z vlastnosti context.Request.Parameters. Následně identifikuje, který z parametrů je view modelem a tento parametr vloží do vlastnosti DataContext daného view.

```
if (reference.Instance is IView view)
{
    var parameters = context.Request.Parameters;
    foreach (var parameter in parameters)
    {
        var obj = parameter.GetValue(context, context.Request.Target);
        if (obj is IViewModel)
        {
            view.DataContext = obj;
            break;
        }
    }
}
```

Pro event agregátor je implementace rozdělena do metod Activate i Deactivate. Pro správnou funkčnost bylo třeba zajistit, aby aktivační strategie měla přístup k event agregátoru, proto bylo vytvořeno privátní pole, do kterého je třeba event agregátor uložit, před prvním použitím. K tomu se využívá zavolání metody Resolve na kontejneru, který je dostupný přes vlastnost context.Kernel.

V metodě Activate se nejprve zjistí, zda daný objekt implementuje interface IListenTo. Pokud ano, tak se následně zkontroluje, zda má aktivační strategie inicializován event agregátor, a nakonec provede registraci objektu u event agregátoru voláním metody Subscribe.

```
if (reference.Instance is IListenTo)
{
    if (_eventAggregator == null)
    {
        _eventAggregator = context.Kernel.Get<IEventAggregator>();
    }

    _eventAggregator.Subscribe(reference.Instance);
}
```

[27] (Ninject, 2018)

V metodě Deactivate je postup prakticky stejný, s tím rozdílem, že se objekt implementující IListenTo odebírá ze seznamu odběratelů event agregátoru pomocí metody Unsubscribe.

```
if (reference.Instance is IListenTo)
{
    if (_eventAggregator == null)
    {
        _eventAggregator = context.Kernel.Get<IEventAggregator>();
    }

    _eventAggregator.Unsubscribe(reference.Instance);
}
```

Po provedení kódu v metodách Activate i Deactivate je následně zavolána původní metoda předka.

4.2.2.2 Konfigurace kontejneru

Registrace jednotlivých komponent je v Ninject kontejneru prováděna pomocí modulů. Tyto moduly jsou samostatné třídy, které dědí ze třídy NinjectModule a poskytuje abstraktní metodu Load, kterou je potřeba překrýt vlastní implementací. Do této metody se umístí jednotlivé registrace. Ninject container podporuje abstraktní factory z pomocí knihovny Ninject.Extensions.Factory.

```
// registrace IEventAggregatoru, implementuje EvenAggregator, singleton lifestyle
Bind<IEventAggregator>().To<EventAggregator>().InSingletonScope();

// registrace IFileIO, implementuje FileIO, transient lifestyle
// vložený filename do konstruktoru
Bind<IFileIO>().To<FileIO>().InTransientScope()
    .WithConstructorArgument("fileName", Properties.Settings.Default.FileName);

// registrace abstraktní factory
Bind<IViewFactory>().ToFactory();
```

4.2.2.3 Použití kontejneru

Ninject kontejner byl použit vytvořením instance kontejneru. Do kontejneru se následně přidali rozšíření a nahráli se registrace jednotlivých komponent. Nakonec byla zavolána metoda Resolve pro vytvoření objektu MainWindowCreator.

```
_ninjectContainer = new StandardKernel();
_ninjectContainer.Components.Add<IActivationStrategy, CustomActivationStrategy>();
_ninjectContainer.Load(new Bindings());

_mainWindowCreator = _ninjectContainer.Get<MainWindowCreator>();
_mainWindowCreator.GetMainWindow().Show();
```

4.2.3 **Implementace pomocí Windsor kontejneru**

Implementace aplikace pomocí Windsor kontejneru byla provedena za pomoci vytvoření rozšířeních kontejneru. Rozšíření se u Windsor kontejneru považují tzv. Facilities. Pro modifikaci chování kontejneru je zapotřebí vytvořit třídu zděděnou ze třídy AbstractFacility, která poskytuje přístup k jádru kontejneru a obsahuje abstraktní metodu Init, kterou je zapotřebí překrýt vlastní metodou. Popis jednotlivých rozšíření a ukázka konfigurace a použití Windsor kontejneru je předmětem této kapitoly. Pro implementaci autor používal dokumentaci k Windsor kontejneru.[28]

4.2.3.1 Rozšíření kontejneru pro MVVM

Pro použití MVVM s Windsor kontejnerem byla vytvořena nová facility MvvmFacility. Dále byla vytvořena metoda ComponentModelCreated, jež byla vložena do delegáta ComponentModelCreated, který je spuštěn po vytvoření modelu komponenty, obsahující veškerá metadata o vytvářené komponentě

```
protected override void Init()
{
    Kernel.ComponentModelCreated += ComponentModelCreated;
}

private void ComponentModelCreated(ComponentModel model)
{
    if (model.Services.Any(x => typeof(IView).IsAssignableFrom(x)))
    {
        model.CustomComponentActivator = typeof(MvvmActivator);
    }
}
```

. Třída ComponentModel má v kolekci Services uložena veškerá interface, které daná komponenta implementuje a toho autor využil k identifikování, zda třída implementuje interface IView. Pokud se jedná o view, vloží se do CustomComponentActivator, což je jedna z vlastností třídy ComponentModel, vlastní aktivátor komponenty.

Vlastní aktivátor komponenty, umožňuje modifikovat vytváření instance objektu překrytím metody CreateInstance, která nejprve sestrojí instanci objektu, voláním překryté metody na předkovi, následně vyhledá v konstruktoru parametr IViewModel a vloží tento parametr do vlastnosti DataContext daného view.

```
protected override object CreateInstance(
       CreationContext context,
       ConstructorCandidate constructor,
       object[] arguments)
{
    var component = base.CreateInstance(context, constructor, arguments) as IView;
    if (component != null && arguments != null)
    {
        var viewModel = arguments.FirstOrDefault(x => x is IViewModel);
        component.DataContext = viewModel;
        return component;
    }

    return component;
}
```

[28] (Castle Project, 2017)

4.2.3.2 Rozšíření kontejneru pro event agregátor

V tomto případě byla opět vytvořena nová facility MessagingFacility. Dále byly vytvořeny dvě metody ComponentCreated a ComponentDestroyed, které byly vloženy do stejnojmenných delegátů objektu Kernel. Tito delegáti jsou zavoláni po sestrojení/uvolnění objektu.

```
protected override void Init()
{
    Kernel.ComponentCreated += ComponentCreated;
    Kernel.ComponentDestroyed += ComponentDestroyed;
}
```

Metoda ComponentCreated je delegátem zavolána, po sestrojení objektu. Tato metoda zkontroluje, zda právě vytvořený objekt implementuje interface IListenTo a jedná se tedy o odběratele, kterého je třeba registrovat. Dále zkontroluje, zda facility má uloženou instanci event agregátoru v privátním poli a následně provede registraci odběratele.

```
private void ComponentCreated(ComponentModel model, object instance)
{
    if(!(instance is IListenTo))
    {
        return;
    }

    if (_eventAggregator == null)
    {
        _eventAggregator = Kernel.Resolve<IEventAgregator>();
    }

    _eventAggregator.Subscribe(instance);

}
```

Metoda ComponentDestroyed je delegátem zavolána při uvolnění objektu z paměti, její funkce je prakticky stejná jako u metody ComponentCreated, jediný rozdíl je v tom, že metoda v posledním kroku zruší registraci odběratele u event agregátoru.

```
private void ComponentDestroyed(ComponentModel model, object instance)
{
    if (!(instance is IListenTo))
    {
        return;
    }

    if (_eventAggregator == null)
    {
        _eventAggregator = Kernel.Resolve<IEventAgregator>();
    }

    _eventAggregator.Unsubscribe(instance);
}
```

4.2.3.3 Konfigurace kontejneru

Konfigurace Windsor kontejneru je v aplikaci implementována pomocí tzv. Installeru, což je třída, do které se ukládají registrace jednotlivých komponent, aby byly přehledně na jednom místě. Registrace se provádí zavoláním metody Register na kontejneru, se všemi komponentami, které chceme v kontejneru zaregistrovat v parametru tohoto volání.
V následujícím kódu jsou ukázky parametru volání vybraných registrací.

```
// registrace IEventAggregatoru, implementuje EvenAggregator, singleton lifestyle
Component.For<IEventAggregator>().ImplementedBy<EventAggregator>()
    .LifestyleSingleton(),

// registrace IFileIO, implementuje FileIO, transient lifestyle
// vložený filename do konstruktoru
Component.For<IFileIO>().ImplementedBy<FileIO>()
    .DependsOn(Dependency.OnValue("fileName", Properties.Settings.Default.FileName))
    .LifestyleTransient(),
```

Protože Windsor kontejner podporuje abstraktní factories prostřednictvím TypedFactoryFacility, je možné factories zaregistrovat následujícím způsobem.

```
Component.For<IViewModelFactory>().AsFactory(),
Component.For<IViewFactory>().AsFactory()
```

Windsor umožňuje konfiguraci pomocí xml, kterou autor využil pro registraci IDataProvideru, je tedy možné vyměnit data provider bez nutnosti rekompilace kódu. Registrace komponenty byla vytvořena v konfiguračním souboru aplikace app.config.

Nejprve bylo nutné informovat aplikaci o nové sekci v konfiguračním souboru.

```
<section name="castle"
type="Castle.Windsor.Configuration.AppDomain.CastleSectionHandler, Castle.Windsor"
/>
```

Následně bylo možné provést registraci komponenty.

```
<castle>
  <components>
    <component service="DIAddressBook.DataProviders.IDataProvider, DIAddressBook"
               type="DIAddressBook.DataProviders.XmlDataProvider, DIAddressBook"/>
  </components>
</castle>
```

4.2.3.4 Použití kontejneru

Kontejner je použit vytvořením nové instance kontejneru, přidáním všech použitých rozšíření a registrací a následným zavoláním metody Resolve s MainWindowCreator místo generického argumentu.

```
_container = new WindsorContainer();
_container.AddFacility<MessagingFacility>();
_container.AddFacility<TypedFactoryFacility>();
_container.AddFacility<MvvmFacility>();
_container.Install(new Installer());
// Nahraje z app.configu nastavení pro IDataProvider
_container.Install(Configuration.FromAppConfig());
_mainWindowCreator = _container.Resolve<MainWindowCreator>();
_mainWindowCreator.GetMainWindow().Show();
```

4.2.4 **Implementace pomocí Unity kontejneru**

Implementace aplikace pomocí Unity kontejneru vyžaduje, stejně jako u ostatních kontejnerů modifikaci chování kontejneru pro specifické požadavky aplikace. V této kapitole jsou tyto modifikace popsány, spolu s konfigurací a samotným použitím Unity kontejneru. Pro implementaci autor používal dokumentaci k Unity kontejneru. [29]

4.2.4.1 Sestrojovací strategie kontejneru pro MVVM

Rozšíření kontejneru pro event agregátor spočívá ve vytvoření vlastní strategie pro sestrojování objektů, která je následně přidána do kontejneru jako jeho rozšíření. Autorem byla vytvořena třída MvvmStrategy dědící ze třídy BuilderStrategy, v které byla překryta metoda PostBuildUp. Metoda PostBuildUp je kontejnerem zavolána ve chvíli, kdy je objekt sestrojen. To, že je třída zděděná ze třídy BuilderStrategy, jí umožňuje přístup k aktuálnímu kontextu v procesu sestrojování objektu, a tedy k získání objektu samotného prostřednictvím pole context.Existing.

```
public override void PostBuildUp(ref BuilderContext context)
{
	IView view = context.Existing as IView;
	if (view != null)
	{
		foreach (var propertyOverride in context.Overrides)
		{
			// získání objektu v protected poli pomocí reflexe
			var propertyValue = propertyOverride.GetType()
				.GetField("Value", BindingFlags.NonPublic
					|BindingFlags.Instance)
				.GetValue(propertyOverride);

			// vložení viewmodelu do pole DataContext
			if (propertyValue is IViewModel)
			{
				view.DataContext = propertyValue;
				break;
			}
		}
	}

	// volání původní metody předka
	base.PostBuildUp(ref context);
}
```

Přepsaná metoda se nejprve pokouší konvertovat sestrojený objekt na IView. Pokud je konverze úspěšná a objekt tedy není null, metoda pokračuje procházením pole Overrides, které obsahuje všechny argumenty vložené do konstruktoru. Jednotlivé objekty argumentů jsou bohužel uloženy v protected poli, takže bylo nutné použít reflexi pro získání tohoto objektu.

Následně se zkontroluje, zda je současný argument view modelem. Pokud ano, je tento view model vložen do pole DataContext daného view a cyklus je ukončen. Nakonec se zavolá původní metoda PostBuildUp.

[29] (Unity Project, 2019)

4.2.4.2 Sestrojovací strategie kontejneru pro event agregátor

Rozšíření kontejneru pro snazší použití event agregátoru, byla opětovně vlastní strategie pro sestrojování objektů. Autorem byla vytvořena třída EventAggregatorStrategy, která opět dědí z třídy BuilderStrategy a překrývá metodu PostBuildUp.

```
public class EventAggregatorStrategy : BuilderStrategy
{
    private IEventAgregator _eventAggregator;

    public override void PostBuildUp(ref BuilderContext context)
    {
        // kontrola, zda objekt implementuje IListenTo
        if (context.Existing is IListenTo)
        {
            // kontrola, zda je inicializován event agregátor
            if (_eventAggregator == null)
            {
                _eventAggregator = context.Container.Resolve<IEventAggregator>();
            }
            // zaregistrování nového odběratele
            _eventAggregator.Subscribe(context.Existing);
        }
        // volání původní metody předka
        base.PostBuildUp(ref context);
    }
}
```

Modifikovaná metoda zkontroluje, zda objekt implementuje interface IListenTo popsané v kapitole 4.1.4.2. Dále zkontroluje, zda strategie má inicializován event agregátor a pokud ne, zavolá kontejner, aby strategii dodal instanci event agregátoru[30]. Zaregistruje nového odběratele u event agregátoru zavoláním metody Subscribe a nakonec opět zavolá původní metodu PostBuildUp na svém předkovi.

Strategie je zaregistrována v kontejneru v třídě CustomUnityExtensions, která dědí ze třídy UnityContainerExtension a umožňuje přidávat nové funkcionality do kontejneru.

Kontejner Unity bohužel neobsahuje žádnou strategii pro uvolňování objektů a není tedy možné jakýmkoliv způsobem zajistit, aby při uvolnění odběratele byl tento odběratel odstraněn ze seznamu odběratelů event agregátoru, což může představovat potenciální problém u velkých aplikací.

[30] Event agregátor má nastaven životní styl na singleton, takže v celé aplikaci je vkládána jediná instance této třídy.

4.2.4.3 Implementace abstraktní factory

Unity kontejner neposkytuje funkci pro automatické vytvoření abstraktní factory, které aplikace využívá pro tvorbu view modelů a view. Autor se proto rozhodl vytvořit vlastní implemetaci abstraktní factory formou rozšiřující metody pro kontejner ve statické třídě TypedFactory. Tato implementace využívá dynamických proxy z balíčku knihoven Castle.Core.

Rozšiřující metoda IContainer.RegisterAsTypedFactory<T>() vygeneruje proxy objekt s vloženým interceptorem a zaregistruje instanci tohoto proxy objektu do kontejneru s mapováním na interface, které bylo podkladem pro vytvoření proxy objektu.

```
public static class TypedFactory
{
    private static readonly ProxyGenerator _generator;
    static TypedFactory()
    {
        _generator = new ProxyGenerator();
    }

    public static IUnityContainer RegisterAsTypedFactory<T>(
       this IUnityContainer container, IInstanceLifetimeManager manager = null)
       where T : class
    {
       object obj = _generator.CreateInterfaceProxyWithoutTarget<T>(
              new TypedFactoryInterceptor(container));
       container.RegisterInstance(
              typeof(T), obj, manager ?? new ContainerControlledLifetimeManager());
       return container;
    }
}
```

Interceptor potom definuje chování proxy objektu. V tomto případě získá všechny parametry, jež byly předány proxy objektu konstruktorem. Převede je do formy, které dokáže použít kontejner a následně vrátí objekt, který pro něj sestrojil kontejner.

```
public class TypedFactoryInterceptor : IInterceptor
{
    private readonly IUnityContainer _container;
    public TypedFactoryInterceptor(IUnityContainer container)
    {
        _container = container ??
              throw new ArgumentNullException(nameof(container));
    }
    public void Intercept(IInvocation invocation)
    {
        if (invocation != null)
        {
            var ctorParams = invocation.Method.GetParameters();
            var paramOverride = new ResolverOverride[ctorParams.Length];
            for (int i = 0; i < ctorParams.Length; i++)
            {
                paramOverride[i] = new ParameterOverride(
                                       ctorParams[i].Name, invocation.Arguments[i]);
            }
            invocation.ReturnValue =
                _container.Resolve(invocation.Method.ReturnType, paramOverride);
        }
    }
}
```

4.2.4.4 Vložení vytvořených strategií do rozšíření kontejneru

Implementace rozšíření kontejneru je realizována pomocí třídy CustomUnityExtensions, která je zděděná ze třídy UnityContainerExtensions. V této třídě se za pomocí překrytí metody Initialize přidají jednotlivé strategie vytváření objektu.

```
public class CustomUnityExtensions : UnityContainerExtension
{
    private MvvmStrategy mvvmStrategy = new MvvmStrategy();
    private EventAggregatorStrategy eventAggregatorStrategy =
                new EventAggregatorStrategy();

    protected override void Initialize()
    {
        Context.Strategies.Add(mvvmStrategy, UnityBuildStage.Initialization);
        Context.Strategies.Add(
            eventAggregatorStrategy, UnityBuildStage.Initialization);
    }
}
```

4.2.4.5 Konfigurace Kontejneru

Konfigurace kontejneru je v této implementaci provedena prostřednictvím modulu, který se chová jako rozšíření pro kontejner. Autor zvolil tento přístup pro zpřehlednění implementace pomocí tohoto kontejneru. V následujícím kódu je ukázka některých registrací realizovaných v tomto modulu.

```
// registrace abstraktních factories
Container.RegisterAsTypedFactory<IViewModelFactory>();
Container.RegisterAsTypedFactory<IViewFactory>();

// registrace IEventAggregatoru, implementuje EvenAggregator, singleton lifestyle
Container.RegisterType<IEventAggregator, EventAggregator>(
    new ContainerControlledLifetimeManager());

// registrace IFileIO, implementuje FileIO, transient lifestyle
// vložený filename do konstruktoru
Container.RegisterType<IFileIO, FileIO>(
    new ContainerControlledTransientManager(),
    new InjectionConstructor(new object[]{Properties.Settings.Default.FileName}));
```

4.2.4.6 Použití kontejneru

Pro použití Unity kontejneru stačí vytvořit novou instanci kontejneru, přidat do kontejneru dříve vytvořená rozšíření a modul registrací. Dále již stačí na kontejneru zavolat metodu Resolve, pro vytvoření objektu MainWindowCreator.

```
public void ApplicationStartup(object sender, StartupEventArgs e)
{
    _container = new UnityContainer();
    _container.AddNewExtension<CustomUnityExtensions>();
    _container.AddNewExtension<Registration>();

    var creator = _container.Resolve<MainWindowCreator>();
    creator.GetMainWindow().Show();
}
```

5 Výsledky a diskuse

V této kapitole jsou porovnány jednotlivé implementace aplikace z pomocí vybraných kontejnerů a autorem vytvořený benchmark vybraných kontejnerů. Závěrem kapitoly autor hodnotí výsledky získané v rámci praktické části této práce.

5.1 Srovnání jednotlivých implementací

Jednotlivé implementace aplikace za pomoci různých IoC/Di kontejnerů shledal autor práce jako velice podobné.

Základní registrace komponent u jednotlivých kontejnerů byly, prakticky stejné pro všechny kontejnery. U každého kontejneru byl vytvořen konfigurační modul, který byl následně do kontejneru nahrán. Vyskytovali se tu pouze malé rozdíly v zápisu registrací, které vyplývali z rozdílných API kontejnerů.

Velký rozdíl v implementaci autor nalezl u kontejneru Unity, při registraci abstraktních factories, které kontejner Unity nepodporuje a autor byl tedy nucen vytvořit vlastní implementaci tvorby proxy factories implementovanou jako rozšiřující metodu pro kontejner.

Samotné použití kontejnerů probíhalo také prakticky stejným způsobem. Vytvořil se objekt kontejneru, do něhož se nahráli rozšíření a registrace a následně byl sestrojen výchozí objekt aplikace.

Při implementaci rozšíření kontejnerů pro event agregátor, autor nalezl velkou odlišnosti v implementaci u kontejneru Unity, jehož API neumožňuje modifikovat chování tohoto kontejneru pro objekty, které mají být uvolněny z paměti a tyto objekty tedy nemohou být odebrány ze seznamu odběratelů event agregátoru.

Rozšíření kontejneru pro MVVM bylo u kontejnerů implementováno opět podobným způsobem, kde se upravila původní aktivační strategie objektu, aby při aktivaci view vložila do vlastnosti DataContext view model předaný konstruktorem.

U implementace pro Unity kontejner ale musela být použita reflexe pro přístup k view modelu předaného konstruktorem, protože byl v kolekci argumentů konstruktoru uložen v protected poli.

U Windsor kontejneru nebyla vlastní aktivační strategie aplikována na všechny objekty aplikace, protože Windsor kontejner umožňuje vložit vlastní aktivační strategii do komponentového modelu sestrojovaného objektu a ta je potom aplikována pouze na tyto objekty.

5.2 Benchmark kontejnerů

Autor se rozhodl porovnat jednotlivé kontejnery i na základě měření výkonu jednotlivých kontejnerů z hlediska rychlosti a náročnosti na paměť. Pro vytvoření benchmarku autor zvolil knihovnu BenchmarkDotNet.

Samotný benchmark probíhal na základě dvou autorem vytvořených testovacích scénářů, při kterých se prováděl jednoduchý algoritmus spočívají v sestrojování objektových grafů objektu ItemSearchViewModel.

Tyto scénáře se odlišovali v nastavení životních cyklů objektu. První scénář měl všechny sestrojované objekty nastavené na životní styl transient a druhý využíval objektů s životním stylem singleton. V kapitole 3.7.4 jsou tyto životní styly popsány. Autor se rozhodl vybrat tyto životní styly, aby ověřil efektivitu životního stylu singleton.

Každý scénář benchmarku byl spuštěn pro každý kontejner několikrát se vstupním parametrem, čítajícím počet sestrojovaných objektových grafů.

- Pro scénář s transient objekty byly autorem zvoleny parametry 125, 250, 500 a 1000 sestrojovaných objektových grafů.
- Pro scénář se singleton objekty autor zvolil parametry 12500, 25000, 50000 a 100000 sestrojovaných objektových grafů.

V každém z těchto 24 testů, bylo provedeno 100 měření a získávali se údaje o průměrné době trvání algoritmu v milisekundách a celková alokovaná paměť potřebná pro provedení testu v MB.

Benchmark byl proveden na počítači s procesorem Intel Core i7 4770K, 16GB RAM na 64bitových Windows 10.

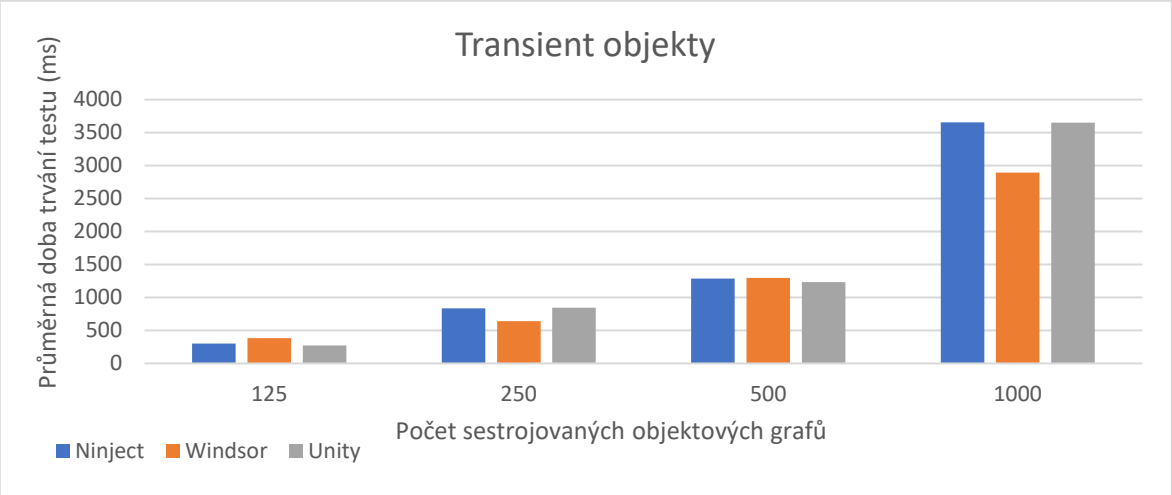
5.2.1 **Výsledky Benchmarku**

5.2.1.1 Transient objekty

Průměrnou dobu trvání sestrojování objektového grafu v ms pro testovací scénář, který používá transient objektů lze vidět v tabulce č. 1 a dále na obrázku č.15 tvořeného grafem.

počet sestrojovaných objektových grafů	Ninject	Windsor	Unity
125	302,687	381,720	273,714
250	833,183	639,898	845,020
500	1287,815	1295,249	1233,494
1000	3655,353	2891,294	3649,935

Tabulka č. 1 – Průměrná doba sestrojení objektových grafů (transient objekty)



Obrázek č. 15 – Graf průměrné doby sestrojení objektových grafů (transient objekty)

Rozborem tabulky a grafu je možné zjistit, že vytváření transient objektů je pro kontejnery časově náročná operace. Je možné vidět, že u kontejnerů došlo ke skokovému navýšení průměrné doby měření při 1000 objektových grafech, přičemž v tomto případě byl kontejner Windsor o více než 0,7s rychlejší než zbylé kontejnery.

Výsledky nákladů na paměť v MB je možno vidět v tabulce č. 2.

počet sestrojovaných objektových grafů	Ninject	Windsor	Unity
125	14,587	12,635	11,046
250	28,761	24,953	21,914
500	56,941	49,503	43,581
1000	122,001	98,668	95,503

Tabulka č. 2 – Paměťové nároky (transient objekty)

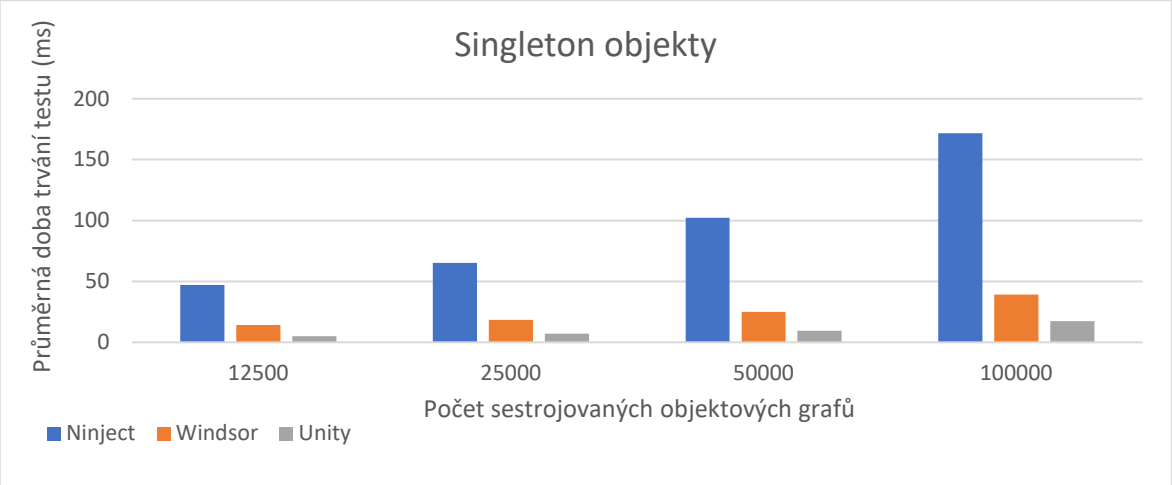
Z tabulky je možné zjistit, že transient objekty zvyšují náročnost i na paměťový prostor aplikace. Testy potvrdily, že s lineárním zvyšováním počtu objektových grafů se i lineárně zvyšují nároky na paměť.

5.2.2 **Singleton Objekty**

V tomto případě testovací scénář sestrojoval objektové grafy s životním stylem singleton. Výsledky průměrné doby sestrojení objektového grafu v ms je možno vidět v tabulce č. 3 a na obrázku č.16 tvořeného grafem.

počet sestrojovaných objektových grafů	Ninject	Windsor	Unity
12500	46,959	14,303	5,150
25000	65,288	18,415	7,036
50000	102,283	25,076	9,626
100000	171,762	39,263	17,434

Tabulka č. 3 – Průměrná doba sestrojení objektových grafů (singleton objekty)



Obrázek č. 16 – Graf průměrné doby sestrojení objektových grafů (singleton objekty)

Z výsledků testů je vidět, že objekty s životním stylem singleton jsou opravdu mnohonásobně efektivnější z hlediska rychlosti sestrojení objektových grafů, čímž se potvrdil autorův předpoklad ze zadání benchmarku. Ačkoliv autor stonásobně navýšil vstupní parametry, sestrojování bylo mnohem rychlejší než v případě životního stylu transient.

Z hlediska paměťových nároků v MB je situace obdobná, jak je vidět v tabulce č.4.

počet sestrojovaných objektových grafů	Ninject	Windsor	Unity
12500	8,458	2,673	0,915
25000	16,249	4,939	1,627
50000	31,891	9,449	3,059
100000	63,269	18,483	5,944

Tabulka č. 4 – Paměťové nároky (singleton objekty)

Tyto výsledky znamenají, že objekty s životním stylem singleton jsou opravdu efektivní a měli by se používat, kdykoliv je to možné, jak je popsáno v kapitole č. 3.7.4.

5.3 Zhodnocení jednotlivých kontejnerů

Před vypracováváním této práce se autor rozhodl, zhodnotit autorem vybrané kontejnery. Je nutné podotknout, že autor již nějakou dobu používá při vývoji software kontejner Windsor, ale v hodnocení se snaží být nestranný.

V porovnání jednotlivých implementací, se autor nesetkal s velkými odlišnostmi, z hlediska implementace. Zřetelné rozdíly byly v syntaxi, vycházející z rozdílných API vybraných kontejnerů. Autora velice překvapil kontejner Ninject, jehož syntax, byla, oproti ostatním kontejnerům, velice intuitivní a kontejner proto poskytoval nejlépe čitelný zápis kódu. Po něm následoval kontejner Windsor, protože syntax kontejneru Unity byla nejhůře čitelná.

Z hlediska funkcí jednotlivých kontejnerů, které autor využil, při implementaci aplikace, byl autor zklamán funkcemi kontejneru Unity, který neposkytoval možnost vytváření abstraktních factories a rovněž neumožňoval upravit chování kontejneru před uvolněním objektu z paměti. Kontejnery Ninject a Windsor, byly v tomto ohledu na stejné úrovni.

Autor by chtěl u kontejnerů porovnat i dokumentaci jednotlivých kontejnerů, protože se jedná o velice důležitou věc, při vyvíjení software. Kontejnery Windsor a Ninject, mají podle autora velmi dobře vypracovanou dokumentaci a autorovi nedělalo veliké problémy najít to co potřeboval, pro úspěšnou implementaci. U kontejneru Unity je dokumentace ve velice špatném stavu a autor byl často nucen získávat informace o jednotlivých třídách kontejneru za pomoci debugování zdrojového kódu, aby byl schopen implementaci z pomocí tohoto kontejneru dokončit.

Výsledky benchmarku jednotlivých kontejnerů ukázaly slabinu kontejneru Ninject, který v těchto testech dopadl nejhůř ze všech kontejnerů. U sestrojování objektových grafů pomocí objektů s životním stylem transient byla doba trvání sestrojení ještě srovnatelná s kontejnerem Unity, ale měl v porovnání s tímto kontejnerem přibližně o třetinu vyšší nároky na paměť. U objektů s životním stylem singleton byly čas i nároky na paměť tohoto kontejneru mnohonásobně vyšší než u ostatních kontejnerů a autor byl tímto výsledkem velice překvapen. U ostatních kontejnerů, kontejner Windsor dopadl velice dobře u transient objektů, ale v případě objektů s životním stylem singleton jasně dominoval kontejner Unity.

Zhodnocením jednotlivých výsledků by autor doporučil vhodný kontejner v závislosti na charakteru vyvíjené aplikace. Pokud se nejedná o příliš velkou aplikaci autor by doporučil kontejner Ninject, protože se mu s ním pracovalo nejlépe. V případě rozsáhlejších aplikací by autor doporučil spíše kontejner Windsor.

6 Závěr

Práce byla koncipována tak, aby demonstroval možnosti, které vývojářům přináší návrhový vzor vkládání závislostí a použití IoC/DI kontejnerů.

V první části, byl představen koncept návrhové vzoru vkládání závislostí a na jakých dalších principech je tento koncept založen. Byly probrány jeho výhody, nevýhody a ukázány způsoby, kterým je možno vkládání závislostí aplikovat. Následně byly představeny IoC/DI kontejnery, které značně usnadňuji aplikaci vkládání závislostí. U kontejnerů byly opět nastíněny možnosti, které s sebou přináší jejich využití a následně byly ukázány způsoby, kterými je možné kontejnery používat.

V druhé části práce, byly poznatky z první části využity k implementaci knihovny obsahující základ aplikace, která umožňuje spravovat kontakty, které pomocí serializace dat do formátu XML nebo JSON. Jednotlivé kontakty je možno přidávat, upravovat a mazat. A je implementována i funkce vyhledávání podle příjmení nebo telefonního čísla. Tato knihovna se stala základem, pro implementaci aplikací využívajících kontejnerů Ninject, Windsor a Unity. Na jednotlivých implementacích byly ukázány možnosti těchto kontejnerů a také jakým způsobem je možné kontejnery rozšířit, aby umožnili zjednodušení práce vývojářů. Jednotlivé implementace byly vzájemně porovnány. Následně byl proveden benchmark rychlosti a paměťových nároků s pomocí knihovny BenchmarkDotNet. Byly vytvořeny dva testovací scénáře, při kterých byly sestrojovány objektové grafy s použitím životních stylů singleton a transient. Benchmark potvrdil autorův předpoklad, že životní styl singleton je mnohonásobně efektivnější než druhý zmiňovaný. Nakonec bylo provedeno srovnání jednotlivých kontejnerů podle syntaxe, dokumentace, funkcí a výkonnostních parametrů.

Výsledkem praktické části jsou funkční aplikace, které demonstrují možnosti vkládání závislostí a jednotlivých kontejnerů. Knihovna pro tyto aplikace obsahuje komponenty, které je možné využít i v jiných aplikacích.

7 Seznam použitých zdrojů

Autofac Contributors. 2018. Autofac Documentation. *autofaccn.readthedocs.io.* [Online] 13. 12 2018. [Citace: 21. 12 2018.] https://autofaccn.readthedocs.io/en/latest/index.html.

Baharestani, Daniel. 2013. *Mastering Ninject for Dependency Injection.* Birmigham : Packt Publishing Ltd., 2013. 978-1-78216-620-7.

Betts, Dominik, a další. 2013. *Dependency Injection With Unity.* Redmond : Microsoft Developer Guidance, 2013. 978-1-62114-028-3.

Castle Project. 2017. Castle Windsor Documentation. *github.com.* [Online] 8. 9 2017. [Citace: 31. 1 2019.] https://github.com/castleproject/Windsor/blob/master/docs/README.md.

Fowler, Martin. 2004. Inversion of Control Containers and the Dependency Injection pattern. *martinfowler.com.* [Online] 23. 1 2004. [Citace: 18. 12 2018.] https://www.martinfowler.com/articles/injection.html.

Gamma, Erich, a další. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software.* místo neznámé : Addison-Wesley, 1995. 978-0-2016-3361-0.

Hordějčuk, Vojtěch. Vkládání závislostí . *voho.eu.* [Online] [Citace: 6. 1 2019.] http://voho.eu/wiki/vkladani-zavislosti/.

Chodounský, Jakub. 2017. Garbage collection in C#. *Chodounsky.net.* [Online] 2. 5 2017. [Citace: 19. 2 2019.] https://chodounsky.net/2017/05/03/garbage-collection-in-c-sharp/.

Landwerth, Immo. 2015. The Future of Unity. *devblogs.microsoft.com.* [Online] 21. 8 2015. [Citace: 10. 2 2019.] https://devblogs.microsoft.com/dotnet/the-future-of-unity/.

Microsoft. 2018. .NET Framework Guide. [Online] 4. 10 2018. [Citace: 20. 10 2018.] https://docs.microsoft.com/en-us/dotnet/framework/index.

—. 2017. Data Binding Overview. *docs.microsoft.com.* [Online] 30. 3 2017. [Citace: 8. 3 2019.] https://docs.microsoft.com/en-us/dotnet/framework/wpf/data/data-binding-overview.

—. 2018. Get started with the .NET Framework. [Online] 4. 10 2018. [Citace: 20. 10 2018.] https://docs.microsoft.com/en-us/dotnet/framework/get-started/.

Ninject. 2018. Ninject wiki. *github.com.* [Online] 5. 5 2018. [Citace: 5. 1 2019.] https://github.com/ninject/ninject/wiki.

Oloruntoba, Samuel. 2015. S.O.L.I.D: The First 5 Principles of Object Oriented Design. *scotch.io.* [Online] 18. 3 2015. [Citace: 20. 12 2018.] https://scotch.io/bar-talk/s-o-l-i-d-the-first-five-principles-of-object-oriented-design.

Prasanna, Dhanji R. 2009. *Dependency Injection.* Greenwich : Manning Publications, 2009. ISBN 978-1-933988-55-9.

Rouse, Margaret. 2005. thread-safe. *whatis.com.* [Online] 1. 9 2005. [Citace: 18. 2 2019.] https://whatis.techtarget.com/definition/thread-safe.

Seeman, Mark. 2011. *Dependency Injection in .NET.* Shelter Island : Manning Publications, 2011. 978-1-9351-8250.

Sexton, Shean. 2011. Editing WPF Code-Behind in Blend. *wpf.2000things.com.* [Online] 29. 4 2011. [Citace: 28. 2 2019.] https://wpf.2000things.com/tag/code-behind/.

StructureMap contributors. 2018. StructureMap documentation. *github.io.* [Online] 12. 07 2018. [Citace: 22. 12 2018.] http://structuremap.github.io/.

Troelsen, Andrew a Japikse, Philip. 2017. *Pro C# 7 With .NET and .NET Core, Eight Edition.* New York : Apress, 2017. 978-1-4842-3017-6.

Unity Project. 2019. Unity Container Documentation. *github.io.* [Online] 2019. [Citace: 12. 1 2019.] https://unitycontainer.github.io/.

Verhas, Peter. 2016. Java Dynamic Proxy: What is a Proxy and How can We Use It. *dzone.com.* [Online] 8. 2 2016. [Citace: 10. 3 2019.] https://dzone.com/articles/java-dynamic-proxy.
What are the downsides to using Dependency Injection? 2011. *stackoverflow.com.* [Online] 14. 10 2011. [Citace: 25. 2 2019.] https://stackoverflow.com/questions/2407540/what-are-the-downsides-to-using-dependency.

8 Přílohy

CD se zdrojovými kódy aplikací a benchmarku