[Vzor: Pevná deska bakalářské práce – **není součástí elektronické verze**]

[Verze 12/2011 platná od 12. 12. 2011 dostupná z http://www.mff.cuni.cz/studium/bcmgr/prace/]

**Univerzita Karlova v Praze**

**Matematicko-fyzikální fakulta**

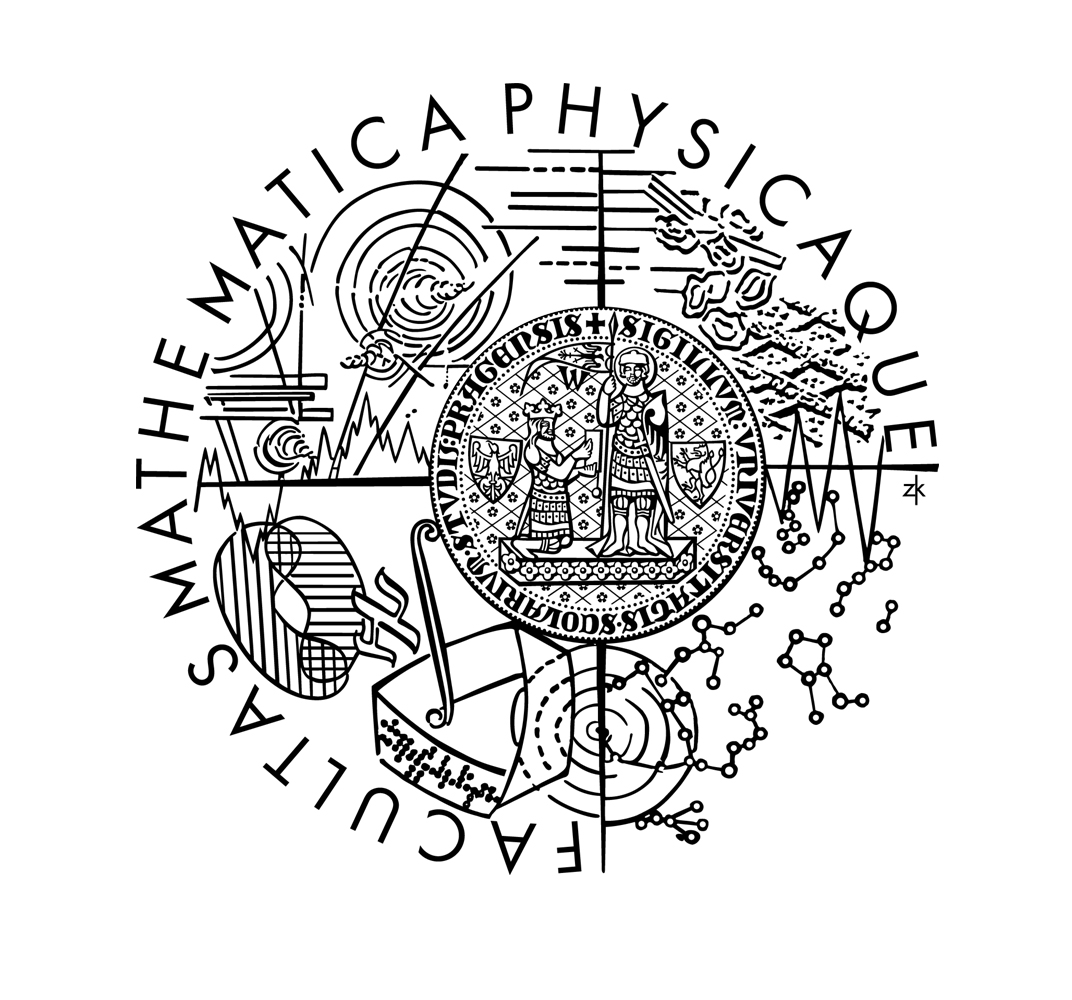
**BAKALÁŘSKÁ PRÁCE**

**Rok Jméno a příjmení autora**

Univerzita Karlova v Praze

Matematicko-fyzikální fakulta

**BAKALÁŘSKÁ PRÁCE**



Miroslav Vodolán

**Editor komponentových architektur pro MEF**

Katedra distribuovaných a spolehlivých systémů

Vedoucí bakalářské práce: Mgr. Pavel Ježek

Studijní program: Informatika

Studijní obor: Programování

Praha 2012

[Vzor: Vevázaný list – kopie podepsaného „Zadání bakalářské práce“. **Toto zadání NENÍ součástí elektronické verze práce NESKENOVAT**.]

[Vzor: Na tomto místě mohou být napsána případná poděkování (vedoucímu práce, konzultantovi, tomu, kdo zapůjčil software, literaturu apod.)]

[Vzor: Strana s čestným prohlášením k bakalářské práci]

Prohlašuji, že jsem tuto bakalářskou práci vypracoval(a) samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona.

V …...... dne............ podpis

Název práce: Editor komponentových architektur pro MEF

Autor: Miroslav Vodolán

Katedra / Ústav: Katedra distribuovaných a spolehlivých systémů

Vedoucí bakalářské práce: Mgr. Pavel Ježek, Katedra distribuovaných a spolehlivých systémů

Abstrakt: Managed Extensibility Framework umožňuje vývoj komponentových aplikací v .NET. Vztahy mezi komponentami však mohou být složité. Pro usnadnění vývoje komponentových aplikací je výhodné tyto vztahy zobrazit a umožnit jejich editaci. Proto jsme vytvořili editor ve formě pluginu pro Microsoft Visual Studio 2010, který umožňuje zobrazení schématu kompozice na základě analýzy zdrojových kódů. V zobrazeném schématu pak poskytuje editace, které se projeví úpravou zdrojových kódů. Možnosti analýzy a nabízených editací jsou dané uživatelskými rozšířeními, které má editor k dispozici. V rámci této práce byl editor naimplementován spolu s rozšířeními, která umožňují jeho použití v projektech aplikací napsaných jazykem C#, kde pomáhá odhalovat chyby v kompozici a usnadňuje změny v komponentové architektuře aplikací.

Klíčová slova: MEF, editor, Visual Studio 2010, komponentové aplikace

Title: MEF Component Architecture Editor

Author: Miroslav Vodolán

Department: Department of Distributed and Dependable Systems

Supervisor: Mgr. Pavel Ježek, Department of Distributed and Dependable Systems

Abstract: [abstrakt v rozsahu 80-200 slov v angličtině; nejedná se však o překlad zadání bakalářské práce]

Keywords: [3 až 5 klíčových slov v angličtině]

Obsah

[Úvod 4](#_Toc323037376)

[Cíle projektu 6](#_Toc323037377)

[MEF podrobněji 7](#_Toc323037378)

[1. Analýza 9](#_Toc323037379)

[1.1 Komponentová architektura s využitím MEF 9](#_Toc323037380)

[1.2 Editace schématu kompozice 9](#_Toc323037381)

[1.3 Analýza aplikace 10](#_Toc323037382)

[1.4 Interpretace metod 13](#_Toc323037383)

[1.5 Typový systém 14](#_Toc323037384)

[1.6 Objektový model 15](#_Toc323037385)

[1.7 Vytváření editací 16](#_Toc323037386)

[1.8 Vykreslování instancí 17](#_Toc323037387)

[2. Rozšiřitelnost Microsoft Visual Studia 2010 19](#_Toc323037388)

[2.1. Projekt VsPackage 19](#_Toc323037389)

[2.2. EnvDTE 19](#_Toc323037390)

[2.3. CodeElement 19](#_Toc323037391)

[2.4. Reakce na uživatelské události 19](#_Toc323037392)

[3. Implementace editoru 20](#_Toc323037393)

[3.1 Struktura 20](#_Toc323037394)

[3.2 Typový systém 21](#_Toc323037395)

[3.2.1 Reprezentace assemblies 21](#_Toc323037396)

[3.2.1.1 UsrAssembly 21](#_Toc323037397)

[3.2.1.2 MSILAssembly 22](#_Toc323037398)

[3.2.1.3 Assembly Runtime 23](#_Toc323037399)

[3.2.1.4 AssemblyLoader 23](#_Toc323037400)

[3.2.2 Reprezentace typů 23](#_Toc323037401)

[3.2.2.1 TypeTicket 23](#_Toc323037402)

[3.2.2.2 TypeDefinition 24](#_Toc323037403)

[3.2.2.3 InternalType 24](#_Toc323037404)

[3.2.2.4 Dědičnost 25](#_Toc323037405)

[3.2.2.5 Generické typy 26](#_Toc323037406)

[3.2.2.6 Metody 26](#_Toc323037407)

[3.2.3 TypesManager 27](#_Toc323037408)

[3.3 Objektový model 28](#_Toc323037409)

[3.3.1 Instance 28](#_Toc323037410)

[3.3.2 Volání metod na instancích 29](#_Toc323037411)

[3.3.3 Vytváření instancí 29](#_Toc323037412)

[3.3.3.1 Přímé vytvoření 30](#_Toc323037413)

[3.3.3.2 Voláním konstruktoru 30](#_Toc323037414)

[3.3.3.3 Injektováním ze zadaných dat 30](#_Toc323037415)

[3.3.4 Sdílené instance 31](#_Toc323037416)

[3.3.6 Koncept dirty instancí 31](#_Toc323037417)

[3.4 Reakce na události vyvolané uživatelem 31](#_Toc323037418)

[3.4.1 Změna aktivního solution 32](#_Toc323037419)

[3.4.2 Změny v projektech aktivního solution 32](#_Toc323037420)

[3.4.3 Změny ve zdrojových kódech 32](#_Toc323037421)

[3.4.3.1 Přidání nového CodeElement 33](#_Toc323037422)

[3.4.3.2 Odebrání CodeElement 33](#_Toc323037423)

[3.4.3.3 Přejmenování CodeElement 33](#_Toc323037424)

[3.4.3.4 Změna těla CodeElement 33](#_Toc323037425)

[3.5 Systém závislostí 34](#_Toc323037426)

[3.5.1 Cíle závislostí 34](#_Toc323037427)

[3.5.1.1 Závislost na CodeElement 34](#_Toc323037428)

[3.5.1.2 Závislost na názvu typu 35](#_Toc323037429)

[3.5.1.3 Závislost na souborech a složkách 35](#_Toc323037430)

[3.5.2 Závislosti instrukcí v IInvokeInfo 35](#_Toc323037431)

[3.5.3 Závislosti stádií reprezentace typu 35](#_Toc323037432)

[3.5.4 Závislosti assemblies 36](#_Toc323037433)

[3.5.5 Závislosti schématu kompozice 36](#_Toc323037434)

[3.6 Editace 36](#_Toc323037435)

[3.6.1 ILanguageDefinition 36](#_Toc323037436)

[3.6.2 Koncept EditTransformation 37](#_Toc323037437)

[3.6.3 Editace poskytované instancí 37](#_Toc323037438)

[3.6.3.1 Accept editace 38](#_Toc323037439)

[3.6.3.2 Remove editace 38](#_Toc323037440)

[3.6.7 Statické editace 38](#_Toc323037441)

[3.7 Komponentový model 38](#_Toc323037442)

[3.7.1 Reprezentace komponent 38](#_Toc323037443)

[3.7.3 Naplnění importů 39](#_Toc323037444)

[3.8 Uživatelské rozhraní 39](#_Toc323037445)

[3.8.1 Seznam dostupných composition point 39](#_Toc323037446)

[3.8.2 Vykreslování schématu kompozice 39](#_Toc323037447)

[3.8.2.1 Vykreslení instance 40](#_Toc323037448)

[3.8.2.2 Určování pozice zobrazených instancí 40](#_Toc323037449)

[3.8.2.3 Importy, exporty a jejich spojení 40](#_Toc323037450)

[3.8.3 Zamykání editací 40](#_Toc323037451)

[4. Uživatelská příručka 42](#_Toc323037452)

[4.1 Popis rozhraní editoru 42](#_Toc323037453)

[4.2 Použití editoru 42](#_Toc323037454)

[4.3 Rozšiřitelnost editoru 42](#_Toc323037455)

[4.3.1 Uživatelské parsery 42](#_Toc323037456)

[4.3.2 Uživatelské interpretery 44](#_Toc323037457)

[4.3.3 Uživatelské definice typů 45](#_Toc323037458)

[4.3.4 Uživatelské zobrazení objektů 48](#_Toc323037459)

[4.3.5 Uživatelské ILanguageDefinition 50](#_Toc323037460)

[4.3.6 Standardní rozšíření 50](#_Toc323037461)

[4.3.7 Doporučená rozšíření 50](#_Toc323037462)

[5. Závěr 51](#_Toc323037463)

[6. Reference 53](#_Toc323037464)

# Úvod

V současné době je při vývoji software kladen důraz na komponentovou architekturu aplikací. Díky rozdělení jednoho velkého problému na několik menších podúloh, získáme možnost vyvíjet aplikace ve větším počtu pracovníků. Navíc, pokud máme nějakou komponentu, která dobře řeší určitou funkčnost, můžeme ji s výhodou využít v několika projektech.

S komponentovou architekturou aplikací souvisí i jejich rozšiřitelnost. Aplikaci postavenou z několika funkčních celků obvykle nebývá problém rozšířit o další komponenty. Za příklad vezměme Microsoft Visual Studio. Díky jeho komponentové architektuře není problém přidat do něj vlastní rozšíření, která například umožní podporu nových programovacích jazyků nebo která přidají nové vývojářské nástroje.

Způsobů jak psát komponentovou aplikaci je několik. Nejjednoduší by se mohlo zdát uvolnění zdrojových kódů všech komponent, ať si je každý kdo je potřebuje použít do svého projektu přidá. To má však svá úskalí. Vývojářům placeného software by se jistě nelíbilo zveřejňování vlastních nápadů v podobě zdrojových kódů. Nevýhodné by to však bylo i z hlediska udržitelnosti takového řešení. Například vydání nové verze komponenty by vedlo ke změnám ve zdrojovém kódu několika projektů.

Jiným možným způsobem je publikace komponent pomocí zkompilovaných knihoven, čímž se odstraní problém se závislostí na jejich zdrojovém kódu. Bylo by ale dobré definovat jednotný způsob, jak takové komponenty psát. V prostředí .NET jsou k tomu účelu vytvořeny Managed Addin Framework (MAF) a Managed Extensibilty Framework (MEF). Oba frameworky řeší problém běhové rozšiřitelnosti, rozdílem ale je to, že MAF se zabývá spíše prací s nahranými rozšířeními, kdežto MEF nabízí propracovanější rozhraní pro vyhledávání rozšíření a definování vztahů mezi nimi. To je také důvodem, proč je tato práce zaměřena právě na koncepci MEF.

Komponentami v prostředí MEF jsou instance tříd, na kterých jsou definovány importy a exporty. Importem rozumíme datovou položku třídy, která očekává objekt nebo objekty vymezené takzvaným kontraktem. Ten může například specifikovat, že do datové položky bude přiřazen objekt splňující rozhraní IContent, což bude uvedeno v příkladu. Na druhé straně, exportem může být jak datová položka, tak celá třída, z níž je získán objekt použitelný pro naplnění nějakého importu.

Ve zdrojovém kódu to pak může vypadat následovně:

using System.ComponentModel.Composition;

namespace Extensions

{

[Export(typeof(ILayout))]

public class NormLayout:ILayout

{

[Import]

public ILogger Logger;

[ImportMany(typeof(IContent))]

public IEnumerable<IContent> Contents;

}

}

Vidíme třídu NormLayout exportovanou dle rozhraní ILayout. Na třídě je definován import objektu splňující rozhraní ILogger a import všech dostupných exportů s kontraktem IContent.

Už tedy víme, jak se komponenty definují. Nyní se podívejme na způsob práce s nimy. Zmiňovali jsme, že MEF umožňuje běhovou rozšiřitelnost aplikací. To znamená, že aplikace musí při svém běhu rozhodnout, jaká rozšíření chce nahrát. MEF ji dává k dispozici několik připravených katalogů, které zajišťují vyhledávání komponent v souborech MSIL knihoven nebo v již spuštěných assemblies. Díky tomu, že si vyhledávání komponent řídí aplikace sama, může jejich nahrávání snadno přizpůsobit uživatelským nastavením, nebo zrovna prováděným akcím.

Samotné komponenty by však neměly význam, kdyby nám chyběla možnost jak naplnit definované importy z dostupných exportů. Tomuto skládání komponent se říká kompozice. Zajišťuje ji třída CompositionContainer, z katalogů, které ji dá aplikace k dispozici. Z nich získá všechny dostupné komponenty a zkouší vyřešit závislosti mezi jejich importy a exporty. Také zaručí, že žádná komponenta nebude nahrána v nekonzistentním stavu – bez naplnění všech importů.

Při kompozici však může dojít k řadě chyb. Chybějící nebo nejednoznačný export pro nějaký import může narušit skládání komponent. Stejně tak nevhodně zvolený kontrakt, který nezajistí typovou shodu importů a exportů vede k vyvolání vyjímky. Ve složitějších aplikacích navíc můžeme ztratit přehled o celkové architektuře, což negativně ovlivňuje další vývoj.

Za účelem ladění MEF kompozice již bylo vyvinuto několik nástrojů. Nejznámějšími jsou Mefx a jeho vizuální podoba Visual MEFX. Oba dva umějí pracovat pouze se zkompilovanými projekty, navíc nenabízejí žádnou grafickou reprezentaci výsledné kompozice, která je jistě mnohem přehlednější, než pouhé čtení zdrojového kódu, jak demonstruje obrázek:

Motivační obrázek srovnávající čitelnost kódu/čitelnost schématu

Náhled na schéma kompozice poskytuje nástroj MEF Visualizer Tool, který však musí být napojen přímo ve zdrojovém kódu a výsledek vrací až po spuštění aplikace. Tyto důvody daly vzniknout projektu na vývoj editoru, který přehledně zobrazí schémata kompozice i ze zdrojových kódů a umožní jejich editaci systémem Drag&Drop. Důležité také je, aby editor upozornil na možné chyby v kompozici a ulehčil tak vývoj aplikace.

# Cíle projektu

V době zadání práce nebyla dostupná žádná vhodná aplikace pro zobrazení komponentové architektury pro knihovnu MEF ze zdrojových kódů rozpracovaného projektu, která by zároveň umožňovala tento kód editovat. Výše zmíněné nástroje se sice zabývají obdobnou problematikou, avšak ani jeden z nich neumožňuje zobrazování schémat ze zdrojových kódů, ani jejich editaci.

Cílém této práce je vývoj editoru, který splní následující kritéria:

* Editor bude integrován do Microsoft Visual Studia 2010
* Umožní provádět analýzu zdrojových kódů rozpracované .NET aplikace otevřené v Microsoft Visual Studiu 2010.
* Na základě provedené analýzy přehledně zobrazí zjištěné schéma kompozice.
* Umožní uživateli v zobrazeném schématu provádět editace
* Dovede reagovat na uživatelské změny zdrojového kódu patřičným překreslením schématu
* Upozorní na případné chyby v kompozici
* Umožní pomocí rozšíření měnit způsob vykreslení schématu kompozice
* Bude rozšiřitelný o schopnosti analýzy zdrojových kódů a MSIL knihoven

Součástí práce také bude několik standardních rozšíření editoru:

* Modul pro parsování jazyka C#
* Modul pro interpretaci sémantického stromu
* Objektový model nutný pro analýzu základních MEF tříd
* Modul pro zobrazení významných MEF objektů

Na začátku práce bude analýza typických použití technologie MEF, dle kterých zjistíme jaké editace je vhodné uživateli nabídnout. Klíčovou částí potom bude návrh a implementace editoru, který tyto editace poskytne uživateli. V závěru práce pak bude uvedena první příručka formou tutoriálu pro uživatele, který chce editor využívat při své práci. A druhá příručka pro uživatele, který píše rozšíření pro editor. Ta bude opět psaná formou tutoriálu, v němž naimplementujeme vzorová rozšíření.

# MEF podrobněji

V úvodu jsme ukázali co jsou to komponenty, jak se definují a také jsme uvedli základní způsob, jakým se s nimi v MEF pracuje. Nyní si podrobněji popíšeme jak probíhá samotná kompozice a představíme si další koncepty, které MEF nabízí.

Již víme, že definované importy jsou při kompozici naplněny z dostupných exportů. Naplnění probíhá tak, že MEF provede přiřazení do datové položky importu, případně použije setter importující property. Přiřazení i volání metod musí probíhat na zkonstruovaných objektech. Přesto však MEF nabízí katalogy, které umožňují přidat do kompozice komponentu pouze na základě typu. MEF tedy musí mít k dispozici konstruktor, kterým takovou komponentu vytvoří. Pokud neuvedeme jinak, je zavolán bezparametrický konstruktor. Navíc, pokud ho MEF nenalezne, skončí kompozice vyvoláním vyjímky.

Abychom mohli sami určit konstruktor, který má být při kompozici zavolán, máme k dispozici koncept importujícího konstruktoru. Jedná se o konstruktor, jehož parametry jsou chápány jako importy. Při kompozici je pak takový konstruktor zavolán s argumenty, získanými z dostupných exportů.

class NormLayout

{

[ImportingConstructor]

public NormLayout(ILogger logger, [ImportMany]IContent[] contents)

{

...

}

}

Ukázka využití ImportingConstructor atributu. Třída NormLayout bude při kompozici vytvořena pomocí konstruktoru s parametry získanými z dostupných exportů.

Zřejmou odlišností importů definovaných v konstruktoru proti importům na datových položkách je nutnost získat je před poskytováním exportů. Komponenta totiž nemůže poskytnout export, dokud není zkonstruována a to může proběhnout až tehdy, když máme k dispozici všechny parametry pro konstruktor. Musíme si tedy dát pozor na cyklické závislosti, které můžou při kompozici snadno vzniknout.



Komponenta ServiceA potřebuje pro volání konstruktoru komponentu ServiceB. Ta však nemůže být zkonstruována, dokud nedostane kompnentu ServiceA.

V úvodu jsme nastínili možnost importovat všechny exporty dostupné pro nějaký kontrakt do jediného importu pomocí atributu ImportMany. Cílový import musí být typu pole, IEnumerable nebo ICollection. Naproti tomu můžeme vytvořit import, který nepřeruší kompozici, pokud pro něj nenajdeme vhodný export. Ve zdrojovém kódu ho označíme následovně:

class NormLayout

{

[Import(AllowDefault=true)]

ILogger logger;

[ImportMany]

IContent[] contents;

...

}

Nebude-li nalezen vhodný export pro datovou položku logger, zůstane v ní defaultní hodnota. Kdybychom nepovolili import s defaultní hodnotou, došlo by k přerušení kompozice s chybovým hlášením.

MEF nenabízí jen rozličné možnosti jak importovat a exportovat data. Umožňuje nám také k těmto datům připojit takzvaná metadata. Do nich se dají uložit například informace o verzi komponenty, které poté můžeme využít pro vyhledání nejnovější dostupné verze komponenty. Využití metadat znázorňuje následující příklad:

Komponenty s různým číslem verze, a najití verze nejnovější

Uvedli jsme nejdůležitější koncepty, které se v MEF používají. K dispozici však máme další možnosti, jak s komponentami pracovat a jak zpřehlednit zápis importů a exportů. Podrobnější informace jsou dostupné na: <http://msdn.microsoft.com/en-us/library/ee155691.aspx> . Pokrytí veškerých možností které MEF nabízí by bylo nad rámec této práce, zabývat se proto budeme pouze výše uvedenými koncepty. Editor však bude navržen s ohledem na snadnou rozšiřitelnost o podporu dalších konceptů.

# 1. Analýza

## 1.1 Komponentová architektura s využitím MEF

Píšeme-li komponentovou aplikaci s využitím MEF, musíme vyřešit problém, jak definovat kontrakty importů a exportů komponent. Mohlo by se zdát, že nejsnazší bude v kontraktu použít třídu, která implementuje požadovanou funkčnost. Při podrobnějším zkoumání ale zjistíme, že toto řešení není příliš vhodné. Pokud známe konkrétní implementaci nějaké funkcionality, nepotřebujeme ji nahrávat pomocí MEF. Lepší by bylo, kdybychom kontrakty omezili jen na specifikovanou funkčnost bez závislosti na implementaci. Toho můžeme dosáhnout tak, že pro každou komponentu definujeme interface, který bude publikován. Interface poté využijeme pro kontrakt v definici importu/exportu.

Závislost pouze na rozhraních komponent nám umožňuje nejen psát přehlednější kód, ale také nám dává možnost získat více implementací pro stejný interface. Příkladem může být různá implementace logování zpráv v aplikaci. Můžeme ho naimplementovat jako jednoduchý výstup do textového souboru a v novější verzi nabídnout logování s grafickým formátováním. Jelikož poskytují stejnou funkčnost, vystačí si se stejným interface. Díky tomu nám stačí naimplementovat jedinou komponentu bez dalších změn v aplikaci.

Jak z výše uvedeného vyplývá, aplikace může být sestavena z množství komponent, které se mohou časem měnit, můžou pocházet z různých zdrojů, nebo dokonce mohou záviset na konkrétních nastaveních uživatele. Zamysleme se tedy, jak by mělo vypadat schéma kompozice, které by přehledně vypovídalo o struktuře aplikace.

Schéma by jistě mělo zobrazit všechny dostupné komponenty, spolu s importy a exporty které definují. Ze schématu také bude muset být patrné, kterou třídou je komponenta implementována a ze kterého katalogu byla získána, což může být důležité pro určení její verze. Samotné zobrazení komponent by však nemělo velký význam, kdybychom neznázornili vztahy mezi nimi, neboť význam kompozice spočívá ve vyhledávání dostupných exportů pro definované importy.

Tato kritéria nám dávají přirozené schéma kompozice, které můžeme znázornit pomocí diagramu, kde spojnice ukazují vztahy mezi importy a exporty. Komponenty budou navíc vnořovány do patřičných katalogů, což umožní aplikaci lépe rozdělit do logických celků.

## 1.2 Editace schématu kompozice

Popsané schéma přehledně zobrazuje komponentovou architekturu aplikace. Položme si však otázku, jaké editace by uživateli pomohly při práci se schématem kompozice.

Předpokládejme že uživatel má vytvořené nějaké komponenty a řeší otázku jejich pospojování.



Při kompozici aplikace musí uživatel zajistit načtení komponent z rozličných katalogů. Pro jejich použití v CompositionContainer navíc musí všechny katalogy vložit do AggregateCatalog.

Uživatel tedy potřebuje do schématu vkládat různé druhy MEF katalogů a měnit jim vlastnosti specifické pro konkrétní katalog. Příkladem může být DirectoryCatalog, u nějž uživatel jistě uvítá možnost měnit cestu pro vyhledávání knihoven, aniž by musel ručně přepisovat zdrojový kód. Další potřebnou operací je změna vztahů mezi komponentami, které docílíme přesunováním komponent mezi kontejnery. Pro snazší orientaci ve zdrojových kódech bude také užitečné, aby se mohl uživatel podívat na místo vzniku editovaných objektů.

Výše popsané editace by bylo možné provádět v externím konfiguračním souboru, který by byl uvnitř aplikace interpretován například při spuštění. Tím bychom ale ztratili možnost kompatibility s již vzniklými projekty, které s žádným konfiguračním souborem nepočítají.

Jinou možností je spolupráce editoru přímo se zdrojovými kódy aplikace. To nám dává výhodu v možnosti nasazení do libovolných projektů bez dodatečných úprav. Je však nutné, aby editor uměl pracovat s různými .NET jazyky. Klíčovovou vlastností editoru proto bude porozumění sémantice zdrojových kódů, jednak proto, aby bylo možné sestrojit schéma kompozice, ale také proto aby na nich mohly být prováděny editace.

## 1.3 Analýza aplikace

Kompoziční schéma aplikace můžeme získat několika způsoby. Implementačně nejsnazší by bylo zkoumání aplikace až po jejím spuštění. Po dobu běhu aplikace bychom sledovali, které komponenty jsou nahrávány. Po skončení aplikace bychom získané výsledky zobrazili uživateli. Tento způsob je však nevhodný z několika důvodů. Především by editoru chyběla interaktivita, neboť spouštění aplikace může trvat poměrně dlouho. Dalším problémem je fakt, že kompozice může být provedena až podle různě složitých požadavků, které by bylo nutné při každém zkoumání stále opakovat. Jistě také zobrazení a editace schématu kompozice může dávat smysl i v době, kdy aplikace není dokončena natolik, aby ji bylo možné vůbec spustit.

Alternativou, která odstraní problém se spouštěním, je interpretování zdrojových kódů bez jejich kompilace. I zde však narazíme na některé problémy, které je nutné vyřešit. Uvažme zda by bylo vhodné interpretovat celou aplikaci od startovní metody, která je jako první zavolána při spuštění. Sice jsme odstranili problém kompilace a spuštění aplikace, ale stále platí, že aplikace nemusí být dostatečně dopracovaná na to, aby bylo možné ze startovní metody rozpoznat kompoziční schéma. Navíc interpretace ze zdrojových kódů, bývá řádově pomalejší než spuštění zkompilovaného programu, interpretaci složitějších aplikací by proto nebylo možné zvládnout v rozumném čase.

Řešením je interpretace pouze těch částí zdrojového kódu, které jsou zajímavé z hlediska kompozice. Nejmenší rozumně interpretovatelnou jednotkou zdrojového kódu je metoda. Definujme tedy composition point jako metodu významnou z hlediska MEF kompozice a uvažme analýzu kódu založenou na spouštění těchto composition point a zkoumání stavu objektů, které se při jejich interpretaci objevily.

U takto získaných objektů můžeme testovat, zda je vhodné je zobrazit uživateli ve schématu kompozice, případně zda se jedná o komponenty, katalogy nebo jiná důležitá MEF primitiva. Navíc si ale můžeme zapamatovat příkazy ve zdrojovém kódu, které se pojí s nějakou významnou akcí. Například naplnění importů komponenty voláním metody CompositionContainer.ComposeParts(). Když bude chtít uživatel zrušit naplnění importů, stačí pouhé smazání uvedeného příkazu.

Spouštění metod nám tedy poskytne dostatečné informace potřebné pro zobrazení a editaci schématu kompozice, zamysleme se ale nad tím, jak poznáme metodu, která je důležitá z hlediska MEF kompozice? Samotné výskyty MEF tříd v metodách ještě nemusí znamenat, že v nich vůbec dochází k nějaké kompozici, naopak metody, kde se z žádnými MEF objekty nepracuje mohou mít na kompozici zásadní význam.

class Composer

{

[Import]

ILogger logger;

DirectoryCatalog \_catalog;

public void makeComposition()

{

loadDir("Extensions");

composeDir();

}

void loadDir(string dir)

{

\_catalog = new DirectoryCatalog(dir);

}

void composeDir()

{

var container = new CompositionContainer(\_catalog);

container.ComposeParts(this);

}

}

Metody loadDir a composeDir sice obsahují MEF objekty, samy o sobě však o kompozici moc nevypovídají. Naopak významná metoda makeComposition, žádný MEF objekt neobsahuje.

Dovolme tedy uživateli, aby sám určil, které metody má editor považovat za composition point. Jedna možnost je nechat uživatele specifikoval seznam významných metod, které bychom si pamatovali v nějakém konfiguračním souboru. Toto řešení by však nebylo přenosné ve zkompilované aplikaci. Navíc bychom se dostali do problémů se synchronizací názvů tříd a metod popisujících, který composition point chceme spustit, neboť názvy se ve zdrojovém kódu mohou často měnit.

Jinou možností je označení composition point atributem, který bude obsahovat jako argumenty vstupní hodnoty parametrů spouštěné metody. S ohledem na styl deklarace importů, exportů a importujících konstruktorů v rámci MEF se autor práce rozhodl pro toto řešení, jehož výhodou je, že se informace o composition point zachová i ve zkompilované aplikaci. I toto řešení má však své nedostatky. Především se jedná o nutnost zasahovat do zdrojového kódu aplikace. Definujme tedy implicitní composition point jako bezparametrický konstruktor komponenty. Je obvyklé, že ke kompozici dochází právě v těchto konstruktorech. Nabídnutím implicitních composition point tedy omezíme uživatelské zásahy do zdrojového kódu na minimum.

Vzhledem k netradičnímu způsobu interpretace, kterým budeme aplikaci analyzovat, je důležité přesně vymezit podmínky, v jakých se composition point spouští. Stanovme tedy, že composition point bude vždy volán na objektu vytvořeném ze třídy, ve které je definován. Tento objekt však musíme nějakým způsobem zkonstruovat. To provedeme pomocí bezparametrického konstruktoru pokud je přítomen, jinak necháme objekt před spuštěním composition point neinicializovaný. Speciálním případem je pak composition point určený na statické metodě. Abychom se co nejvíce přiblížili skutečnému běhu .NET aplikací, budeme simulovat že se jedná o první použití statické třídy. Před spuštěním composition point tedy zavoláme statický konstruktor této řídy.

## 1.4 Interpretace metod

Aby byl editor schopen interpretovat zadaný composition point, musí nejdřívě porozumět jazyku kterým je napsán, stejně tak jako jazykům metod z něj volaných. Metody, které je nutné interpretovat však nemusí pocházet pouze z dostupných zdrojových kódů, ale mohou být implementovány v referencovaných knihovnách, což si žádá podporu pro zpracování MSIL instrukcí.



Z metody makeComposition je volána metoda LoadCatalogs v Project2 psaná ve Visual Basicu a ComposeCatalogs ze zkompilované knihovny, dostupná pouze v MSIL instrukcích.

Spouštění zdrojových kódů interpretováním se obvykle provádí tak, že zdrojový kód nejdříve zpracujeme z hlediska syntaxe, kdy získáme jednotlivá slova příkazů. V další fázi se snažíme zjistit význam těchto slov. Rozhodujeme zda se jedná například o deklaraci proměnné, volání metody,… Tomuto procesu se říká parsování. Jeho výsledkem jsou tzv. syntaktické/sémantické stromy.



Ukázka syntaktického a sémantického stromu, vzniklých parsováním řádku zdrojového kódu.

Vzhledem k tomu, že interpretaci metod budeme muset při analýze zdrojových kódů provádět často, nebylo by vhodné neustále opakovat parsování všech volaných metod, neboť může být časově náročné. Editor si proto bude výsledky parsování pamatovat do té doby, dokud se nezmění zdrojový kód, ze kterého pocházejí.

Rozparsované metody můžeme nadále zpracovat pomocí interpreteru. Ten bude vykonávat získané instrukce tak, aby co nejvíce simulovat běh .NET aplikace. Díky tomu získáme potřebné informace pro vykreslení schématu kompozice a jeho následnou editaci. Budeme však muset nějakým způsobem reprezentovat objekty, neboť nad nimi probíhají všechny důležité operace. Nejsnazší by se mohlo zdát, vytvářet stejné objekty, jako .NET aplikace při svém běhu. Na to bychom však potřebovali .NET reprezentaci všech typů, se kterými bychom pracovali. Tuto reprezentaci ale typicky mít nebudeme, neboť budeme pracovat s nezkompilovanými zdrojovými kódy.

Řešením by se mohlo zdát vytváření .NET typů ze zdrojových kódů pomocí Reflection. Zde však také narazíme na jistá omezení. Vytvořené typy už bychom nemohli snadno například odstraňovat, což pro nás bude důležitá operace, neboť odpovídá odstranění nějaké třídy ze zdrojového kódu.

Z těchto důvodů musíme vytvořit vlastní reprezentace typů a objektů, neboli objektový model, který nám umožní volání metod, dědičnost a další .NET operace nad objekty.

## 1.5 Typový systém

Interpretace metod si žádá objektový model, který by umožnil simulovat práci s .NET objekty. Pro jeho implementaci však potřebujeme typový systém, nad kterým budeme objekty vytvářet.

Rozmysleme si, co všechno budeme od námi vytvářených typů požadovat. Jistě budou muset obsahovat informace o metodách, které můžeme na objektu daného typu zavolat. Metody však budeme uvažovat několika druhů.

* Nevirtuální metody – obvyklé metody, které se na objektu volají podle typu, na který je právě přetypován.
* Virtuální metody – metody, které jsou ovlivněny dědičností a nezávisí u nich na konkrétním přetypování objektu, ale pouze na typu ze kterého byl objekt vytvořen.
* Abstraktní metody – metody, pro které není v místě deklarace dostupná jejich implementace. Díky nim budeme moci reprezentovat typy získané z abstraktních tříd, nebo z interface.
* Statické metody – metody definované na statických třídách – ty existují v jedinné instanci, která je vytvořena až při prvním pokusu o její použití. Statické třídy se chovají jako sdílené objekty, které vytvoříme až při prvním pokusu o vyvolání některé jejich metody.
* Konstruktory – metody, používané ke konstrukci objektů. Mohou být statické i instanční.

Další objektovou operací jsou aritmetické a logické operátory. Ty však také dokážeme implementovat pomocí volání metod, jak ukazuje následující tabulka:

Tabulka ukazující názvy metod použité pro jednotlivé operátory, tak jak jsou překládány do MSIL kodu, např. + op\_Addition

Stejným způsobem dokážeme vyřešit implicitní/explicitní typové konverze. Například implicitní konverzi z typu A na typ B obstará metoda: B op\_Implicit(A).

Na stejném principu funguje i .NET reprezentace operátorů, kterou můžeme vidět v MSIL kódu přeložených aplikací. Obdobou jsou potom properties objektů, které se skládají z metody pro nastavení hodnoty a metody pro získání hodnoty. Pro property s názvem Test jsou vytvořeny metody get\_Test() a set\_Test(). Datové položky jsou v .NET reprezentovány skutečnou adresou v operační paměti. My však pro usnadnění práce parserům a interpreterům budeme datové položky opět reprezentovat jako by se jednalo o property s getter a setter metodou.

Dalším konceptem, který je v prostředí MEF často využíván jsou generické třídy. Jedná se vlastně o třídy, které jsou parametrizované nějakými typy. V .NET jsou generických třídy řešeny tak, že se pro každou použitou kombinaci parametrů generické třídy vytvoří typ zvlášť. Tento mechanismus proto budeme využívat i v našem typovém systému.

V .NET reprezentacích typů máme možnost získat seznamy atributů, které jsou na typu a jeho jednotlivých metodách definovány. Vzhledem k tomu, že v MEF jsou atributy hojně využívány, mohlo by se zdát, že pro náš typový systém bude také výhodné umožnit dotazy na definované atributy. Zde však hraje velkou roli fakt, že .NET vytváří typové reprezentace už při kompilaci, tudíž má relativně dost času na jejich tvorbu, kdežto náš typový systém bude muset podléhat častým změnám, které budou odrážet změny ve zdrojových kódech. Navíc budeme vytvářet pouze ty typy, které nás budou zajímat z hlediska kompozice. Které typy to jsou se však dozvíme až potom co na nich patřičné atributy objevíme. Proto pro nás bude výhodnější získávat informace z atributů již při zkoumání zdrojových kódů a tyto informace vkládat do typů už zpracované.

Typy budeme vytvářet z typových definic. Tak budeme v rámci projektu nazývat třídy, interface, enum a další elementy, které definují nějaký typ. Typové definice však nebudeme získávat pouze ze zdrojových kódů. Důležité také bude zpracování typových definic v referencovaných knihovnách, případně v knihovnách získaných při MEF kompozici.

Typové definice získané ze zdrojových kódů se budou často měnit. Jistě by nebylo výhodné, vytvářet typy ze všech typových definic, které v aplikaci objevíme. Jednak bychom museli téměř při každém uživatelském zásahu do zdrojového kódu nějaký typ přegenerovat a také pro zobrazení schématu kompozice budeme obvykle potřebovat pouze několik málo typů, které stačí vytvořit až při první žádosti o jejich použití.

## 1.6 Objektový model

Jak jsme výše uvedli, je pro spouštění composition point nezbytný objektový model, který bude schopen pracovat nad námi vytvořeným typovým systémem. Naimplementujeme tedy vlastní objektový model, který nám umožní vytvářet objekty i z nezkompilovaných typových definic. Tyto objekty budeme v rámci projektu nazývat instance.

Výhoda instancí nespočívá pouze v tom, že je narozdíl od .NET objektů můžeme vytvářet z typových definic získaných ze zdrojového kódu, ale také v plné kontrole nad jejich vytvářením a voláním jejich metod. To je totiž důležité, proto aby byl editor schopen rozpoznat, že analyzované kódy jsou příliš složité – nejspíš zacyklené, a mohl analýzu ukončit aniž by došlo k jeho „zamrznutí“. Vzhledem k tomu, že se jedná o algoritmicky neřešitelný problém známý pod názvem „halting problem“, nemáme jinou možnost, jak se zacyklení vyhnout. V rámci měření instrumentace bude editor hlídat počet zavolaných funkcí a počet vytvořených instancí v rámci jednoho composition point. Interpretery navíc budou moci volitelně udávat informace o počtu vykonaných instrukcí, potřebných pro interpretaci metody.

Instance dále využijeme k tomu, abychom si v nich pamatovali místa ve zdrojovém kódu, která se pojí s nějakým důležitým voláním. Ta potom bude editor využívat k vytváření editací.

Koncept instancí s sebou však přináší i jisté problémy. Tak jak jsme ho uvedli, by bylo totiž velmi pracné reprezentovat primitivní typy .NET. Vzhledem k tomu, že volání na instancích vracejí opět instance a jiné operace k dispozici nemáme, nedokázali bychom z nich získat například ani číselnou hodnotu. Bude tedy dobré, aby instance reprezentující objekty jednoduchých typů zpřístupnili reprezentovaný objekt v .NET podobě.

## 1.7 Vytváření editací

Důležitou vlastností editoru bude nabízení editací nad zobrazeným schématem kompozice. V kapitole 1.2. jsme rozmýšleli, které editace by bylo vhodné uživateli nabídnout. Nyní se zaměřme na to, jaké změny ve zdrojovém kódu tyto editace vyžadují.

Zde je uvedeno několik modelových editací, které budeme uživateli nabízet.

* Přesunutí katalogu A z katalogu B do katalogu C – stačí smazat volání funkce, které přiřazuje A do B a vytvořit volání, které přiřadí A do C
* Změna cesty pro DirectoryCatalog – cesta musí být určena v konstruktoru DirectoryCatalogu, proto stačí změnit tuto položku na požadovanou cestu.
* Přidání komponenty do CompositionContainer – Pokud do kontejneru nebyla ještě žádná kompnenta přidána, přidáme ji vytvořením volání funkce ComposeParts. Pokud již nějaké komponenty obsahuje, přidáme ji jako nový parametr do zmíněného volání.
* Vytvoření nového objektu – deklarujeme proměnnou, do které bude nový objekt uložen. Vytvoření samotného objektu se provede pomocí vyvolání konstruktoru.
* Smazání objektu – abychom byli schopni korektně smazat objekt, musíme ze zdrojového kódu odstranit všechna volání, přiřazení a výrazy ve kterých se vyskytl.

K úpravám zdrojového kódu však musíme zmínit ještě několik důležitých upřesnění.Pokud chceme odstranit nějaký objekt z volání metody, měli bychom zohlednit, zda se vyskytuje pouze jako volitelný argument, v tom případně nemusíme odstraňovat celé volání. Dále, pokud odstraňujeme objekt z nějakého přiřazení, musíme dát pozor, zda se neodstraňujeme deklaraci. V tom případě by se totiž mohlo stát, že musíme proměnnou deklarovat znovu, na místě jejího dalšího použití.



Ukázka případu, kdy je nutné po odstranění objektu Obj1 předeklarovat proměnnou variable1. Za povšimnutí také stojí fakt, že nemůžeme předeklarování provést pomocí klíčového slova var, neboť variable1 by byla deklarována pod jiným typem.

Dalším významným problémem, který je při úpravách zdrojového kódu nutné vyřešit, je rozsah platnosti proměnných, ve kterých jsou objekty dostupné. Může se totiž stát, že objekt který přidáváme třeba do nějakého katalogu nemusí mít platnost ve stejném místě. Aby byl editor použitelný i v těchto situacích, musíme umět přesunovat příkazy ve zdrojovém kódu.



Platnost proměnné component končí před začátkem platnosti kontejneru. Chceme-li do něj komponentu přidat, musíme přesunout řádek rušící platnost proměnné component.

Přesunování příkazů ve zdrojovém kódu by však mohlo způsobit nechtěné změny v programu. Pokud však zajistíme, že přesunutí příkazů nezmění pořadí prováděných operací na jednotlivých objektech, jistě se ani nezmění celkový význam zdrojového kódu. Díky tomu však nemusí být možné jednoduše pozměnit kód, tak aby se projevila požadovaná editace a přitom nedošlo k žádným vedlejším efektům.

Ukázka zdrojového kódu, kde není možné přesunutím řádků přidat komponentu do CompositionContainer

Stejně tak nebude možné, aby editor přesunul objekt platný pouze v jedné metodě do objektu platného v jiné metodě. V takových případech editor dá uživateli najevo, že požadovanou editaci není možné provést.

Aby byl editor maximálně rozšiřitelný, musí být výše uvedené úpravy zdrojového kódu reprezentovány nezávisle na cílovém jazyku. Až při jejich provádění se patřičné jazykové rozšíření postará o provedení těchto úprav.

Už víme, jak se dají provádět úpravy ve zdrojovém kódu, stále však neumíme o objektu rozhodnout, které editace má nabízet. Je zřejmé, že tyto editace závisí na typu objektu, navíc ale musíme vzít v potaz jeho aktuální stav. Nemá například smysl nabízet editaci odebrání komponenty z kontejneru, který žádnou komponentu neobsahuje.

Máme tedy možnost vypsat pravidla, která rozhodnou, jaké editace zobrazíme pro daný objekt na základě zkoumání jeho stavu. Takový objekt pak bude muset shromažďovat veškeré údaje o všech volaných metodách, neboť nebude schopen dopředu odhadnout, zda nebudou v budoucnu potřeba. Zajímavější přístup je ale ten, kdy upravíme chování objektu tak, aby sám už v průběhu interpretace rozhodoval, které editace pro něj dávají smysl. Navíc díky rozšiřitelnosti typového systému může uživatel editace upravovat podle svých potřeb a tím editor přizpůsobit konkrétnímu projektu.

## 1.8 Vykreslování instancí

V průběhu interpretace composition point, získáme seznam všech instancí se kterými se pracovalo. Jakým způsobem ale zjistit, které instance jsou důležité z hlediska kompozice, abychom je mohli zobrazit ve schématu? Zobrazování každé instance typu string nebo int by rozhodně přehledné schéma nevytvořilo. Naopak pokud se v composition point objeví instance typu DirectoryCatalog, je téměř jisté, že v obvyklých případech bude tento katalog použit při kompozici.

Dává tedy dobrý smysl zobrazování instancí podle toho, zda mají typ, který je zajímavý z hlediska kompozice. Určit které typy jsou zajímavé a které nikoliv je však problematické. Každému uživateli může vyhovovat něco jiného. Dovolme tedy specifikovat způsob zobrazení instance, na základě jejího typu, přes uživatelská rozšíření. Editor se poté rozhodne pro zobrazení či nezobrazení instance ve schématu podle dostupných definic zobrazení.

Jedinou vyjímku tvoří zobrazení komponent. Komponenty mohou být různých typů, navíc v běžných případech typ komponenty způsob jejího zobrazení ovlivňovat nemusí. Pokud tedy editor v composition point zaregistruje komponentu, bude automaticky zobrazena ve schématu kompozice.

Máme definován způsob zobrazení komponent na základě analýzy jednoho spuštění konkrétního composition point. Po změnách interpretovaných metod je však nutné schéma kompozice obnovit. Opětovným spuštěním composition point získáme aktualizované podklady pro zobrazení schématu. Problém však nastane, budeme-li chtít uživateli například umožnit pozicování objektů ve schématu kompozice. Nové spuštění composition point vytvoří totiž nové instance pro zobrazení, které s těmi starými nejsou nijak svázány. Potřebujeme tedy identifikaci instancí napříč jednotlivými spuštěními composition point.

Identifikace založená na pořadí v jakém se instance v composition point objevují by nebyla příliš vhodná, neboť by ji mohlo narušit pouhé přidání nové instance. Zjišťování které instance přibyly, ubyly nebo jak se promíchalo jejich pořadí by korektní výsledek zaručilo, jeho implementace by však byla obtížná. Využijme tedy toho, že instance jsou po vytvoření obvykle přiřazeny do nějaké proměnné. Název proměnné není přiliš často měněn, můžeme tedy identifikaci instancí odvodit z něj. To nám dostatečně spolehlivě umožní zapamatovat si pozici a další údaje spojené se zobrazením instance napříč překresleními schématu kompozice.

# 2. Rozšiřitelnost Microsoft Visual Studia 2010

## 2.1. Projekt VsPackage

## 2.2. EnvDTE

## 2.3. CodeElement

## 2.4. Reakce na uživatelské události

# 3. Implementace editoru

## 3.1 Struktura



Na obrázku je zachycena základní struktura editoru. Po spuštění Visual Studia je zavedena assembly MEFEditor, obstarávající uživatelské rozhraní, propojení s Visual Studiem, nahrání standardních rozšíření a spuštění hlavní assembly Main.

Main po svém startu prohledá složku Extensions v kořenovém adresáři editoru a nahraje nalezená uživatelská rozšíření. Následně je napojen modul ChangesHandling pomocí objektu EnvDTE na události týkající se změn aktivních solution, přidávání/odebírání projektů a zásahů do zdrojových kódů. Jakmile tento modul zaregistruje nově otevřené solution, ohlásí v něm nalezené typové definice jako nově přidané, což způsobí že je TypeSystem modul načte spolu s referencovanými knihovnami.

Díky načtení typového systému modul Drawing získá seznam composition point, který je zobrazen uživateli v prostředí editoru. Aktivováním některého composition point v uživatelské nabídce dojde k jeho spuštění modulem ObjectModel. Na základě získaných výsledků je pak modul Drawing schopen zobrazit schéma kompozice.

ObjectModel ke své práci využívá modul DependencyHandling, řídící závislosti na zdrojových kódech a nahraných knihovnách. Díky němu dokáže editor například reagovat na změny provedené do zdrojových kódů patřičnými úpravami v typovém systému. Na základě těchto změn je poté rozhodováno, zda je nutné překreslit schéma kompozice.

Editace jsou v editoru řešeny na úrovni parserů a interpreterů. Parser typicky přidává do syntaktického a sémantického stromu informace o místě výskytu jednotlivých příkazů. Tyto informace potom může interpreter nabídnout zpracovávaným instancím, které jsou díky tomu schopné nezávisle na zpracovávaném jazyku vytvořit editace zobrazované uživateli.

Ne všechny editace však souvisejí s konkrétní instancí. Pro tyto editace je implementován modul Editations. Typickou editací poskytovanou tímto modulem je vytvoření nového objektu v composition point.

## 3.2 Typový systém

Stejně tak, jako jsou v .NET typy organizovány v jednotlivých assemblies, budou i typy našeho typového systému roztříděny do assemblies. Jejich reprezentaci obstarají objekty splňující rozhraní IAssembly. Toto rozhraní nám umožní vyhledávání typů podle plného jména, což je důležité pro parsery hledající typ dle jména získaného ze zdrojového kódu. Rozhraní dále zpřístupňuje seznam referencovaných assemblies a komponenty v assembly obsažené. Pomocí nich pak může například DirectoryCatalog pohodlně vyhledávat komponenty v objevených .NET knihovnách.

V souvislosti s pojmenováváním typů zavedeme několik definic. Pro plné jméno typu budeme používat označení fullname. Ve fullname jsou všechny případné generické parametry dosazené opět nějakým fullname. Naproti tomu rawname je fullname, bez dosazených parametrů. Alternativní název k fullname nazýváme alias. Posledním označením, které budeme potřebovat je signature. Získáme ho tak, že z rawname/fullname odstraníme generické parametry.

fullname : System.Collections.Generic.Dictionary<System.String,System.Int32>

rawname : System.Collections.Generic.Dictionary<TKey,TValue>

signature : System.Collections.Generic.Dictionary<,>

alias : string *- alternativní název k System.String*

Ukázka pro fullname, rawname, signature a alias. Pro pochopení dalšího textu je nutné dobře rozlišovat uvedené jmenné konvence.

Vlastní typy jsou v našem typovém systému navrženy tak, aby mohly snadno reagovat na změny typových definic, ze kterých pocházejí. To si vynucuje použití několika stádií, ve kterých budeme typy udržovat. Jedná se o TypeTicket, který je závislý pouze na signature typu. Z TypeTicket budeme vytvářet TypeDefinition, ta už bude obsahovat kompletní informace získané z typové definice. TypeDefinition však stále nebude řešit dědičnost. Ta bude zohledněna až v InternalType, což už je reprezentace typu použitelná pro objektový model.

### 3.2.1 Reprezentace assemblies

Editor používá tři typy assemblies pro nahrávání InternalTypes. Liší se v tom, odkud získávají typové definice. Pro správu typových definic získaných ze zdrojových kódů slouží UsrAssembly. Editor však musí umět získávat typové definice i z MSIL knihoven. Tato funkcionalita je implementována v MSILAssembly, která potřebné informace získává pomocí Mono.Cecil.

Posledním typem assembly je Runtime. Tato assembly slouží pro nahrávání typových definic ze standardních a uživatelských rozšíření. Při vyhledávání typů pak mají prioritu typy definované v Runtime, díky čemuž můžeme pomocí uživatelských rozšíření upravit chování jinde definovaných typů.

#### 3.2.1.1 UsrAssembly

UsrAssembly reprezentuje assembly získanou ze zdrojových kódů projektu otevřeného ve Visual studiu. Inicializuje se pomocí EnvDTE.Project, což je CodeModel objekt, který nám poskytuje hierarchickou reprezentaci daného projektu. Jediným významným krokem při inicializaci UsrAssembly je nahrání referencovaných projektů a knihoven. To probíhá tak, že každou získanou referenci nahrajeme pomocí volání AssemblyLoader.LoadAssembly(). Pokud se nějakou assembly nepodaří nahrát, je o tom uživatel informován pomocí varování.

Zásadním úkolem UsrAssembly je správa všech typů, které jsou dostupné v příslušném projektu. Typy v UsrAssembly mohou být odstraňovány, přidávány nebo kompletně měněny v závislosti na akcích, které uživatel provádí ve zdrojovém kódu. Aby bylo možné tyto změny zachytit a správně ošetřit, využívá UsrAssembly CodeModelManager, jenž je napojen na modul ChangesHandling. Dostává tedy informace o přidávání a odebírání CodeModel objektů, na základě kterého dokážeme odebírat a přidávat typy z UsrAssembly. Další změnou, která musí být do typového systému promítnuta je změna ve struktuře typu, čímž rozumíme změnu v názvech metod, datových položek, nebo jejich přidání či odebírání. Zaregistrujeme-li takovou změnu, odstraníme TypeDefinition a InternalType stádia změněného typu, pokud byla přítomna. Změna struktury typu však neovlivní TypeTicket, tudíž ho můžeme využít na vytvoření opravené TypeDefinition.

Jiným typem změn, které mohou nastat jsou změny v tělech metod. Tyto změny neovlivní TypeDefinition ani InternalType, není tedy nutné rušit jejich platnost.

Pro správu TypeTicket, TypeDefinition a InternalType využívá UsrAssembly třídu TypesManager, která sdružuje všechny potřebné operace pro vytváření, rušení a opravování typů.

#### 3.2.1.2 MSILAssembly

Pro nahrávání assemblies z .NET knihoven je používána třída MSILAssembly. Při její konstrukci ji předáme umístění knihovny, která má být nahrána a dále příznaky určující způsob nahrání assembly. MSILAssembly totiž může být nahrána se zapnutým/vypnutým vyhledáváním komponent a s úplným nebo dirty vytvářením typových definic. Upřesnění způsobů nahrání slouží pro úsporu strojového času na testy, které nepotřebujeme provádět vždy. Zejména jsou tato omezení vhodná pro nahrávání referencovaných knihoven, jelikož se převážně jedná o systémové knihovny, které jsou rozsáhlé, a nemají na výslednou kompozici prakticky žádný vliv.

Samotné získávání údajů ze souboru knihovny je prováděno pomocí nástrojů Mono.Cecil. Pro tyto účely jsou mnohem vhodnější než .NET nástroj Reflection, neboť umožňují zkoumat knihovnu i bez jejího zavádění do aplikační domény, což je značně pomalé. Navíc assembly nahraná pomocí Reflection bývá zamknuta pro zápis, což znemožňuje použití v prostředí, kde jsou zkoumané knihovny vyvíjeny a často překompilovávány.

Samotné načítání typových definic je umožněno díky kolekci Types, získané z Mono.Cecil.AssemblyDefinition. Z kolekce získáme Mono.Cecil reprezentaci typu, kterou potřebuje třída MSILTypeDefinition pro vytvoření TypeDefinition.

Nahrávání typů v MSILAssembly nevyužívá TypeTicket, neboť TypeTicket by musel obsahovat referenci na Mono.Cecil.AssemblyDefinition, ze které pochází. Tato reference pak znemožní garbage collectoru odstranění zkoumané assembly z paměti. To by způsobovalo neúměrné paměťové nároky editoru.

#### 3.2.1.3 Assembly Runtime

Typové definice získané z uživatelských rozšíření editoru jsou sdružovány v assembly Runtime. Ta slouží pro jednotnou správu těchto rozšíření a stará se o jejich správné nahrání a otestování. Vzhledem k účelu k jakému tato assembly slouží v ní nejsou definovány žádné reference a neprovádí se zde ani vyhledávání komponent. Stejně tak zde nejsou řešeny žádné závislosti na okolní zdroje, neboť nahrání rozšíření je prováděno pouze při startu editoru a není dovoleno je dynamicky měnit za běhu.

Po nahrání všech typových definic do assembly Runtime je spuštěna kontrola, která se pokusí vytvořit InternalType z každé negenerické typové definice. Pokud se vytvoření nějakého typu nezdaří, je uživateli vypsáno chybové hlášení. Díky tomu je možné už při spouštění editoru odhalit některé chyby v uživatelských rozšířeních.

Obdobný test pro generické typy by však byl jen těžko realizovatelný vzhledem k různorodosti parametrů, které generické typy obvykle podporují. Z tohoto důvodu editor neobsahuje žádnou předběžnou kontrolu správnosti typových definic generických typů.

#### 3.2.1.4 AssemblyLoader

AssemblyLoader je statická třída, určená pro nahrávání assemblies z rozličných zdrojů.

Zdrojem pro nahrání assembly může být EnvDTE.Project, cesta k souboru s MSIL knihovnou, nebo VSLangProj.Reference. AssemblyLoader nejdříve vytvoří patřičnou assembly a potom ji zaregistruje do DataCache. Při opětovném dotazu na stejnou assembly je vrácena cachovaná assembly, což výrazně urychluje práci editoru.

Abychom vyřešili cyklické závislosti mezi assemblies, je vytvářená assembly nejprve zaregistrována s příznakem InBuildState. Reference jsou vyhledány až po této registraci.

### 3.2.2 Reprezentace typů

Typy v našem typovém systému jsou navrženy tak, aby nebylo složité reagovat změny typových definic ze kterých pochází. Kvůli tomu jsou typy reprezentovány ve třech různých stádiích. Prvním stádiem je TypeTicket, který nese pouze signature typu a metodu na vytvoření TypeDefinition, které již implementuje ITypeDefinition interface. Nese tedy kompletní informace o typu až na dědičnost, virtuální metody a hodnoty generických parametrů. Ty jsou řešeny až v InternalType, který můžeme z TypeDefinition vytvořit. Tím získáme reprezentaci typu, použitelnou pro vytváření instancí, a dotazovaní se na dědičnost.

#### 3.2.2.1 TypeTicket

TypeTicket je v editoru používán jako poukázka na vytvoření TypeDefinition. Tento způsob reprezentace je vhodný zejména pro typy získané ze zdrojových kódů editovaného solution. Hlavní výhodou je to, že TypeTicket v sobě nese málo informací, tudíž ho většina změn provedených do zdrojového kódu neovlivní. Pouze pokud je cílový typ odstraněn, musíme zrušit i TypeTicket. Přejmenování typu je v rámci editoru interpretováno modulem ChangesHandling jako odebrání starého a přidání nového typu. Odpadá tedy nutnost měnit signature, se kterou byl TypeTicket vytvořen.

V případě, kdy potřebujeme reprezentovaný typ použít, nám však TypeTicket stačit nebude. Musíme nejprve vytvořit TypeDefinition zavoláním metody TypeTicket.UseTicket(), která vrátí objekt splňující rozhraní ITypeDefinition. Ten pak můžeme využít na vytvoření InternalType.

Jako nevhodné se ukázalo použití TypeTicket pro reprezentaci typů získaných z MSIL knihoven. Využívané informace z Mono.Cecil v sobě interně nesou odkaz na assembly ze které pocházejí. Tudíž jejich držení v paměti znemožňuje garbage collectoru odstranit tuto assembly. Na druhou stranu, u typů získaných ze zkompilovaných knihoven nedochází ke změnám příliš často, tudíž reprezentace pomocí TypeDefinition nečiní žádné problémy.

#### 3.2.2.2 TypeDefinition

Objekty splňující ITypeDefinition rozhraní jsou považovány za TypeDefinition stádium reprezentace typu. V editoru je toto rozhraní implementováno třídami UsrTypeDefinition vytvářených z typových definic ve zdrojových kódech otevřeného solution, MSILTypeDefinition z typových definic MSIL knihoven. Rozhraní ITypeDefinitinon musí být také implementováno standardními a uživatelskými rozšířeními objektového modelu.

Úkolem TypeDefinition je nést informace o metodách, předcích, datových položkách a dalších prvcích, které slouží jako podklady k vytvoření InternalType. Vzhledem k tomu, že TypeDefinition již obsahuje komplexní informace o typu, je ovlivněna nejen odebráním cílové typové definice, ale také změnami ve struktuře typu.

To se nejvíce týká UsrTypeDefinition, neboť ji zneplatní například i změna parametrů nějaké její metody. Bylo by však náročné zjišťovat která konkrétní změna struktury typu byla na UsrTypeDefinition provedena, proto je při každé takové změně zneplatněna celá. Pokud budeme chtít využívat změněný typ, je nutné opětovné vytvoření TypeDefinition z TypeTicket. Tato nová TypeDefinition bude obsahovat pozměněnou metodu, tudíž nově vytvořený InternalType ponese aktuální informace.

TypeDefinition je vázáno na rawname konkrétní typové definice. Pro generické třídy to tedy znamená, že z jedné TypeDefinition může být vytvořeno několik různých InternalType – pro každou použitou kombinaci generických parametrů jeden.

#### 3.2.2.3 InternalType

InternalType je reprezentace typu použitelná v našem objektovém modelu. Abychom ji vytvořili, musíme konstruktoru InternalType předat IAssembly, ve které bude obsažen, dále TypeSystemInfo obsahující údaje o generických parametrech a nakonec TypeDefinition popisující vytvářený typ.

Tvorba InternalType probíhá ve dvou krocích. Při volání konstruktoru jsou inicializovány položky FullName, Alias a InternalAssembly pomocí TypeSystemInfo a TypeDefinition. Dále je nastavena položka InBuildState, která signalizuje, že tvorba InternalType nebyla dokončena. V tomto kroku nepotřebujeme vytvářet žádný jiný InternalType objekt. Nemůže zde tedy dojít k zacyklení.

Po volání konstruktoru je v příslušné assembly typ registrován pod svým fullname, následně je na něm zavolána metoda InternalType.BuildType(), která dokončí vytváření InternalType. Při tomto dokončování je nejprve vyřešena dědičnost přes všechny předky voláním metody InternalType.inheritFrom(). Zde je však nutné mít přístup k InternalType předka. Ten však nemusí být v této době zkonstruován (jeho prvním použitím je právě vyřešení dědičnosti vytvářeného typu). Při konstrukci InternalType předka se ale můžeme dostat do problémů s cyklickou závislostí. Příkladem buď typ A, dědící od typu B a typ B dědící od A. Jelikož je dědičnost řešena až po zaregistrování typu v patřičné assembly, typ B nalezne vytvořený typ A. Ten však bude mít nastaven příznak InBuildState.

Obrázek znázorňující postup konstrukce typu A, typu B a následné selhání kvůli kruhové závislosti.

Typ B proto vyvolá vyjímku, značící cyklickou závislost a uživateli je zobrazeno varování, které informuje že při vytváření typu došlo k chybě.

Po vyřešení dědičnosti jsou do InternalType přidány getter a setter metody pro každou datovou položku definovanou v TypeDefinition. Dále je pak vytvořeno ComponentInfo, které zpřístupňuje údaje o importech, exportech a composition point, definovaných na typu.

Vytvoření ComponentInfo probíhá tak, že interpretujeme instrukce získané z atributů důležitých pro MEF pomocí volání ObjectModelManager.InterpretInline, které bude podrobněji popsánov v kapitole TODO. Díky této interpretaci získáme reprezentaci atributu, která nám poskytne informace pro vytvoření ComponentInfo.

Tím bylo vytváření InternalType dokončeno a můžeme nyní nastavit příznak InBuildState na false. InternalType poskytuje seznam předků, informace o tom zda reprezentuje abstraktní typ, z jaké pochází assembly, jaké má fullname, případně jaký má alias. Dále zpřístupňuje seznam datových položek a seznam implementovaných i abstraktních metod. Pro potřeby komponentového modelu obsahuje položku InternalType.ComponentInfo.

Abychom mohli najít implementaci metody, na základě různých kritérií, obsahuje InternalType metodu GetMethod. Ta je využívána především parsery, které musí podle parametrů volané metody najít její vhodné přetížení.

Pro testování zda je reprezentovaný typ potomkem jiného typu, zadaného pomocí fullname slouží metoda Is.

#### 3.2.2.4 Dědičnost

Dědičnost se v typovém systému řeší na úrovni InternalType. Při dědění od předků jde o to, aby byly přidány, doimplementovány, případně přepsány metody předka, metodami které poskytuje potomek. K tomuto účelu slouží metoda InternalType.Inherit(), která je volána potomkem na svého předka. Při volání je předán seznam implementací metod, které zatím nebyly spojeny s žádnou virtuální ani abstraktní metodou. Předek tyto metody projde a na základě svých seznamů abstraktních, virtuálních a implementovaných metod se pokouší spojit odpovídající záznamy. Pro každou metodu pro kterou se mu to povede, zavolá na vytvářeném potomkovi InternalType.AddMethod(). Tím je zajištěn mechanismus virtuálních a abstraktních metod.

Pokud se nepodaří vytvářenému typu najít implementace všech abstraktních metod, nastaví si příznak InternalType.IsAbstract na true.

Jelikož do dědičnosti zasahuje i assembly Runtime, je například možné, aby ve zdrojovém kódu uživatelem upravený katalog odděděný od standardního MEF katalogu, podporoval zobrazení ve schématu kompozice a nabízel standardní editace.

#### 3.2.2.5 Generické typy

Aby bylo možné efektivně hledat v seznamech typů i přes parametričnost jejich názvů vzniklou kvůli generice, využijeme vlastností námi definovaných tvarů jmen.

Je snadné nahlédnout, že signature vytvořená z fullname i z rawname má stejný tvar. Díky tomu můžeme efektivně vyhledávat v seznamu TypeTicket a TypeDefinition indexovaném podle signature.

Pro vytvoření InternalType reprezentujícího generický typ získáváme informace o generickýh parametrech z fullname a rawname odpovídající TypeDefinition. Z rawname totiž získáme názvy generických parametrů a z fullname jejich hodnoty.

#### 3.2.2.6 Metody

Reprezentace metod v typovém systému podléhá požadavku na možnost využití virtuálního volání metod. Kvůli tomu, musíme oddělit deklaraci metody od její implementace. Deklarace splňují rozhraní IMethodInfo. Samotná implementace metody je reprezentována pomocí IMethod. Toto rozdělení vychází ze způsobu, jakým jsou ve zdrojovém kódu metody zapsány. Metodu můžeme mít deklarovanou například v abstraktní třídě. V té ale nemusí být uvedena implementace metody. Implementaci získáme až v potomkovi abstraktní třídy, který může být definován i v jiné assembly.

IMethodInfo obsahuje informace o návratovém typu, seznam svých parametrů, určuje zda se jedná o virtuální či statickou metodu, případně zda je metoda konstruktorem. V deklaraci metody si nemůžeme pamatovat přímo její rawname, neboť to je závislé až na konkrétní implemetnaci metody. Proto je v IMethodInfo dostupná pouze signature. Stejně tak se mohou lišit i definice parametrů metody v místě deklarace a implementace metody. Rozdílné mohou být nejen názvy, ale také defaultní hodnoty parametrů. Proto je důležité zejména pro potřeby parserů a interpreterů rozlišovat, v jakém kontextu používáme údaje z IMethod a z IMethodInfo.

IMethod reprezentuje skutečnou implementaci dané metody. Obsahuje odkaz na svou deklaraci, názvy a defaultní hodnoty svých parametrů platných v rámci těla metody, rawname a údaje potřebné pro interpretování metody. Těmi jsou IInvokeInfo s instrukcemi metody a jméno jazyka, ve kterém jsou instrukce napsány.

S využitím výše zmíněného konceptu můžeme nyní při parsování dohledat konkrétní deklaraci metody přes IMethodInfo. Při vyhledávání zohledníme různé způsoby přetěžování metod. O skutečnou implementaci si však řekneme za pomoci IMethodInfo až u konkrétního objektu, na kterém chceme metodu zavolat. Ten může velice rychle pomocí hashování najít odpovídající IMethod i se zohledněním virtuálnosti a následně metodu spustit.

Aby celý systém volání metod správně fungoval, je nezbytné při řešení dědičnosti spojit deklarace metod s patřičnými implementacemi. To nám zaručí, že všechny implementace abstraktních/virtuálních metod bude dohledatelná přes stejné IMethodInfo.

TODO: Rawname u IMethod

### 3.2.3 TypesManager

Za účelem zjednodušit správu stádií reprezentace typů v našem typovém systému je v editoru používán TypesManager. Každá assembly při své inicializaci vytvoří TypesManager, který je průběžně informován o nově objevených/zaniklých typech.

Nové typy jsou do TypesManageru přidávány metodami TypesManager.SetTicket() a TypesManager.AddDefinition(), v závislosti na tom, zda má assembly k dispozici TypeTicket nebo TypeDefinition. U přidaných typů TypesManager hlídá jejich platnost. TypeTicket je platný, dokud nebude příslušná typová definice odstraněna ze zdrojových kódů. Že došlo k odstranění zjistí TypesManager pomocí modulu ChangesHandling.

Rozhodnout o platnosti TypeDefinition je však složitější, neboť závisí na změnách metod, datových položek nebo seznamu předků. Všechny události, které by mohli zneplatnit TypeDefinition jsou proto pokryty v seznamu závislostí, dostupného v ITypeDefinition.Dependencies. Díky hlídání těchto závislostí dokáže TypesManager zjisit, kdy byla TypeDefinition zneplatněna. Pokud k zneplatnění dojde, je TypeDefinition odstraněna. Stejně tak jsou odstraněny všechny InternalTypes, které byly z této TypeDefinition vytvořeny. Podrobnějí popis závislostí je uveden v kapitole 3.5.

Platnost InternalType je jednak omezena platností TypeDefinition, ze které byl vytvořen, ale také změnou structure v libovolném předkovi. Všechny závislosti, které má InternalType proti TypeDefinition navíc, jsou uvedeny v InternalType.Dependencies. Pokud dojde ke zneplatnění některé závislosti InternalType, je tento InternalType odstraněn. Nemusíme však odstraňovat žádnou TypeDefinition, neboť závislosti uvedené v InternalType se jí netýkají.

Nad typy, které jsou v TypesManager obsaženy, můžeme vyhledávat pomocí TypesManager.FindType(). Tato metoda vyhledá pro zadaný fullname odpovídající InternalType. Vyhledávání probíhá tak, že k fullname zkusí najít odpovídající typ v již vytvořených InternalType. Pokud není nalezen žádný záznam, je fullname převedeno na signature a hledá se pro něj příslušná TypeDefinition. Pokud ani ta není nalezena, je poslední možností jak typ získat její vytvoření z TypeTicket. Pokud hledaný TypeTicket neexistuje je vráceno null, jako indikace, že požadovaný typ se v TypesManager nenalézá.

Pokud však máme k dispozici příslušnou TypeDefinition, získáme generické informace z daného fullname a vytvoříme nový InternalType. Na použité TypeDefinition je následně zavolána metoda ITypeDefinition.PreloadRoutines(), umožňující zjistit, zda byl InternalType správně vytvořen a zda jsou přítomné všechny potřebné prerekvizity. Pokud se vyskytne chyba, může PreloadRoutines() vrátit false, což způsobí zobrazení chybové hlášky uživateli.

Kontrola pomocí PreloadRoutines je obzláště užitečná pro uživatele, kteří si chtějí psát vlastní rozšíření typového systému editoru. Uživatelé totiž mohou v PreloadRoutines otestovat, zda je jejich rozšíření schopné pracovat s dostupnými typovými definicemi a na základě toho vypsat případné chybové hlášky již při startu editoru. To vede k výraznému ulehčení ladění uživalských rozšíření.

## 3.3 Objektový model

Vytvořený typový systém využíváme pro práci s instancemi. Ty jsou implementovány třídou Instance. V uživatelských rozšířeních je však přístup k instancím omezen pouze na rozhraní IInstance.

Nezbytnou součástí objektového modelu je systém pro volání metod na instancích. Samotné spuštění metody zajišťuje ObjectModelManager ve spolupráci s dostupnými rozšířeními, v podobě interpreterů a parserů.

Pro ulehčení použití instancí v rámci editoru není jejich vytváření omezeno pouze na volání konstruktoru, tak jak je tomu v .NET. Při analýze zdrojových kódů je někdy výhodné získat instanci aniž by na ní byl volán konstruktor. S výhodou také využijeme instance, vytvořené ze zadaného .NET objektu. To nám pomůže s reprezentací primitivních .NET objektů. Pro reprezentaci statických tříd pak využijeme instance vytvořené jako sdílené.

Korektní reprezentace veškerých typů v .NET by byla výpočetně příliš náročná, proto je editor zaměřen převážně na technologii MEF. Aby však byl schopen alespoň omezeně pracovat i s dalšími typy, nabízí objektový model koncept dirty instancí.

### 3.3.1 Instance

Objekty našeho objektového modelu jsou v editoru implementovány třídou Instance. Umožňuje nám spouštět metody, poskytuje datové položky použitelné v interpretovaných metodách a implementuje rozhraní pro vytváření editací specifických pro jednotlivé objekty.

Inicializace instance se stará o nastavení typu a získání defaultního ID instance, které je unikátní v rámci spuštění jednoho composition point. Toto ID je využíváno při vykreslování schématu kompozice. Díky němu můžeme získat souvislost mezi instancemi, které reprezentují stejný logický objekt, ale vznikly v různých spuštěních composition point. ID je v průběhu interpretace měněno na základě názvu proměnných a typů, se kterými daná instance souvisí.

Pro simulaci datových položek objektů v instanci využijeme metody SetData a GetData. Tyto metody nám umožní uložit/načíst libovolná data podle zadaného názvu. Metoda SetData však není dostupná přímo přes rozhraní IInstance. Nastavit data instanci je možné pouze v metodě na této instanci zavolané, přes rozhraní IExecInfo, které bude popsáno později. Toto opatření omezuje nechtěná přepsání cizích dat.

Pojmenování dat, která jsou uložena v instanci však musí podléhat určité konvenci. Pro efektivnější reprezentaci objektů, které jsou v instanci pouze „zabaleny“, by měl být obalovaný .NET objekt dostupný pod názvem určeným konstantou Constants.DataField\_Value. Tímto způsobem pak můžeme snadno získávat hodnotu z instancí reprezentujících objekt primitivního .NET typu. Instance které nepoužívají skutečný .NET objekt pro své fungování musí nechat v této položce hodnotu null.

Data získaná z instance pomocí GetData nebo IInstance.Value jsou však velmi závislá na implementaci typové definice. Proto je důležité, aby uživatel píšící rozšiřující typovou definici, která využívá uvedené služby, nejprve zkontroloval zda odpovídají chování, které od nich očekává v ITypeDefinition.PreloadRoutines().

Příkladem buď vytváření typové definice pro DirectoryCatalog. V konstruktoru DirectoryCatalog potřebujeme získat hodnotu z instance, která reprezentuje objekt typu string. Tuto hodnotu nejsnáze získáme pomocí IInstance.Value. V PreloadRoutines naší typové definice bychom však měli otestovat zda dostupná implementace typové definice pro string hodnotu v IInstance.Value opravdu uchovává. Test provedeme například vytvořením instance typu string a ověřením že IInstance.Value není null. V opačném případě můžeme v PreloadRoutines ohlásit, že naše typová definice nemůže s dostupnou implementací typové definice pro string pracovat.

### 3.3.2 Volání metod na instancích

Volání metod na instanci je prováděno přes IInstance.CallMethod(). CallMethod dostane jako parametr IMethodInfo, určující kterou metodu chceme volat. Dalším parametrem je ICallInfo, obsahující hodnoty argumentů a editace dostupné pro toto volání. Podrobné využití editací bude popsáno v kapitole 4.3.3.

K danému IMethodInfo je hledána implementace v podobě IMethod voláním InternalType.GetMethod(). Díky tomu, že již při hledání vhodné IMethodInfo byla zohledněna virtuálnost a přetížení metod, můžeme správnou implementaci získat rychlým přístupem za pomoci hashování. Pokud není implementace nalezena, je uživateli zobrazeno varovné hlášení, které může pomoci při ladění rozšíření editoru. Za normálních okolností by totiž k takové situaci docházet nemělo, neboť ošetření sémantických chyb je obvyklé provádět na úrovni parseru – před voláním metod.

Po nalezení potřebné IMethod je vytvořeno ExecInfo, zapouzdřující konext volání, současnou instanci jako this objekt a ICallInfo pro předání argumentů. Samotné spuštění se provede pomocí ObjectModelManager.RunMethod(), kde je metoda rozparsována a následně interpretována. RunMethod vrací návratovou hodnotu interpretované metody, případně null, pokud interpretace/parsování selže.

Při volání IMethod může dojít k vyjímce v parsování/interpretování. Tyto vyjímky mají za účel okamžité zastavení provádění composition point, tudíž musí být odchytávány pouze v místě spuštění composition point. Pokud k takové vyjímce dojde, je uživateli vypsána chybová hláška vysvětlující proč byl běh composition point předčasně zastaven.

### 3.3.3 Vytváření instancí

Abychom mohli v editoru používat instance, musíme je nejdříve vytvořit. Vytváření instancí se v některých ohledech podobá konstrukci objektů v .NET. Jsou zde však určitá specifika, která možnosti použití instancí oproti .NET objektům rozšiřují. Především se jedná o vytvoření instance bez volání konstruktoru, což bývá výhodné v případech, kdy nejsou dostupné argumenty pro vyvolání konstruktoru. Další možností, jak lze instanci vytvořit je injektováním ze zadaných dat, kdy je na instanci zavolán speciální konstruktor, přijímající jako argument libovolný .NET objekt.

O vytváření instancí se stará třída InstanceCreator, která umožňuje vytvoření instance všemi uvedenými způsoby. Každá vytvořená instance je v InstanceCreator registrována pro kontrolu překročení maximálního počtu vytvořených instancí. Už při vytváření instance se také rozhoduje, zda bude zobrazena ve schématu kompozice.

#### 3.3.3.1 Přímé vytvoření

V některých případech není při vytváření instance možné zavolat její konstruktor. Typickým případem je například spuštění composition point. Ten je metodou třídy (nazvěme ji vstupní třída), z níž musí být vytvořena instance, na které následně zavoláme onen composition point.

Pokud má vstupní třída bezparametrický konstruktor, je přirozené vytvořit instanci pomocí něj. Bezparametrický konstruktor však nemusí existovat. V tomto případě nebudeme na instanci volat žádný konstruktor a vytvoříme ji přímo, bez volání konstruktoru. To sice zamezí inicializaci datových položek vstupní třídy, ale vzhledem k účelu, ke kterému composition point slouží nám to příliš nevadí. Navíc uvedené omezení nebrání použití, které bylo při návrhu editoru preferováno, kdy za composition point označíme přímo konstruktor vstupní třídy a můžeme tedy specifikovat argumenty pro její konstrukci.

#### 3.3.3.2 Voláním konstruktoru

Pro některé jazyky mají konstruktory sémanticky odlišný význam od obyčejných metod. V našem objektovém modelu se však jedná o obyčejnou metodu, nesoucí speciální příznak IsConstructor, který může být využit při parsování/interpretování. Díky tomu je výsledný objektový model silnější, přičemž dává možnost odlišit chování metod od konstruktorů, pokud by to bylo potřeba.

Přestože tento způsob vytváření instancí není z pohledu objektového modelu nijak zajímavý, obsahuje InstanceCreator metodu InstanceCreator.ConstructInstance, usnadňující práci interpreterům/parserům, jelikož v jednom kroku provede přímé vytvoření instance a následné zavolání konstruktoru na vytvořené instanci.

TODO IInstance.IsConstructed

#### 3.3.3.3 Injektováním ze zadaných dat

ITypeDefinition vyžaduje dvojici metod ITypeDefinition.CanLoadFrom() a ITypeDefinition.DirectLoad(), sloužící pro přímé nahrávání instance ze zadaných dat. CanLoadFrom vrací true, na objekt, ze kterého je schopna vytvořit instanci. DirectLoad se poté chová jako konstruktor, který instanci inicializuje ze zadaného .NET objektu.

### 3.3.4 Sdílené instance

V programovacích jazycích existuje koncepce tříd, z nichž nelze vytvářet objekty. V .NET se těmto třídám říká statické třídy. V našem objektovém modelu jsou statické třídy reprezentovány pomocí sdílených instancí. Přístupné jsou přes InstanceCreator.GetSharedInstance(), kde najdeme instanci podle daného typu a vrátíme ji. Pokud instance nalezena není, je vytvořena voláním statického konstruktoru a přidána do kolekce sdílených instancí.

Tím je zajištěno, že v rámci jednoho spuštění composition point bude vytvořena nejvýše jedna sdílená instance pro každý typ. Vytváření statických instancí navíc typicky probíhá těsně před prvním použitím, když interpreter potřebuje získat sdílenou instanci aby na ní mohl například zavolat nějakou metodu.

### 3.3.6 Koncept dirty instancí

Vzhledem k tomu, že korektní reprezentace všech objektů použitých při interpretaci composition point by byla výpočetně velmi náročná, umožníme instancím nacházet se v dirty stavu. Dirty instance mohou vzniknout jako výsledek volání metody jejíž instrukce jsou označené dirty příznakem. Tento příznak mají metody pocházející z referencovaných MSIL knihoven, které obvykle nemají na kompozici velký vliv, proto ušetříme editoru práci s interpretováním jejich metod.

Dirty stav instance značí, že neznáme aktuální hodnoty jejích datových položek. Nemůžeme tedy korektně volat metody na dirty instanci, ani ji nemůžeme použít jako argument v nějakém volání. Dojde-li v průběhu interpretace k pokusu o takové volání, přejdou všechny zůčastněné instance do stavu dirty.

Pokud se dirty stav rozšíří i na instance, které by měli být zobrazeny ve schématu kompozice, je uživatel varováním informován, že zobrazené schéma kompozice nemusí být korektní.

## 3.4 Reakce na události vyvolané uživatelem

Editor musí pro potřeby překreslování schématu kompozice a udržování typového systému ve stavu odpovídajícímu zdrojovým kódům sledovat události, které vyvolává uživatel při práci ve Visual Studiu. Pro tyto účely je v editoru implementován modul ChangesHandling.

K událostem, jenž jsou sledovány, patří otevření/zavření solution, které způsobí znovunahrátí typového systému, stejně tak jako přidávání nebo odebírání projektů aktivního solution.

Jiným typem událostí, které je nutné sledovat, jsou změny zdrojových kódů v aktivním solution. Na jejich základě se editor rozhoduje, zda je nutné překreslit schéma kompozice, případně zda je nutné provést změny do typového systému.

Všechny potřebné události ošetřujeme pomocí handlerů zaregistrovaných u objektu typu EnvDTE.DTE.Events, který je přístupný pluginům Visual Studia. Implementace těchto handlerů jsou ve statické třídě VSEventManager, která spravuje události získané od Visual Studia.

### 3.4.1 Změna aktivního solution

Změny aktivního solution jsou indikovány událostmi otevření nového solution SolutionEvents.Opened a zavřením starého solution SolutionEvents.BeforeClosing.

Při otevření nového solution musí editor načíst celý typový systém. Z projektů v solution nejprve vytvoří UsrAssemblies a načte jejich referencované knihovny. V každém projektu pak vyhledá soubory se zdrojovými kódy. Nad každým takovým souborem je vytvořen objekt FileItemManager, který ohlásí přítomnost všech objevených typových definic a zároveň si zapamatuje údaje potřebné pro zjišťování změn v těchto definicích.

Po načtení typového systému je umožňěno třídě ComponentManager získat InternalType reprezentace typů pro objevené komponenty. U rozsáhlých aplikací může procházení typových definic a vytváření komponent trvat i několik sekund. Proto třída VSEventManager vyvolává události využitelné pro informování uživatele o průběhu načítání solution.

Při události BeforeClosing je nejprve zkontrolováno, zda editor stále nenačítá minulé solution. Pokud ano, tak je tato akce přerušena. Poté dojde k uvolnění všech dat, která editor pro solution používal, z paměti. Od této doby je editor neaktivní do doby, dokud není otevřeno další solution.

### 3.4.2 Změny v projektech aktivního solution

Visual studio umožňuje odebírání/přidávání projektů do aktivního solution. Tyto změny vyvolají události SolutionEvents.ProjectAdded nebo SolutionEvents.ProjectRemoved. Pokud k nim dojde, je znovunačten celý typový systém, tak jako by bylo zavřeno a znovu otevřeno aktivní solution. Vzhledem k tomu, že projekty nejsou měněny často je toto řešení výhodnější, než složité simulace přidávání a odebírání patřičných UsrAssemblies z typového systému.

Další změnou, která u projektu může nastat je jeho přejmenování. Tuto změnu však nemusíme registrovat, neboť jméno assembly není pro typový systém důležité. Významější změnou, která může být u projektu registrována je však přidání/odebrání referencované knihovny. Tuto změnu budeme řešit stejně jako v případě přidání/odebrání projektu, znovunačtením celého typového systému, neboť referencované knihovny se opět nemění příliš často.

### 3.4.3 Změny ve zdrojových kódech

Pro získávání informací o změnách ve zdrojových kódech nám Visual Studio nabízí sadu událostí CodeModelEvents, které jsou vyvolány například při přidání nebo odebrání typové definice ze zdrojového kódu. Použití těchto událostí se však ukázalo jako velmi nespolehlivé. Pro potřeby editoru je tedy naimplementován vlastní systém vyvolávání událostí vzniklých úpravami zdrojových kódů, založený na spolehlivé události TextEditorEvents.LineChanged, kterou Visual Studio vyvolá při každé změně ve zdrojovém kódu.

Tato událost však nese pouze informaci o pozici začátku a konce vybraného textu po provedení změny. Abychom mohli zjistit, jak dlouhý úsek původního textu změně odpovídá, musíme si pro každý soubor pamatovat celkovou délku před a po změně.

TODO: obrázek vysvětlující proč potřebujeme informaci o délce souboru

Tyto informace nemůžeme zpracovat přímo v handleru události, neboť by docházelo k výrazným prodlevám mezi stisknutím klávesy a odpovídající reakcí v textu zdrojového kódu. Místo toho budeme všechny změny pouze registrovat a zpracujeme je dávkově až v době, kdy uživatel se zdrojovým kódem nepracuje.

Zpracování získaných změn pak probíhá ve FileItemManager objektu pro každý upravený soubor. FileItemManager si udržuje informace o pozicích CodeElement, které jsou v něm obsaženy, a díky zaznamenaným změnám dokáže zjistit, do kterých CodeElement bylo zapisováno a jak se změnili jejich pozice.

Pouze z údajů o místě výskytu a délce změny zdrojového kódu však není možné ve všech případech korektně opravit pozice CodeElement objektů. Proto jsou tyto pozice kontrolovány a při zjištění rozdílu je příslušný CodeElement označen jako změněný.

#### 3.4.3.1 Přidání nového CodeElement

FileItemManager spravuje seznam všech CodeElement obsažených ve zdrojovém kódu příslušného souboru. Pokud je v souboru zjištěna změna, je soubor zkontrolován a CodeElement objekty, které nebyly v seznamu jsou ohlášeny pomocí DependencyManager.OnAdd. Díky tomu editor zjistí přidání nových typových definic.

#### 3.4.3.2 Odebrání CodeElement

Obdobně jako v případě přidávání CodeElement je ve FileItemManager využit seznam všech dosud zjištěných CodeElement objektů. Při změně souboru je zjištěn nový seznam dostupných CodeElement a všechny objekty, které byly ve starém seznamu, ale v novém se již nevysktují jsou voláním DependencyManager.OnRemove prohlášeny za odebrané. Editor má díky tomu dostatečné informace pro odebírání typových definic z typového systému.

#### 3.4.3.3 Přejmenování CodeElement

Pokud přejmenujeme například třídu ve zdrojovém kódu, tato změna se projeví smazáním CodeElement objektu příslušejícího ke třídě s původním názvem a vytvořením nového CodeElement pro nový název třídy. Jedná se tedy o odebrání a přidání CodeElement objektu, které registrujeme výše uvedeným způsobem.

#### 3.4.3.4 Změna těla CodeElement

Pro zjišťování přidávání a odebírání CodeElement nevyužíváme informaci o místě, kde ke změně došlo. Pro potřeby editoru však musíme umět rozhodnout i o tom, zda došlo ke změně nějakého CodeElement. Taková změna například pro aktuálně vybraný composition point znamená, že musíme zaktualizovat schéma kompozice.

Za změnu těla CodeElement považujeme takovou změnu, která zasáhla CodeElement na místě, které nepatří žádným jeho potomkům.

TODO obrázek, který ukazuje změnu uvnitř třídy vnořené ve třídě – změna těla je registrována pouze u vnitřní třídy

CodeElement je také považován za změněný, pokud se jeho uchovávaná pozice po opravě neshoduje se skutečnou pozicí. Všechny změněné CodeElement objekty jsou ohlášeny pomocí DependencyManager.OnChange.

## 3.5 Systém závislostí

S využitím událostí, které nám poskytuje modul ChangesHandling implementujeme závislosti na typových definicích, zdrojových kódech interpretovaných metod a dalších cílových objektech. Závislostí uvažujeme několik druhů, podle typu změny, kterou sledujeme. Typ změn je určen dle DependencyType, která může nabývat následujících hodnot:

* change – změna na cílovém objektu, nezahrnuje však odstranění ani přidání objektu
* bodyChange – změna ve zdrojovém kódu metody
* remove – odstranění cílového objektu
* all – pokud dojde k libovolné výše uvedené změně nebo přidání objektu

Díky závislostem máme jednoté rozhraní, jak hlídat platnost TypeDefinition nebo InternalType. Stejně tak můžeme hlídat závislosti interpretace composition point, podle kterých poznáme kdy je nutné překreslit schéma kompozice. Dalším využitím je hlídání platnosti cachovaných rozparsovaných instrukcí, které mohou záviset jak na dostupných typech, tak na zdrojových kódech ze kterých pocházejí.

Závislosti jsou reprezentovány objektem IDependency, získaným od DependencyManager. Ten nám také umožní k závislostem přiřazovat metody, které budou spuštěny, pokud dojde ke změně na patřičné události.

### 3.5.1 Cíle závislostí

Závislost je určena typem změny, kterou sledujeme a cílovým objektem, na němž chceme změnu sledovat. Cílovým objektem může být CodeElement, název typu nebo cesta k souboru či ke složce.

#### 3.5.1.1 Závislost na CodeElement

Změny v objektech CodeElement jsou hlídány pomocí FileItemManager, jak bylo popsáno v kapitole 3.4.3. Cílovým CodeElement objektem tedy může být metoda, nebo CodeElement patřící typové definici.

Závislosti na metodách jsou využívány pro InvokeInfoCache, která díky změnám v těle metody zneplatní výsledky parserů, které z ní vycházejí. Pokud se změněná metoda navíc vyskytla při interpretování aktuálně zvoleného composition point, dojde k jeho překreslení.

Změna v CodeElement typové definice je pak využita pro zneplatnění TypeDefinition a InternalType, které z ní pocházejí. Pokud dojde k odstranění CodeElement typové definice, je navíc odstraněn příslušný TypeTicket.

#### 3.5.1.2 Závislost na názvu typu

O správu typů se stará TypesManager, přítomný v každé IAssembly používané v editoru. Pro každou změnu na typu pak TypesManager volá patřičnou událost u DependencyManager.

Závislosti na názvu typu jsou důležité pro parsery. Pokud je nějaký typ přidán nebo odebrán, může tím být ovlivněn výsledek parsování. Při parsování tedy registrujeme veškeré typy, které parser požaduje. Pro názvy těchto typů pak vytváříme závislosti, které výsledek parsování zneplatní.

TODO: Ukázka, jak může přidání typu ovlivnit výsledek parsování

Pokud byly zneplatněné instrukce navíc použity při interpretaci aktuálně vybraného composition point dojde k překreslení schématu kompozice.

#### 3.5.1.3 Závislost na souborech a složkách

Systém závislostí umožňuje hlídat změnu souboru nebo složky. Cílovým objektem pro takovou závislost je objekt typu string nesoucí cestu k souboru nebo složce. Samotné sledování změny je pak prováděno třídou FileSystemChanges, která využívá systémové prostředky.

Závislosti na souborech jsou důležité pro zjišťování změn v MSIL knihovnách. Pokud vytvoříme MSILAssembly z nějaké knihovny, je tato assembly cachována až do doby, dokud se nezmění soubor ze kterého byla vytvořena.

Závislost na složkách je využívána například DirectoryCatalogem, díky které můžeme překreslit schéma kompozice při změně sledované složky.

### 3.5.2 Závislosti instrukcí v IInvokeInfo

Protože editor umožňuje uchovávat výsledky parsování, musí být také schopen rozhodnout do kdy jsou tyto výsledky platné.

Výsledky parsování mohou být zneplatněny dvěma způsoby. Buď dojde ke změně zdrojových instrukcí metody ze které pocházejí, nebo se změní typy použité v průběhu parsování. Změny zdrojových instrukcí jsou ošetřeny závislostí na patřičný CodeElement objekt. Změnu použitých typů pak registrujeme pomocí závislostí na jméno každého typu, který byl parserem použit.

Uvedené závislosti jsou využity ve třídě InvokeInfoCache, která se stará o cachování mezivýsledků parsování.

### 3.5.3 Závislosti stádií reprezentace typu

Správa stádií typu je postavena na hlídání jejich závislostí. Tuto správu vždy zařizuje příslušný TypesManager.

Závislosti TypeDefinition a InternalType pocházející z MSIL knihoven jsou jednoduché. Závisí pouze na souboru MSIL knihovny a jejích referencích. U reprezentace typových definic získaných ze zdrojových kódu je však situace složitější. Na čem mohou jednotlivá stádia záviset je uvedeno zde:

* TypeTicket – závisí pouze na odstranění CodeElement objektu patřičné typové definice.
* TypeDefinition – závisí na libovolné změně CodeElement objektu patřičné typové definice
* InternalType – kromě závislostí, které má společné s TypeDefinition, navíc obsahuje závislost na jméno typu každého předka

### 3.5.4 Závislosti assemblies

Assemblies získané z projektů aktivního solution žádné závislosti nedefinují. Pokud dojde k nějaké změně na projektu, je znovunačten celý typový systém. MSILAssemblies jsou však závislé na souboru, ze kterého byly nahrány a také na souborech referencovaných knihoven. Tyto závislosti jsou pak využívány pro zneplatnění cachovaných assemblies, případně pro překreslení schémetu kompozice.

### 3.5.5 Závislosti schématu kompozice

Závislosti schématu kompozice slouží pro zjištění okamžiku, kdy je nutné toto schéma překreslit. V průběhu interpretace zvoleného composition point jsou registrovány závislosti všech interpretovaných instrukcí. Dále jsou sbírány závislosti z uživatelských typových definic. Ty mohou přidat závislosti na soubory nebo složky, což s výhodou využijeme například pro sledování změn složky, se kterou pracuje zobrazený DirectoryCatalog.

Pokud je registrována změna na některé závislosti, je uživatelské rozhraní editoru patřičně upozorněno a provede překreslení schématu kompozice.

## 3.6 Editace

V editoru jsou editace řešeny s ohledem na nezávislost na konkrétním jazyku, díky rozhraní ILanguageDefinition, které nám dává základní operace pro práci se zdrojovými kódy.

Samotná editace pak vznikne složením několika transformací zdrojového kódu. Díky tomu můžeme například snadno spojit transformaci pro přeházení řádků zdrojového kódu s transformací, která vytvoří volání funkce. Ve výsledku pak dostaneme editaci, která například umožní přijmout komponentu pomocí CompositionContainer objektu.

V prostředí editoru pak rozlišujeme editace poskytované instancí a takzvané statické editace, které se k instancím přímo nevztahují.

### 3.6.1 ILanguageDefinition

Zápis změn do zdrojového kódu metod je prováděn pomocí ILanguageDefinition objektu, odpovídajícího jazyku měněné metody. Díky tomu nemusíme vytvářet editace pro každý podporovaný jazyk zvlášť, ale stačí nám editace využívající jednotné rozhraní. Uvedené rozhraní obsahuje následující prvky:

* Write
* Read
* Command
* Call

…. TODO: vypsat všechny prvky + příklady na C#

Protože je editor rozšiřitelný o objekty typu ILanguageDefinition, je možné přidat podporu editací pro libovolný jazyk, vyhovující uvedenému rozhraní.

### 3.6.2 Koncept EditTransformation

Již víme, že při interpretaci jsou ke každé metodě poznamenávány IInterpretedLine objekty, které reprezentují interpretovaný příkaz. Tyto objekty nesou informace o místě, kde se příkaz vyskytl a jaké instance byly příkazem ovlivněny. Dále nám zprostředkovávají informace o následujícím a předchozím příkazu.

Nad těmito IInterpretedLine definujeme transformace jako objekty splňující rozhraní IEditTransformation. To má následující položky:

* IPosition Map(IPosition) – transformace konkrétní pozice.
* IInterpretedLine Map(IInterpretedLine) – transformace celého interpretovaného příkazu.
* Apply() – zapíše současnou transformaci do zdrojového kódu.

V editoru je definováno několik standardních transformací, které mohou být využity tvůrci uživatelských rozšíření. Zde je výčet standardních transformací dostupných přes třídu StandardTransforamtions následujícími metodami:

* Write –umožňujíce zapsat data na zadanou pozici.
* Remove–odstraní data mezi zadanými pozicemi
* CreateObject –vytvoří objekt zadaného typu na zadané pozici
* GetShifting – korektní přesunutí interpretovaného příkazu za jiný interpretovaný příkaz. Pokud není možné korektní transformaci vytvořit, vrací null.

S výhodou pak využíváme skládání transformací, když potřebujeme například pro přidání argumentu do volání funkce nejprve přesunout několik řádků. Přeházení řádků provedeme pomocí Shifting transformace, kterou poté dáme jako výchozí transformaci pro volání StandardTransformations.Write.

### 3.6.3 Editace poskytované instancí

V průběhu interpretování mohou interpretery přidávat libovolné editace konkrétním instancím. K tomuto účelu slouží rozhraní IEditations, dostupné v IInstance.InstanceEditations. Díky němu můžeme přidat libovolnou editaci, která bude k dispozici v kontextovém menu instance zobrazené ve schématu kompozice. Pomocí IEditations objektu však můžeme přidávat i speciální editace pro akceptování instancí systémem drag&drop a odstranění instance z composition point.

Další možnosti editací závisí na dostupných rozšíření editoru. Podrobnější popis práce s editacemi bude uveden v kapitole zabývající se rozšířeními editoru.

### 3.6.3.1 Accept editace

Editace pro akceptování instance jsou v editoru využity při přesunování objektů ve schématu kompozice systémem drag&drop. Pro přidání takové editace je využíváno volání IEditations.AddInstanceAccepter, kterému zadáme jméno typu instancí které chceme přijímat. Dalším parametrem je pak objekt IInstanceAccepter, který je použit pro vlastní provedení editace.

Před tím, než umožníme instanci aby byla akceptována, odstraníme ji z jejího logického rodiče. Logickým rodičem zde označujeme instanci, která vykresluje své logické potomky. Příkladem logického rodiče může být CompositionContainer, který zobrazuje komponenty na nichž byla volána metoda ComposeParts. Ve schématu jsou tyto komponenty zobrazeny uvnitř CompositionContainer instance.

Vztahy rodiče a potomka jsou mezi instancemi definovány pomocí volání IEditations.AddChild a IEditations.RemoveChild. Díky nim je navíc označeno místo, kde byl potomek přidán, což nám umožní odebrat potomka předtím, než bude akceptován jinou instancí.

### 3.6.3.2 Remove editace

TODO: refactoring IRemoveProvider

### 3.6.7 Statické editace

Naproti editacím, které souvisí s konkrétní instancí, editor pracuje i s editacemi, které se v průběhu interpretování příliš nemění. Těmto editacím proto říkáme statické. Mezi obvyklé statické editace patří editace pro přidávání komponent do composition point. Další statické editace jsou pak přidávány uživatelskými typovými definicemi. Typické využití je přidání editace pro vytvoření objektu.

## 3.7 Komponentový model

TODO

### 3.7.1 Reprezentace komponent

Za komponenty jsou v editoru považávány takové instance, jejichž typ má hodnotu ComponentInfo různou od null. Tato položka pak slouží jako podklad pro zobrazování komponent ve schématu kompozice. Stejně tak mohou být uvedené informace zpracovány uživatelskými typovými definicemi, které na jejich základě mohou zobrazovat vztahy mezi importy a exporty komponent, případně mohou simulovat naplňování importů z dostupných exportů. IComponentInfo definuje následující položky:

* Exports – exporty definované na položkách komponenty
* SelfExports – exporty definované na celé třídě komponenty, exportována je celá komponenta
* Imports – importy definované na komponentě
* ImportingConstructor – konstruktor, který má být použit pokud je potřeba zkonstruovat komponentu.
* CompositionPoints- metody ozačené patřičným atributem jako composition point

Importy a exporty pak obsahují údaje o kontraktu, o skutečném typu importované/exportované položky, metadata a další údaje, které jsou zobrazovány uživateli ve schématu kompozice.

### 3.7.3 Naplnění importů

Editor neposkytuje žádné speciální rozhraní, které by provádělo naplnění importů z exportů. Místo toho dává uživatelským typovým definicím dostatečné údaje, aby mohly provést naplnění samy. Jedná se především o setter metody, které nastaví hodnotu importu, případně o importing konstruktor, přes který jsou importy nastaveny.

Simulace kompozice je však implementována v doporučených rozšíření. Jedná se o třídu CompositionEngine, která simuluje chování kompozice, tak jak probíhá v MEF. Dále poskytuje chybové výstupy, které se zobrazují ve schématu kompozice. Díky tomu je odstraňování chyb v kompozici usnadněno.

## 3.8 Uživatelské rozhraní

Jak jsme v analýze uvedli, je pro nás výhodné zobrazovat schéma kompozice na základě interpretace composition point. Seznam dostupných composition point v aktivním solution je získáván ze třídy ComponentManger, která navíc poskytuje událost vyvolanou při změně v tomto seznamu.

Po vybrání composition point uživatelem je spuštěna interpretace tak, že nejprve vytvoříme instanci z InternalType, na kterém byl definován. Tuto instanci inicializujeme pomocí patřičného bezparametrického konstruktoru. Následně je na instanci zavolána metoda, odpovídající vybranému composition point, s případnými argumenty, které byly získané z CompositionPoint atributu.

Výsledkem interpretace je seznam objevených instancí, pro které byla nalezena vhodná definice zobrazení. Tyto instance jsou pak zobrazeny ve schématu kompozice, které je překresleno, pokud dojde ke změně na libovolné závislosti objevené během interpretování.

### 3.8.1 Seznam dostupných composition point

V tomto seznamu jsou zobrazovány aktuálně dostupné composition point objevené v aktivním solution. Díky využití ComponentManger.ComponentChange je seznam obnovován, pokud je přidán nebo odebrán nějaký composition point.

Názvy composition point zobrazené uživateli jsou odvozeny od fullname příslušné metody. Ten je pak zkrácen tak, aby byl v seznamu unikátní, ale byl co nejkratší.

### 3.8.2 Vykreslování schématu kompozice

V průběhu interpretace jsme získali seznam instancí, pro které máme vhodnou definici zobrazení. Ze seznamu pak vybereme ty, které nemají žádného logického předka. Toto opatření je důležité, neboť logický předek se o vykreslení logického potomka musí postarat sám. Vybrané instance pak zobrazíme podle patřičných definic zobrazení.

Definice zobrazení mají při vykreslování dostupné prostředky, díky kterým mohou zobrazovat instance potomků, importy/exporty a také spojení mezi nimi. Jejich využití je blíže popsáno v kapitole 4.3.4.

#### 3.8.2.1 Vykreslení instance

Každá instance musí být vykreslena uvnitř prvku InstanceCanvas, který může definice zobrazení vytvořit za pomoci IDrawingConnector, který má k dispozici. InstanceCanvas pak umožňuje akceptování instancí při události drop. Dále spravuje informace o zobrazených importech/exportech a na jejich základě umožňuje zobrazování patřičných spojnic.

Vykreslení instance je prováděno voláním InstanceCanvas.DrawInstance. Nejprve je získána kresba z patřičné definice zobrazení. Pomocí této kresby je vytvořen objekt InstanceThumb, který mimo vlastního zobrazení kresby implementuje WPF události potřebné pro výpočet a zobrazení náhledu při události drag. InstanceThumb také řeší, zda je možné provést drop do InstanceCanvas nad kterým se nachází. Pokud se totiž nepodaří provést transformace zdrojového kódu, která by umožnila drag instanci uvolnit z jejího logického rodiče, není drop povoleno. Další nezbytná funkčnost InstanceThumb je zobrazování kontextového menu s editacemi příslušejícími zobrazované instanci.

TODO: lepší názvosloví pro drag drop

#### 3.8.2.2 Určování pozice zobrazených instancí

Výchozí pozice zobrazených instancí je určena tak aby byly řazeny přibližně do čtverců. Ddíky tomu není schéma příliš široké ani vysoké. Editor však umožňuje uživateli tuto výchozí pozici měnit.

Aby byla informace o umístění instance zachována i při opakované interpretaci composition point, využívá editor kolekci persistentních dat, indexovaných dle IInstance.ID. Zde jsou uloženy pozice instancí vzhledem ke svým rodičům.

#### 3.8.2.3 Importy, exporty a jejich spojení

Každý InternalCanvas registruje zobrazené instance a další InternalCanvas v nich vnořené. Díky tomu má přehled o JoinGlyph objektech, které reprezentují zobrazené importy a exporty. To umožňuje vykreslení spojnic mezi těmito objekty s využitím WPF.

### 3.8.3 Zamykání editací

Vzhledem k tomu, že editor sleduje změny zdrojových kódů až s určitým zpožděním, mohlo by dojít k nesprávným zápisům do zdrojových kódů. Tento problém je řešen zabráněním editoru v provedení editací po dobu od zaregistrované změny zdrojového kódu po zpracování registrovaných změn.

TODO: Editor upozorňuje na chybné operace, řeší privacy policy, hlídá velikost zabrané paměti a instrumentaci

TODO: Problém s vytvářením instancí v parserech, kvůli cachování!!!

TODO:ObjectModelManager.RunMethod, InterpretInline -> vysvětlit IInterpretedLine

# 4. Uživatelská příručka

## 4.1 Popis rozhraní editoru

TODO: obrázek uživatelského rozhraní s popisem ovládacích prvků

## 4.2 Použití editoru

TODO: Příklad vytvoření kompozice na konkrétním projektu

## 4.3 Rozšiřitelnost editoru

Aby bylo možné rozšiřovat schopnosti analýzy editoru, případně aby mohl být editor upraven pro konkrétní projekt, je koncipován s ohledem na rozšiřitelnost v následujících oblastech:

* Parsování – parsery jsou v editoru používány pro převod zdrojového kódu na instrukce interpretovatelné interpretery. Do prostředí editoru může uživatel přidat parser pro libovolný jazyk.
* Interpretování – instrukce přeložené v parserech jsou spouštěny v příslušném interpreteru. Výsledkem interpretování jsou pak informace sloužící pro zobrazení schématu kompozice a jeho následné editování.
* Definice typů – aby bylo možné definovat vlastní chování objektů vybraných typů, používá editor assembly Runtime. Ta získává typy právě z uživatelských typových definic. Díky tomu může uživatel definovat vlastní editace na objektech libovolného typu.
* Zobrazení objektů – objekty objevené při interpretaci mohou být zobrazeny ve schématu kompozice. Zda se objekt zobrazí, rozhoduje přítomnost definice zobrazení pro typ objektu. Uživatel tedy může definovat zobrazení téměř pro libovolný typ objektu.

Rozšiřitelnost je v editoru zajištěna technologií MEF. Veškeré rozšiřující služby jsou pak exportovány na základě odpovídajícího interface. Všechny potřebné interface jsou definovány v knihovně MEFEditor.ExtensionPoints.dll.

Abychom uživatele seznámili s principem rozšiřitelnosti editoru, vytvoříme několik ukázkových rozšíření. Dále popíšeme, které služby jsou pro uživatelská rozšíření dostupná.

V následujících kapitolách předpokládáme pro vývoj rozšíření .NET projekt typu Class Library s přidanými referencemi na knihovny MEFEditor.ExtensionPoints a System.ComponentModel.Composition. Všechny knihovny, ze kterých mají být získána uživatelská rozšíření musí být umístěné ve složce Extensions v kořenovém adresáři editoru.

### 4.3.1 Uživatelské parsery

Vzhledem k tomu, že by bylo nevýhodné interpretovat zdrojové kódy přímo na základě zdrojového textu, existuje v editoru možnost zdrojové kódy před interpretováním přeložit do snáze interpretovatelných instrukcí. Parsování navíc může probíhat v několika krocích. Výsledek jednoho parseru může být vstupem pro druhý parser. Rozparsované instrukce si editor uchováva dokud nedojde ke změnám na zdrojových kódech, ze kterých vznikly, případně dokud se nezmění typ použitý během parsování.

Doporučený postup při psaní parseru pro editor tedy je oddělení syntaktického a sémantického zpracování zdrojového kódu. Díky němu nemusí při změně na typech použitých při parsování syntaktický parser opakovat svou práci.

Aby mohl být uživatelský parser importován do editoru, musí splňovat rozhraní IParser, podle kterého musí být také exportován. IParser obsahuje následující položky:

* string[] Languages – obsahuje seznam jazyků instrukcí, které mohou být tímto parserem zpracovány.
* IInvokeInfo ParseInline(IParsingServices services, IInvokeInfo info) – metoda provádějící parsování „jednořádkových“ instrukcí, jejichž výsledkem je výraz vracející objekt.
* IInvokeInfo ParseMethod(IParsingServices services, IInvokeInfo info) – metoda provádějící parsování celých metod. Výsledkem jsou instrukce jejichž vykonání simuluje spuštění metody.

Výsledkem parsování je objekt splňující rozhraní IInvokeInfo. Jazyk, kterým budou výsledné instrukce zapsané záleží na parseru. Podle jazyku je však rozhodnuto, který parser/interpreter je dále dostane ke zpracování. Pokud není žádný nalezen, ohlásí editor chybu.

Parser, který zpracovává instrukce přímo z editoru, je vybrán na základě jazyka uvedeného v příslušném CodeElement.Language, získaném ze zdrojového kódu. Více informací o CodeElement zde: <http://msdn.microsoft.com/en-us/library/envdte.codeelement.aspx>. ParseMethod je využíváno pro zpracování metod získaných ze zdrojových kódů. Instrukce které dostane jsou tedy typu CodeFunction. ParseInline je však používána pro získávání instancí z CodeAttribute a také pro získávání instancí defaultních hodnot parametrů či datových položek tříd. Formát instrukcí pro defaultní hodnoty datových položek tříd jsou zde: <http://msdn.microsoft.com/en-us/library/envdte.codevariable.initexpression.aspx> a pro parametry zde: <http://msdn.microsoft.com/en-us/library/envdte80.codeparameter2.defaultvalue.aspx>

Popsání implementace celého parseru by bylo nad rámec této práce. Proto zde uvedeme pouze služby, které dává editor parserům k dispozici. Tyto služby jsou přístupné přes rozhraní IParsingServices objektu, se kterým jsou metody ParseMethod a ParseInline zavolány. Pokud není nějaká služba při parsování dostupná, její příslušející objekt má hodnotu null.

Přístup k rutinám, souvisejícím s kontextem, ve kterém bylo parsování spuštěno nám dává IParsingServices.Context. Kontext zohledňuje typovou definici, které se parsování týká. Typ vytvořený z této typové definice je pak přístupný v IContext.ContextType. Vyhledávání typů podle jména také podlého kontextu typové definice. Jedná se především o dostupné namespace, které jsou zohledněny v metodě IContext.FindType(). Ta se na základě daného jména typu pokusí najít odpovídající fullname se zohledněním dostupných namespace. Dle tohoto fullname pak vrací odpovídající IType reprezentaci typu. Všechny typy, které by ovlivnili výsledek hledání pomocí FindType jsou automaticky přidány jako závislost vytvářeného IInvokeInfo. Díky těmto závislostem jsou instrukce zneplatněny pokud je například přidán typ, který nebyl při parsování dostupný. To pak může vést k překreslení schématu kompozice.

Pro vytváření instancí slouží InstanceCreator. Pří parsování by však měl však být využíván pouze pro vytváření instancí primitivních typů, neboť výsledek parsování může být pro interpretování použit několikrát díky cachování. Zbylé instance musí být vytvářeny až v průběhu interpretování.

Ohlášení chyb, způsobených nesprávnými instrukcemi je prováděno vyhozením vyjímky ParsingException. Parametry při vyhazování vyjímky umožňují specifikovat důvod proč byla vyjímka vyhozena, případně kde ve zdrojovém kódu k vyjímce došlo.

### 4.3.2 Uživatelské interpretery

Interpretování instrukcí se v editoru provádí pomocí interpreterů. Uživatelské interpretery musí splňovat rozhraní IInterpreter. Podle tohoto rozhraní musí být také exportován. IInterpreter obsahuje tyto položky:

* string[] Languages – obsahuje seznam jazyků, které může interpreter interpretovat
* IInstance InterpretInline(IInvokeInfo,IExecInfo) – spustí interpretaci „jednořádkových“ instrukcí. Vrací instanci, kterou mají tyto instrukce vytvořit.
* IInstance InterpretMethod(IInvokeInfo,IExecInfo) – spustí interpretaci instrukcí metody. Vrací instanci, která je návratovou hodnotou interpretované metody.

Implementace interpreteru velmi závisí na instrukcích, které zpracovává. Proto zde uvedeme pouze služby, které má interpreter od editoru k dispozici a způsob, jakým mají být tyto služby využívány. Tyto služby jsou přístupné přes rozhraní IExecInfo objektu, se kterým jsou metody InterpretInline a InterpretMethod zavolány. Interpreter má k dispozici všechny služby dostupné parserům. Ty jsme však popsali v předchozí kapitole. Zabývat se tedy budeme pouze službami specifickými pro interpretery. Pokud některé služby nejsou pro konrétní interpretaci dostupné, mají příslušné objekty hodnotu null.

Pro přístup k instanci, na které byly vyvolány instrukce, slouží IExecInfo.This. Nastavování datových položek této instance lze provést pomocí IExecInfo.SetData. Jiným způsobem se datové položky v instancích nastavit nedají. Pro snadný přístup k datovým položkám This instance pak slouží IExecInfo.GetData.

Informace o argumentech interpreter získává z IExecInfo.CallInfo. V získaném objektu jsou instance, které byly předány interpretované metodě jako argumenty. ICallInfo však ještě obsahuje ICallEditations objekt, který dává přístup k editacím nad místem volání interpretované metody.

Údaje o prostředí, ve kterém je analyzovaná aplikace spouštěna jsou dostupné v IExecInfo.EnvironmentInfo. Díky tomu může interpreter získat například údaje o složce, ve které je analyzovaná aplikace spouštěna.

Poslední službou dostupnou v rámci interpretování, kterou si popíšeme je IExecInfo.Analyzing. Pomocí této služby poskytuje interpreter editoru informace používané při editacích. Aby mohl editor provádět přehazování řádků zdrojového kódu, musí mu interpreter oznámit pozici jednotlivých řádků. To se provádí pomocí IAnalyzing.SetLine(). Řádkem zde rozumíme takovou část kódu, která se dá samostatně přesouvat. Například v C# budeme za řádek považovat každý neblokový příkaz ukončený středníkem. Blokové příkazy musí editor označit celé pomocí IAnalyzing.SetBlock(). U příkazu if to bude samotný příkaz spolu s if-blokem a else-blokem. Uvnitř bloků je pak opět používáno označování řádků pomocí SetLine.

Pokud interpreter některé instrukce nevykonává v pořadí, v jakém jsou zapsané ve zdrojovém kódu, musí při zpracování řádku, kde dochází ke skoku zavolat IAnalyzing.SetJump(). Tím je zabráněno přesunování řádků ve složitých situacích.

Pro každý zpracovaný řádek interpreter určuje se kterými instancemi se na něm pracovalo. Díky tomu můžeme při editacích rozhodnout, zda přehazování řádků nezpůsobí nechtěné změny v sémantice metody. Toto označování se provádí pomocí IAnalyzing.AffectedInstance().

Na řádce, kde je instance přiřazena do nějaké proměnné, musí intepreter zavolat IAnalyzing.BeginScope(). Pro každou proměnnou, jejíž instance je z proměnné odstraněna je třeba zavolat IAnalyzing.EndScope(). Dle těchto informací dokáže editor získat platný název instance při editacích.

Je důležité uvedená volání IAnalyzing provádět před započetím práce s instancemi, kterých se týkají. Tyto informace jsou totiž editorem interně využívány pro vytváření editací nad instancemi.

### 4.3.3 Uživatelské definice typů

Editace, které objekt nabízí jsou definovány typovou definicí typu objektu. Abychom uživateli dovolili definovat vlastní editace, umožňuje editor nahrát uživatelské definice typů v podobě rozšíření do assembly Runtime. Při vyhledávání typu v různých assemblies jsou pak upřednosťnovány typy z Runtime. Díky tomu můžeme například nahradit typové definice získané z MEF knihoven vlastními typovými definicemi, které již umožní potřebné editace.

Uživatelské typové definice musí splňovat interface ITypeDefinition, dle kterého také musí být exportovány. Pro usnadnění vytváření typových definic je v MEFEditor.ExtensionPoints.dll připravena abstraktní třída RuntimeDefinition, která spolupracuje se standardními rozšířeními editoru.

Pro ukázku tedy vytvořme uživatelské rozšíření, které bude demonstrovat editace použitelné v editoru. Toto rozšíření bude přepisovat chování třídy MEFEditor.Diagnostic.

V projektu vytvářené rozšiřující knihovny nejprve vytvoříme třídu DiagnosticDefinition, odděděnou od třídy RuntimeDefinition. Zároveň vytvoříme implementaci abstraktní metody PreloadRoutines, která bude sloužit pro kontrolu požadavků, které budeme mít na assembly Runtime.

Dále budeme potřebovat instrukce, které definují chování metod reprezentovaného typu. RuntimeDefinition nám umožňuje využít interpreter, který dovede spouštět metody ve tvaru IInstance method(IExecInfo info). Instrukce simulovaných metod tedy budou metody naší typové definice.

[Export(typeof(ITypeDefinition))]

public class DiagnosticDefinition : RuntimeDefinition

{

public DiagnosticDefinition()

{

//here will be initialization of rawname and methods

}

private IInstance \_ctor(IExecInfo info){

//instructions for constructor

}

private IInstance \_start(IExecInfo info){

//instructions for Start method

}

private IInstance \_stop(IExecInfo info){

//instructions for Stop method

}

private IInstance \_accept(IExecInfo info){

//instructions for Accept method

}

public override bool PreloadRoutines(ITypeLoadingConnector connector, IAssembly runtime)

{

//checking routines

}

}

Takto vypadá naše typová definice, připravená k implementaci.

Implementaci začneme inicializací RuntimeDefinition.\_rawName. Jelikož reprezentovaná třída není generická, bude \_rawName shodné s jejím fullname. V konstruktoru dále definujeme metody, které bude naše typová definice poskytovat. Pro každou metodu tedy vytvoříme IInvokeInfo pomocí RuntimeDefinition.Create(), specifikující instrukce metody a poté ji přidáme do seznamu dostupných metod pomocí RuntimeDefinition.AddMethod().

\_rawName = typeof(Diagnostic).FullName;

var ctor = Create(\_ctor, \_rawName);

ctor.IsConstructor = true;

var start = Create(\_start, Constants.Type\_Void);

var stop = Create(\_stop, Constants.Type\_Void);

var accept = Create(\_accept, Constants.Type\_Void,"object");

accept.SetParametric();

AddMethod("Diagnostic", ctor);

AddMethod("Start",start );

AddMethod("Stop", stop);

AddMethod("Accept", accept);

Tělo konstruktoru nyní vypadá následovně. Upozorněme na nutnost označit nastavení ctor.IsConstructor, označující že se jedná o metodu konstruktoru a na accept.SetParametric(), které nastaví poslední parametr metody accept jako parametrický.

Nyní se již můžeme pustit do implementace metod, které budou simulovat chování typu MEFEditor.Diagnostic. Konstruované instanci pomocí instrukcí \_ctor inicializujeme datové položky s využitím info.SetData(). Dále přidáme InstanceAccepter, který bude přijímat objekty typu System.Object při drag&drop editaci ve schématu kompozice vytvořením metody Accept. V metodách \_start, \_stop pouze odsimulujeme zapnutí a vypnutí časovače.

private IInstance \_ctor(IExecInfo info){

//Stopwatch from System.Diagnostic

info.SetData("watch", new Stopwatch());

info.SetData("instances", new List<IInstance>());

//Create accepter, which can accept instances of typ System.Object

var accepter=new InstanceAccepter(info.This,"Accept");

//created accepter will be assigned into instance

info.This.Editations.AddInstanceAccepter("System.Object", accepter);

return info.This; //We should return created instance from constructor

}

private IInstance \_start(IExecInfo info){

//Get initialized stopwatch

var watch = info.GetData("watch") as Stopwatch;

//start time measuring

watch.Start();

//recommended void value

return info.Void;

}

private IInstance \_stop(IExecInfo info){

var watch = info.GetData("watch") as Stopwatch;

//stop time measuring

watch.Stop();

return info.Void;

}

Implementace několika metod, z vytvářené typové definice.

Pokračujme implementací metody \_accept. Před zpracováním argumentů poskytnutých metodě nastavíme InstanceAccepter objektů typu System.Object na poslední volání metody Accept. Díky tomu budou objekty přijímány přidáním argumentu do tohoto posledního volání.

private IInstance \_accept(IExecInfo info){

//shortcut for callEditations

var callEdits = info.CallInfo.CallEditations;

//we want accept instances on last Accept call

if (callEdits != null) //test if call editations are available

{

//remove old instance accepter

info.This.Editations.RemoveAccepter("System.Object");

//create new argument accepter

var argAccepter=callEdits.ArgumentAccepter("System.Object");

//assign argument accepter with System.Object types

info.This.Editations.AddInstanceAccepter("System.Object", argAccepter);

}

Po zkontrolování zda jsou dostupné editace pro současné volání, na toto volání nastavíme přijímání objektů.

Teď můžeme začít se zpracováním argumentů. Ty jsou dostupné v IExecInfo.CallInfo. Získané argumenty označíme jako logické potomky instance na které bylo volání provedeno. Tím jednak upozorníme editor, že tyto logické potomky budeme zobrazovat uvnitř objektu Diagnostic, zároveň umožníme jejich odstranění metodou drag&drop díky poskytnutým RemoveProvider. O samotné zobrazení se však musí postarat definice zobrazení, kterou implementujeme v příští kapitole.

//get stored instances

var instances = info.GetData("instances") as List<IInstance>;

//index of proceeded argument

int argIndex=0;

//add all accepted instances

foreach (var arg in info.CallInfo.Arguments)

{

//store instance

instances.Add(arg.Instance);

//call editations may not be available

if (callEdits != null)

{

//remover which can remove argument at specified index

var remover=callEdits.ArgumentRemover(argIndex);

//set as optional because removing only this argument, not whole call

remover.IsOptional = true;

//add child, which can be removed via drag&drop in composition scheme

info.This.Editations.AddChild(arg.Instance, remover);

}

++argIndex;

}

Zpracování argumentů získaných při volání metody Accept. Upozorněme, že parametrické argumenty nejsou v uživatelských typových definicích uloženy v poli, jak je tomu například v C#. Důvodem je snazší přístup k těmto argumentům.

Nyní nám zbývá vrátit návratovou hodnotu, která bude určovat počet dosud akceptovaných instancí. Použijeme pro to IExecInfo.InstanceCreator(), který nám umožní vytvoření instance typu int ze skutečného .NET objektu.

//return instance, representing number of accepted instances

return info.InstanceCreator.CreateInstance("int", instances.Count);

}

Prvním parametrem v CreateInstance je typ instance, kterou chceme vytvořit. Druhým parametrem jsou potom data, která chceme použít pro inicializaci.

Abychom však měli jistotu, že typová definice int, kterou používáme opravdu umožňuje nahrání z těchto dat, musíme přidat test do metody PreloadRoutines.

TODO, přidat test do PreloadRoutines – zatím tu nejsou, neboť API pro testy se bude ještě měnit

### 4.3.4 Uživatelské zobrazení objektů

Objekty jsou ve schématu kompozice zobrazovány podle toho, zda pro ně máme vhodnou definici zobrazení. Naimplementujme tedy vzorovou definici zobrazení pro typovou definici vytvořenou v předchozí kapitole. Zobrazovaný objekt bude uvnitř sebe vykreslovat instance získané voláním Accept. Pro demonstraci vykreslovacího rozhraní pak spojíme importy a exporty všech komponent, které budou mezi instancemi obsaženy. U importů navíc zobrazíme demonstrační chybová hlášení.

Rozhraní, které musí definice zobrazení splňovat je IDrawingDefinition. Podle tohoto rozhraní ji také exportujeme pro použití v editoru. Rozhraní obsahuje property FullName, které určuje jméno typu zobrazitelné definicí. Definice zobrazení však musí podporovat i zobrazení všech odvozených typů. Zobrazení objektu se získává voláním GetDrawing(), které dostane jako parametr vykreslovanou instanci a IDrawingConnector, zprostředkovávající vykreslovací rozrhraní editoru. GetDrawing vrací FrameWork element objekt, který je zobrazen ve schématu kompozice.

Vzhledem k tomu, že zobrazování je ve schématu kompozice prováděno pomocí WPF technologie, musíme do projektu přidat reference na knihovny PresentationCore a PresentationFrameWork, které nám zpřístupní potřebné WPF služby.

Implementaci definice zobrazení začneme vytvořením třídy DiagnosticDrawing. Property FullName bude vracet plné jméno typu MEFEditor.Diagnostic. Vlastní kreslení pak bude implementováno v metodě GetDrawing.

[Export(typeof(IDrawingDefinition))]

public class DiagnosticDrawing : IDrawingDefinition

{

public string FullName{get{return typeof(Diagnostic).FullName;}}

public FrameworkElement GetDrawing(IInstance instance, IDrawingConnector connector)

{

//here will be created drawing for composition scheme

}

}

Třída připravená pro implementaci metody GetDrawing, která vykreslí zadanou instanci.

Pro vytvoření WPF kresby instance máme k dispozici IDrawingConnector, který obsahuje metody pro vytvoření import/export konektorů. Ty můžeme využít pro znázornění importů a exportů na vlastních zobrazeních komponent. V tomto příkladě však využijeme metodu IDrawingConnector.CreateInstanceCanvas, která nám vrátí objekt rozšiřující běžný WPF canvas o rutiny umožňující zobrazit instance a spojení mezi importy a exporty.

V metodě GetDrawing nejprve získáme datové položky, jejichž data budeme zobrazovat. Po získání datových položek provedeme inicializaci WPF objektů, které zobrazíme ve výsledné kresbě.

//drawing will be returned for display in composition scheme

var drawing = new StackPanel();

drawing.Background = Brushes.Green;

//get data from drawed instance

var watch = instance.GetData("watch") as Stopwatch;

var instances = instance.GetData("instances") as List<IInstance>;

//output from watch

var caption = new TextBlock();

caption.Text = string.Format("{0}ms",watch.ElapsedMilliseconds.ToString());

//this canvas can be used for displaying accepted instances

var canvas = connector.CreateInstanceCanvas();

//define layout

drawing.Children.Add(caption);

drawing.Children.Add(canvas);

//collected import/export points

var imps = new List<IJoinPoint>();

var exps = new List<IJoinPoint>();

Inicializační kroky metody GetDrawing.

Nyní už můžeme přejít k vykreslení instancí získaných metodou Accept. Instance vykreslíme pomocí volání IInstanceCanvas.DrawInstance(), které zohlední typ instance vybráním správné definice zobrazení. Abychom mohli zobrazit spojení mezi importy a exporty, musíme nejprve získat příslušné IComponentInfo. To nalezneme v reprezentaci typu, se kterým byla instance vytvořena. Z importů/exportů v IComponentInfo pak získáme objekty typu IJoinPoint, které reprezentují konektor na zobrazené komponentě. IJoinPoint objekty následně využijeme pro zobrazení spojů mezi importy a exporty. Pro dostupné importy navíc nastavíme demonstrační chybová hlášení.

foreach (var inst in instances)

{

//draw each instance in collection according to its drawing definition

canvas.DrawInstance(inst);

var componentInfo = inst.CreationType.ComponentInfo;

if (componentInfo == null)

//no component info is available

continue;

//collect imports and exports

foreach (var imp in componentInfo.Imports)

imps.Add(canvas.GetJoinPoint(inst, imp));

foreach (var exp in componentInfo.Exports)

exps.Add(canvas.GetJoinPoint(inst, exp));

}

//join collected imports and exports

foreach (var impPoint in imps)

{

//shows messages in ToolTip bubble

impPoint.SetMessages("Import error", "Import warning");

foreach (var expPoint in exps)

//draw joining line between import and export

canvas.AddJoin(impPoint, expPoint);

}

return drawing;

}

Vykreslení instancí a následné pospojování každého importu s každým exportem. Na konci metody GetDrawing je pak vrácena vytvořená kresba.

Tímto jsme dokončili implementaci vlastní definice zobrazení. Objekt vrácený metodou GetDrawing bude zobrazen ve schématu kompozice a bude podporovat vkládání a odebírání ostatních objektů zobrazených ve schématu pomocí drag&drop mechanismu.

### 4.3.5 Uživatelské ILanguageDefinition

### 4.3.6 Standardní rozšíření

### 4.3.7 Doporučená rozšíření

# 5. Závěr

V úvodních kapitolách jsme uvedli cíle, kterých jsme v rámci práce chtěli dosáhnout. Cíle byly následující:

* Editor bude integrován do Microsoft Visual Studia 2010
* Umožní provádět analýzu zdrojových kódů rozpracované .NET aplikace otevřené v Microsoft Visual Studiu 2010.
* Na základě provedené analýzy přehledně zobrazí zjištěné schéma kompozice.
* Umožní uživateli v zobrazeném schématu provádět editace
* Dovede reagovat na uživatelské změny zdrojového kódu patřičným překreslením schématu
* Upozorní na případné chyby v kompozici
* Umožní pomocí rozšíření měnit způsob vykreslení schématu kompozice
* Bude rozšiřitelný o schopnosti analýzy zdrojových kódů a MSIL knihoven

Implementovaný editor je plně integrován do prostředí Microsoft Visual Studia 2010 ve formě pluginu. Umožňuje analyzovat zdrojové kódy získané v otevřeném solution. Na základě analýzy pak dokáže vykreslit schéma kompozice, ve kterém umožní uživateli provádět editace. Editor je schopen zaznamenat změny ve zdrojových kódech a na jejich základě v případě potřeby překreslí schéma kompozice. To může být navíc překresleno i při změnách souborů knihoven a složek, kterých se schéma kompozice týká.

Pomocí uživatelských rozšíření můžeme kompletně změnit vzhled zobrazovaného schématu kompozice. Editor je také rozšiřitelný o uživatelské interpretery a parsery, díky nimž je možné zlepšovat schopnosti analýzy zdrojových kódů i MSIL knihoven. Vzhledem k tomu, že na vytváření editací se podílejí právě interpretery a parsery, je editor rozšiřitelný i o možnosti nabízených editací.

V průběhu vývoje editoru se ukázalo, že není výhodné, aby na chyby ve schématu kompozice upozorňoval sám editor. Místo toho byla tato funkčnost dána na starost rozšiřujícím modulům.

Následující moduly, zmíněné v kapitole Cíle projektu, jsou implementovány v knihovně RecommendedExtensions.dll.

* Modul pro parsování jazyka C#
* Modul pro interpretaci sémantického stromu
* Objektový model nutný pro analýzu základních MEF tříd
* Modul pro zobrazení významných MEF objektů

Umožňují práci a zobrazování schémat kompozice nad projekty psanými jazykem C#. Obsažená rozšíření objektového modelu jsou schopná simulovat průběh MEF kompozice a díky tomu dokáží zobrazit vztahy mezi komponentami, případně chyby, které se při kompozici objevily.

Výsledkem této práce je plugin Visual Studia 2010, který umožňuje zobrazení a editaci schématu MEF kompozice, získaného analýzou zdrojových kódů otevřeného solution. Editor však není díky značné rozšiřitelnosti omezen pouze na využití pro MEF. Dodáním patřičných rozšíření umožňuje vizualizaci a editaci libovolných objektů a vztahů mezi nimi, které byly získány na základě interpretace zdrojového kódu nebo MSIL kódů referencovaných knihoven.

# 6. Reference

[Vzor: Seznam použité literatury je zpracován podle platných standardů. Povinnou citační normou pro bakalářskou práci je ISO 690. Jména časopisů lze uvádět zkráceně, ale jen v kodifikované podobě. Všechny použité zdroje a prameny musí být řádně citovány.]

**Seznam použité literatury**

[Vzor: Tabulky v bakalářské práci, existují-li.]

**Seznam tabulek**

[Vzor: Použité zkratky v bakalářské práci, existují-li, včetně jejich vysvětlení.]

**Seznam použitých zkratek**

[Vzor: Přílohy k bakalářské práci, existují-li (různé dodatky jako výpisy programů, diagramy apod.). Každá příloha musí být alespoň jednou odkazována z vlastního textu práce. Přílohy se číslují.]

**Přílohy**

[Vzor: Pevná zadní deska bakalářské práce – **není součástí elektronické verze**.]