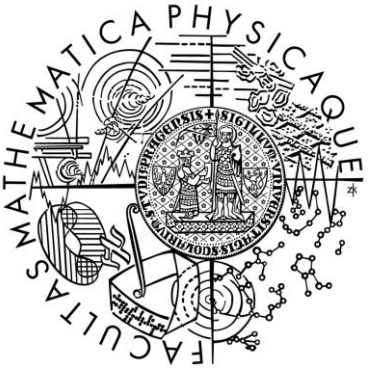
Univerzita Karlova v Praze

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

Poděkování

Rád bych poděkoval vedoucímu práce Mgr. Pavlu Ježkovi za připomínky a nápady, bez kterých by tato práce nemohla vzniknout. Také bych chtěl poděkovat mojí rodině za podporu, kterou mi věnují.

Prohlašuji, že jsem tuto bakalářskou práci vypracoval 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#. V těchto aplikacích pomáhá odhalovat chyby kompozice 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: Managed Extensibility Framework allows development of component-based .NET applications. However relations between components can be quite complex. So it would be advantageous to visualise these relations to user to simplify the development of component-based applications and to provide their editing. This thesis provides a solutioin in form of an editor plugin for the Microsoft Visual Studio 2010 allowing to view a composition schema based on analysis of a source code. The editor allows user to edit the generated composition schema and is able to correct the original source code according to user made changes. Possibilities of the analysis and the offered editing actions are influenced by users extensions, the editor is extensible with. As part of this thesis, we implemented the editor with extensions allowing to use it in application projects written in C# language. It helps to detect composition errors in these applications and simplifies changes in the application component architecture.

Keywords: MEF, editor, Visual Studio 2010, component-based applications.

Obsah

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)* [1] a *Managed Extensibilty Framework (MEF)* [2]. 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 objekty 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.

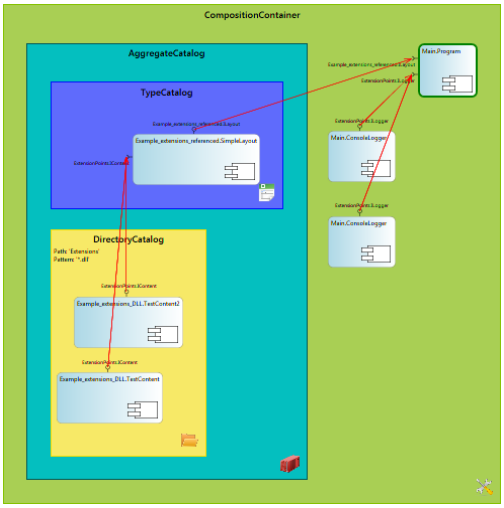
Už tedy víme, jak se komponenty definují. Nyní se podívejme na způsob práce s nimi. Zmiňovali jsme, že MEF umožňuje běhovou rozšiřitelnost aplikací. To znamená, že aplikace musí při svém běhu rozhodnout, jaká rozšíření chce nahrát. MEF ji dává k dispozici několik připravených katalogů, které zajišťují vyhledávání komponent v souborech MSIL [3] 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* [4] a jeho vizuální podoba *Visual MEFX* [5]. 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:

```
class Program
{
    [Import]
    ILayout compositionResult=null;

    [ImportMany]
    ILogger[] loggers=null;

    [CompositionPoint]
    void Compose()
    {
        var consolelogger = new ConsoleLogger();
        var consolelog = new ConsoleLogger();
        var typecatalog = new TypeCatalog(typeof(SimpleLayout));
        var directorycatalog = new DirectoryCatalog(@"Extensions");
        var aggregatecatalog = new AggregateCatalog();
        var compositioncontainer = new CompositionContainer(aggregatecatalog);
        aggregatecatalog.Catalogs.Add(typecatalog);
        aggregatecatalog.Catalogs.Add(directorycatalog);
        compositioncontainer.ComposeParts(consolelog, consolelogger, this);
    }
```



Metoda Compose provádí kompozici, která je přehledně znázorněna na schématu vpravo.

Náhled na schéma kompozice umí zobrazit nástroj *MEF Visualizer Tool* [6], 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.

Cílem 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 kompozice.
- 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 doporučených 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 uveden příklad použití editoru na konkrétním projektu. Pro uživatele, který píše vlastní rozšíření editoru, naimplementujeme ukázková rozšíření, která vysvětlí princip rozšiřitelnosti editoru.

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.

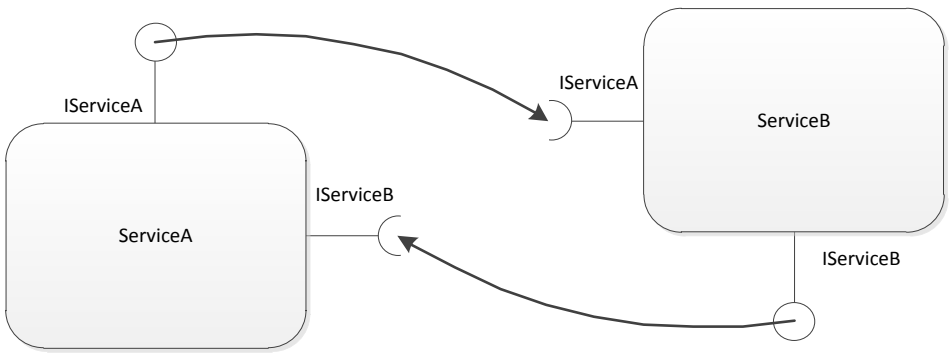
Již víme, že definované importy jsou při kompozici naplněny z dostupných exportů. Naplnění probíhá tak, že MEF provede přiřazení do datové položky importu, případně použije setter importující property. Přiřazení i volání metod musí probíhat na zkonstruovaných objektech. Přesto však MEF nabízí katalogy, 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. Jestliže takový konstruktor není nalezen, 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. V kódu ho určíme následovně:

```
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. Příklad cyklické závislosti naznačuje obrázek:



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

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 adrese [7]. 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 přidat jedinou komponentu bez dalších změn v aplikaci.

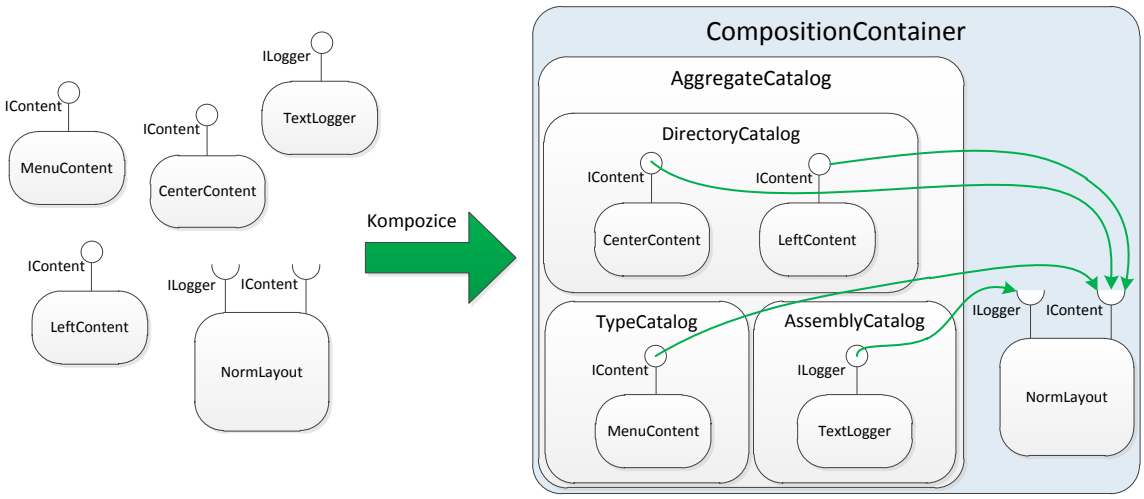
Jak z výše uvedeného vyplývá, aplikace může být sestavena z množství komponent, které se mohou časem měnit, můžou pocházet z různých zdrojů, nebo dokonce mohou záviset na konkrétních nastaveních uživatele. Zamysleme se tedy, jak by mělo vypadat schéma kompozice, které by přehledně vypovídalo o struktuře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



Obrázek 1.2-1 Při kompozici aplikace musí uživatel zajistit načtení komponent z rozličných katalogů. Pro jejich použití v CompositionContainer navíc musí všechny katalogy vložit do AggregateCatalog.

Jak vidíme na obrázku 1.2-1, uživatel 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 existujícími 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 zobrazit je 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 s žádnými MEF objekty nepracuje, mohou mít na kompozici zásadní význam. Takovou situaci demonstruje následující obrázek:

```
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 specifikovat seznam významných metod, které bychom si pamatovali v nějakém konfiguračním souboru. Toto řešení by však nebylo přenosné ve zkompilované aplikaci. Navíc bychom se

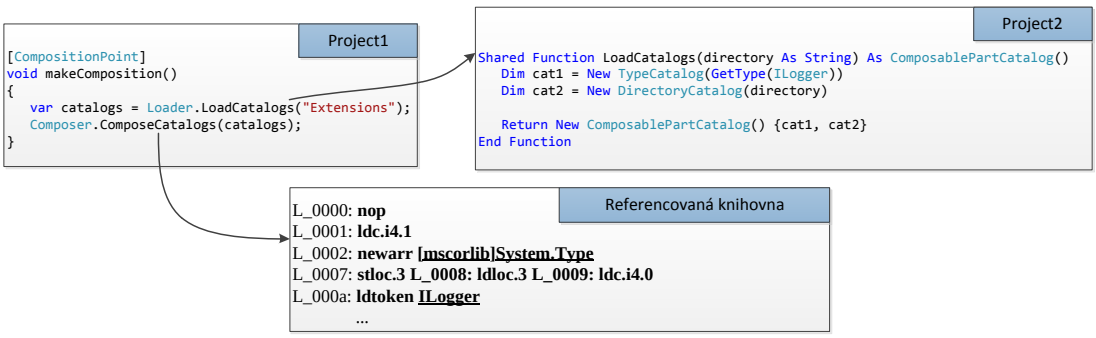
dostali do problémů se synchronizací názvů tříd a metod popisujících, který *composition point* chceme spustit, neboť názvy se ve zdrojovém kódu mohou často měnit.

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 třídy.

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. Příklad takového volání demonstruje následující obrázek:

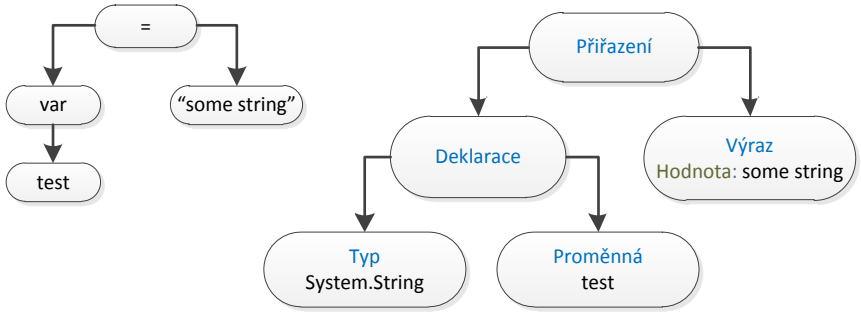


Obrázek 1.4-2 Z metody makeComposition je volána metoda LoadCatalogs v Project2 psaná ve Visual Basicu a ComposeCatalogs ze zkompilované knihovny, dostupné pouze v MSIL instrukcích

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. Následuje příklad takových stromů:

```
var test = "some string";
```



Obrázek 1.4-3 Ukázka syntaktického (vlevo) a sémantického (vpravo) stromu. Uvedené stromy vznikly 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 dostaneme 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, neboť Reflection nepodporuje odstranění typu z aplikační domény. Pro nás to však 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. Využijeme reprezentaci, kterou používá .NET a je kompatibilní s Common Language Specification [20]. Tato reprezentace je popsána na adrese [8]. Podle ní budeme například binární operátor + chápat jako metodu `op_Addition`.

Stejným způsobem dokážeme vyřešit implicitní/explicitní typové konverze. Implicitní konverzi z typu A na typ B obstará metoda `B op_Implicit(A)`. Na stejném principu funguje i .NET reprezentace konverzních 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. Indexery budeme převádět do metod `get_Item` a `set_Item` s příslušným počtem parametrů. Poslední objektovou operací, kterou budeme řešit pomocí metod jsou inicializátory. Inicializátor bude prováděn pomocí metody `op_Initializer`, která bude přijímat proměnlivý počet argumentů typu `object`.

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 generické 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. Nakonec budeme *typové definice* získávat z uživatelských rozšíření. Jejich vytváření je popsáno v kapitole 4.1.4 .

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. Typy budeme tedy vytvářet 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* [9], 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*.

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.

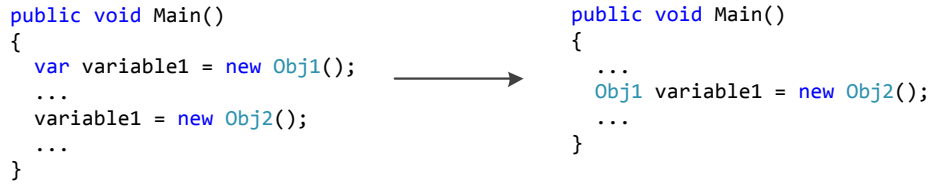
1.7 Vytváření editací

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. Následuje 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.

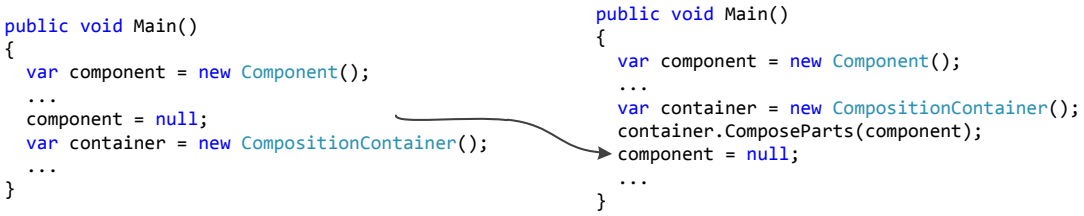
- **Přidání komponenty do CompositionContainer** – Pokud do kontejneru nebyla ještě žádná kompnenta přidána, přidáme ji vytvořením volání funkce `ComposeParts`. Pokud již nějaké komponenty obsahuje, přidáme ji jako nový parametr do zmíněného volání.
- **Vytvoření nového objektu** – Deklarujeme proměnnou, 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 neodstraňujeme deklaraci. Potom by se totiž mohlo stát, že musíme proměnnou deklarovat znovu, na místě jejího dalšího použití, jak ukazuje obrázek:



Obrázek 1.7-1 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, jak je znázorněno dále:



Obrázek 1.7-2 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. Aby byly editace prováděné přesunováním objektů ve schématu kompozice pro uživatele intuitivní, budeme změny provádět až za místem volání posledních metod zúčastněných objektů. To nám zaručí, že editujeme objekty ve stavu, v jakém jsou vidět na schématu kompozice. Kvůli tomu však nemusí být

možné jednoduše pozměnit kód tak, aby se projevila požadovaná editace a přitom nedošlo k žádným vedlejším efektům. Tento případ je znázorněn zde:

```
public static void Compose()
{
    var container = new CompositionContainer();
    var component = new Example();

    component = component.returnNull();
}
```

V tomto případě nemůžeme přidat component do container, neboť poslední volání metody na component je returnNull. Přidat ji do container bychom museli až za tímto voláním, jenomže tam už má proměnná component hodnotu null.

Stejně tak nebude možné, aby editor přesunul objekt platný pouze v jedné metodě do objektu platného v jiné metodě. V takových případech dá editor 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 v konkrétní metodě, 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 překreslit. 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*. Zjišťovat, 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 příliš 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

Propojení editoru s Visual Studiem je zprostředkováno přes assembly vytvořenou z projektu VsPackage. Ten slouží pro vývoj rozšíření Visual Studia. Obsahuje předpřipravené třídy, pomocí kterých přidáme položku do menu Visual Studia pro spuštění našeho editoru. V rámci připravených tříd získáme nezbytnou službu pro interakci s Visual Studiem - `EnvDTE.DTE`, popsanou v následující kapitole.

Zkompilováním projektu VsPackage dostaneme *vsix* soubor, který nainstaluje náš editor jako rozšíření Visual Studia. Více informací o vytváření rozšíření pomocí VsPackage je uvedeno zde [10].

2.2. EnvDTE.DTE

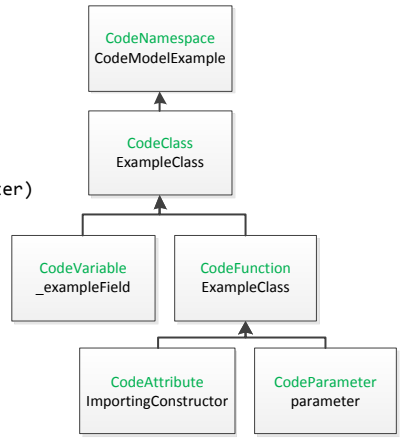
Základní přístup ke službám, které Visual Studio nabízí, probíhá přes rozhraní `DTE`. Pro potřeby editoru budeme z `DTE` využívat události týkající se uživatelských akcí a informace o otevřeném solution a jeho projektech. Za aktivní solution budeme z pohledu editoru považovat solution, dostupné v položce `DTE.Solution`. Z něj získáváme seznam projektů a pro každý projekt pak jeho zdrojové kódy a reference na knihovny.

O přidávání a odebírání prvků, případně o změně aktivního solution, nás informují události dostupné v `DTE.Events`. Více informací o použití objektu `DTE` na adrese [11].

2.3. Code Model

Ke zdrojovým kódům projektů aktivního solution editor přistupuje pomocí *Code Model*. Nad každým souborem se zdrojovým kódem udržuje Visual Studio stromovou strukturu složenou z objektů odvozených od `CodeElement` [16], které reprezentují dostupné *typové definice*, jejich metody, atributy a další elementy zdrojového kódu. Na obrázku je znázorněna tato struktura:

```
namespace CodeModelExample
{
    class ExampleClass
    {
        int _exampleField;
        [ImportingConstructor]
        public ExampleClass(int parameter)
        {
          ...
        }
    }
}
```



Obrázek 2.3-1 Příklad Code Model reprezentace zdrojového kódu jmenného prostoru CodeModelExample.

Díky *Code Model* můžeme zjišťovat, jaké *typové definice* jsou v solution dostupné. Nedokážeme však přistupovat k příkazům jednotlivých metod, neboť `CodeFunction` objekt nám nabízí pouze text zdrojového kódu metody. Proto je nutné, aby editor dokázal parsovat zdrojový text metod sám.

Při použití *Code Model* musíme dávat pozor na nedeterministické chování jednotlivých elementů. Spolu s tím, jak se mění zdrojové kódy, mění se i odpovídající `CodeElement` objekty. Některé změny však mohou celý `CodeElement` zneplatnit, takže přístup k jeho členům vyvolá vyjímku. Editor musí tyto změny včas registrovat, aby zabránil nesprávnému použití `CodeElement` objektů.

Další nepříjemnou vlastností je to, že se nedozvíme, zda můžeme například získat seznam předků nějaké třídy, aniž bychom vyvolali vyjímku. Tato situace nastává, když je mezi předky třídy uveden identifikátor, který nedefinuje žádný dostupný typ. Vyvolávání těchto vyjímek pak může zpomalovat prohledávání *Code Model*. Podrobné informace o použití *Code Model* jsou k dispozici zde [12].

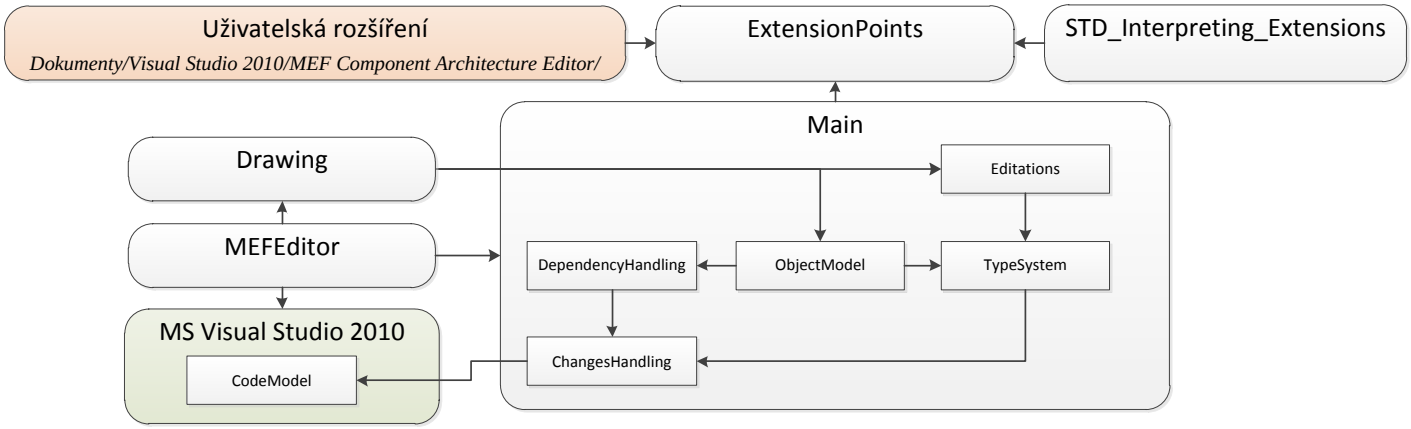
2.4. Reakce na události vyvolané uživatelem

Aby mohl editor překreslovat schéma kompozice na základě akcí prováděných uživatelem ve zdrojovém kódu, zachytává události poskytované Visual Studiem. Nejvýhodnější by se mohlo zdát použití událostí definovaných v `Events2.CodeModelEvents`, které informují o přidávání, odebírání a změnách `CodeElement` objektů. Jejich použití se však ukázalo jako nespolehlivé. Události například nejsou vyvolány vždy, když dojde ke změnám zdrojových kódů, případně je ohlášena změna na nesprávném `CodeElement`. Editor proto musí využívat událost `Events.TextEditorEvents.LineChanged`, díky které získává údaje o provedených změnách a sám podle nich určuje, které `CodeElement` byly změněny.

Další události, které je nutné sledovat souvisí se změnou aktivního solution, nebo změnou jeho struktury. Tyto události jsou definovány ve členech `SolutionEvents` a `SolutionItemsEvents` objektu `DTE.Events`.

3 Implementace editoru

3.1 Struktura



Obrázek 3.1-1 Znázornění struktury implementace editoru

V následující části se budeme zabývat implementací editoru, která je dostupná v solution z přílohy [A]. V rámci této práce byla také implementována *doporučená rozšíření*, která si popíšeme v kapitole 4.3. K jejich implementaci i k implementaci samotného editoru existují dokumentace automaticky generované ze zdrojových kódů. Tyto dokumentace se nachází v příloze [D].

Na obrázku *3.1-1* je zachycena základní struktura editoru. Po spuštění Visual Studia je zavedena VsPackage assembly `MEFEditor`, obstarávající propojení s Visual Studiem a nahrání standardních rozšíření z assembly `STD_Interpreting_Extensions`. Dále poskytuje uživatelské rozhraní assembly `Drawing` a nakonec spouští hlavní assembly `Main`. Ta po svém startu nahraje rozšíření ze složky uživatelských rozšíření. Assembly `ExtensionPoints` slouží pro definici rozhraní, která musí uživatelská rozšíření používat.

Po nahrání rozšíření do editoru je napojen modul `ChangesHandling` pomocí objektu `EnvDTE.DTE`, na události týkající se změn aktivního solution, zásahů do zdrojových kódů a přidávání/odebírání projektů. 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.

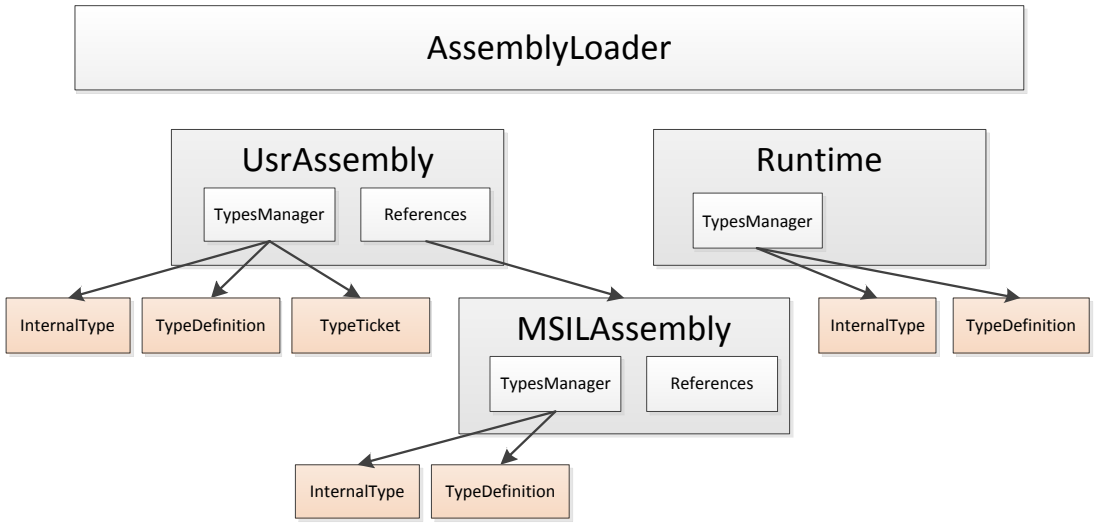
Díky načtení typového systému získá `Drawing` seznam *composition point*, který je zobrazen uživateli v rozhraní editoru. Aktivováním některého *composition point* v uživatelské nabídce, dojde k jeho spuštění modulem `ObjectModel`. Na základě získaných výsledků je pak assembly `Drawing` schopna zobrazit 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 úprav 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 příkazů. Tyto informace potom může interpreter nabídnout zpracovávaným *instancím*, které jsou díky tomu schopné nezávisle na interpretovaném jazyku vytvořit editace zobrazované uživateli. Ne všechny editace však souvisejí s konkrétní *instancí*. Tyto editace jsou řešeny modulem `Editations`. Typickou editací poskytovanou tímto modulem je vytvoření nového objektu v *composition point*.

3.2 Typový systém

Namespace: *MEFEditor.Main.TypeSystem*



Obrázek 3.2-1 Hierarchie používaná pro správu typů v typovém systému

Obrázek 3.2-1 znázorňuje jakým způsobem jsou spravovány typy v našem typovém systému. 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` snadno vyhledávat komponenty v objevených .NET knihovnách.

V souvislosti s pojmenováváním typů zavedeme několik definic. Pro plné jméno typu budeme používat označení *fullname*. Pokud pomocí *fullname* označujeme generický typ, všechny jeho parametry musí být dosazené opět v podobě *fullname*. Parametry generického typu však nejsou omezeny pouze na názvy typů, jak je tomu v .NET. Parametrem může být i libovolná hodnota neobsahující speciální znaky a znaky pro *menší než*, *větší než*, *čárka*. Této vlastnosti využíváme například pro definici pole, která je popsáno v kapitole 3.2.3.

Alternativní název k *fullname* nazýváme *alias*. Dalším typem jména, které rozlišujeme, je *rawname*. Jedná se o *fullname*, ale bez dosazených parametrů. Posledním označením, které budeme potřebovat je *signature*. Získáme ho tak, že z *rawname*/*fullname* odstraníme parametry. Příklad je uveden zde:

```
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*. Z *TypeTicket* budeme vytvářet *TypeDefinition*, která 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 *InternalType*. 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 s využitím `Mono.Cecil`[13].

Posledním typem assembly je `Runtime`. Tato assembly slouží pro nahrávání *typových definic* ze standardních a uživatelských rozšíření. Při vyhledávání typů pak mají nejvyšší 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 *Code Model* 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. Tyto změny poskytuje `CodeModelManager`, jenž je napojen na modul `ChangesHandling`. Dostává tedy informace o změnách `CodeElement` objektů, na základě kterých dokážeme odebírat a přidávat typy v `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*. Voláním metod, které pocházejí z dirty *typových definic* vznikají *dirty instance*, popsané v kapitole 3.3.6. 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 knihovny `Mono.Cecil`. Pro tyto účely jsou mnohem vhodnější než .NET `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* stádia reprezentace typu. Nahrávání typů v `MSILAssembly` nevyužívá *TypeTicket*, neboť ten by musel obsahovat referenci na `Mono.Cecil.AssemblyDefinition`, ze které pochází. Tato reference pak znemožní *garbage collectoru* [15] 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. Díky tomu při opětovném dotazu na stejnou assembly nedojde k jejímu vytváření, ale pouze k předání odkazu na již registrovanou assembly.

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*. To 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 konkrétní změnu *struktury typu*, která 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 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 parametrech generických typů a nakonec `ITypeDefinition` popisující vytvářený typ.

Tvorba `InternalType` probíhá ve dvou krocích. Při volání konstruktoru jsou inicializovány položky `FullName`, `Alias` a `Assembly` pomocí `TypeSystemInfo` a `ITypeDefinition`. 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 ale 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`. 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 `ITypeDefinition`. 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, voláním `InterpretInline` na statické třídě `ObjectModelManager`, která bude podrobně

popsána v kapitole 3.3.1. Touto interpretací získáme reprezentaci atributu, která nám poskytne informace o exportech, importech a dalších prvcích definovaných na typu.

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.

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.addMethods`. Při volání jí předáme seznam implementací metod, definovaných v *typové definici*. Tyto metody se pak pokoušíme spojit s deklaracemi abstraktních a virtuálních metod předků. Pro každou spojenou metodu přidáme do `InternalType` metodu indexovanou deklarací získanou od předka. Tím zajistíme fungování systému virtuálních a abstraktních metod.

Pokud se nepodaří vytvářenému typu najít implementace všech abstraktních metod, nastaví si příznak `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 obvyklé 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 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 rozhraním `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` 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 budou dohledatelná přes stejné `IMethodInfo`.

3.2.3 Speciální typy

V našem typovém systému existuje několik typů, které mají odlišné chování od běžných typů v .NET. Nejvýznamnějším takovým typem je `System.Array<Type,Dimension>`, využívaný pro reprezentaci polí. Díky němu nemusíme používat žádné speciální reprezentace pro pole, neboť se jedná o obvyklý generický typ. Následujícímu zápisu v C# `string[,] variable`, by odpovídal zápis `System.Array<System.String,2> variable`, v našem typovém systému.

Dalším speciálním typem, který však nemá obdobu v .NET je typ `System.Proxy<Type>`. Objekty tohoto typu se tváří jako by byly vytvořené z `Type`. Rozdílem ale je, že každá metoda volaná na takovém objektu je přesměrována na metodu, kterou uvedeme při konstrukci objektu. Tento systém je využitelný pro vytváření objektů z interface, které potřebujeme pro naplnění metadat importů v průběhu simulace kompozice.

Pro reprezentaci návratových hodnot `void` funkcí využíváme objekty vytvořené z typu `System.Void`. Posledním speciálním typem je `System.Null`, jehož objekty reprezentujeme `null` hodnotu. Objekty typu `System.Null` mají speciální chování z hlediska dědičnosti. Tváří se, že jsou potomky libovolného typu, a že každý typ je potomkem `System.Null`. Této vlastnosti je využíváno při rozhodování, zda můžeme volat nějakou funkci, na místech, kde nemáme typovou informaci o všech argumentech. Typickým případem je volání explicitní *composition point* metody, kde jsou jako vstupní argumenty zadány hodnoty `null`.

Podrobnosti týkající se používání speciálních typů uživatelskými rozšířeními jsou v kapitole 4.2.1.

3.2.4 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 nebo zaniklých typech.

Nové typy jsou do `TypesManager` objektu přidávány metodami `SetTicket` a `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` díky modulu `ChangesHandling`.

Rozhodnout o platnosti *TypeDefinition* je však složitější, neboť závisí na změnách metod, datových položek nebo seznamu předků. Všechny události, které by mohly zneplatnit *TypeDefinition*, jsou proto pokryty v seznamu závislostí, dostupného v položce `ITypeDefinition.Dependencies`. Díky obsluze těchto závislostí `TypesManager` zjistí, kdy byla *TypeDefinition* zneplatněna. Pokud k zneplatnění dojde, je *TypeDefinition* odstraněna. Stejně tak jsou odstraněny všechny z ní vytvořené *InternalType* reprezentace. Podrobnější popis závislostí je uveden v kapitole 3.5.

Platnost *InternalType* je jednak omezena platností *TypeDefinition*, ze které byl vytvořen, ale také změnou *InternalType* libovolného předka. Všechny závislosti, které má *InternalType* proti *TypeDefinition* navíc, jsou uvedeny v položce `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.

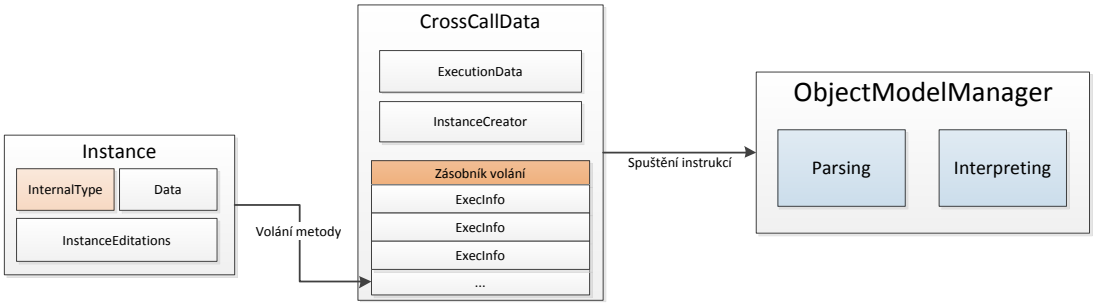
Nad typy, které jsou v `TypesManager` obsaženy, můžeme vyhledávat pomocí `TypesManager.FindType`. Tato metoda vyhledá pro zadaný *fullname* odpovídající *InternalType*. Vyhledávání probíhá tak, že k *fullname* zkusí najít typ v již vytvořených *InternalType*. Pokud není nalezen žádný záznam, je *fullname* převedeno na *signature* a hledá se pro něj příslušná *TypeDefinition*. Když ani ta není nalezena, je poslední možností jak typ získat 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 `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

***Namespace**: MEFEditor.Main.ObjectModel*



Obrázek 3.3-1 Průběh zpracování volání metody na instanci, objektovým modelem

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*. Volání metod probíhá podle schématu na obrázku 3.3-1. 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*, podrobně popsaných v kapitole 3.3.6.

3.3.1 ObjectModelManager

Základem objektového modelu je schopnost vytvářet *instance* a volat na nich metody. Proto jsou zde spravovány parsery, interpretery a jazykové definice, které má editor k dispozici. S jejich využitím pak `ObjectModelManager` implementuje metody `RunMethod` a `InterpretInline`, které slouží ke spouštění zadaných instrukcí.

`RunMethod` je využívána *instancemi*, ke spouštění instrukcí metod. Zajišťuje přeložení instrukcí pomocí dostupných parserů a cachovaných instrukcí. Přeložené instrukce jsou poté interpretovány patřičným interpreterem. Pro vlastní spuštění metody je nutné do `RunMethod` zadat aktivní `CrossCallData` objekt, který reprezentuje zásobník volání. Všechny *instance* vytvořené v průběhu interpretované metody budou pro vlastní volání metod opět používat stejný `CrossCallData` objekt. Proto může být využíván pro hlídání instrumentace a pro registrování všech závislostí, které se při interpretaci vyskytnou.

V průběhu interpretace metody, vytvářejí interpretery objekty splňující rozhraní `IInterpretedLine`, které nesou údaje o interpretovaných instrukcích. Každý takový objekt reprezentuje ucelenou část zdrojového kódu. Tu je možné přesunovat mezi ostatními částmi určenými `IInterpretedLine`, pokud je průnik jejich `IInterpretedLine.Borders` prázdný. Například v jazyce C# části kódu odpovídají jednotlivým řádkům odděleným středníky. `IInterpretedLine` jsou editorem využívány pro tvorbu editací. Jejich využití je blíže popsáno v kapitole 3.6.2.

Pro potřeby editoru nevystačíme pouze s interepretováním metod na *instancích*. Abychom mohli vytvářet *instance* z atributů nebo z defaultních hodnot proměnných či parametrů, implemetuje `ObjectModelManager` metodu `InterpretInline`. Tato metoda očekává instrukce definující nějakou *instanci*. Bližší popis těchto instrukcí a očekávaných výsledků interpretace je popsán v kapitole 4.1.2.

3.3.2 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. 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. Tuto kontrolu je možné provést v metodě `PreloadRoutines` rozšiřující *typové definice*.

Příkladem buď vytváření *typové definice* pro `DirectoryCatalog`. V konstruktoru tohoto katalogu 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.

3.3.3 Volání metod na instancích

Volání metod na *instanci* je prováděno přes `IInstance.CallMethod`. Tato metoda dostane jako parametr `IMethodInfo`, určující kterou metodu chceme volat. Dalším parametrem je `ICallInfo`, obsahující hodnoty argumentů a editace dostupné pro toto volání. Podrobná implementace editací bude popsána v kapitole 4.1.4.

Díky tomu, že již při hledání vhodné `IMethodInfo` byla zohledněna virtuálnost a přetížení metod, můžeme správnou implementaci v podobě `IMethod` 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. Toto `ExecInfo` je vloženo na spodek zásobníku volání, v objektu `CrossCallData`. Samotné spuštění se provede pomocí volání `RunMethod` na `ObjectModelManager`. Ten provede rozparsování metody a její následnou interpretaci. `RunMethod` vrací návratovou hodnotu interpretované metody, případně vyvolá vyjímku, pokud interpretace/parsování selže. 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.

3.3.4 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. Jednotlivé způsoby vytvoření *instance* jsou popsány zde:

Přímé vytvoření

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

bezparametrický konstruktor, je přirozené vytvořit *instanci* jeho voláním. *Vstupní třída* však nemusí bezparametrický konstruktor obsahovat. 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.

Vytvoření voláním konstruktoru

Pro některé jazyky mají konstruktory sémanticky odlišný význam od obyčejných metod. V našem objektovém modelu se však jedná o obyčejnou metodu, označenou příznakem `IsConstructor`, který může být využit při parsování a interpretování. Zda byl konstruktor na *instanci* zavolán, poznáme podle příznaku `IsConstructed`. Tento příznak je využívaný například v *doporučených rozšířeních* pro simulaci kompozice, kde umožňuje rozhodnout, které komponenty jsou zkonstruované.

I přesto, že se konstruktor chová jako obyčejná metoda, je možné toto chování přeci jen odlišit v interpreterch či parserech, pokud by to bylo potřeba. Přestože tento způsob vytváření *instancí* není z pohledu objektového modelu nijak zajímavý, obsahuje `InstanceCreator` metodu `ConstructInstance`, která v jednom kroku provede přímé vytvoření *instance* a následné zavolání konstruktoru na vytvořené *instanci*.

Vytvoření injektováním ze zadaných dat

Rozhraní `ITypeDefinition` vyžaduje dvojici metod `CanLoadFrom` a `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.

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.

3.3.5 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 *sdílených instancí* navíc typicky probíhá těsně před prvním použitím, když interpreter potřebuje získat *sdílenou instanci*, aby na ní mohl zavolat nějakou metodu, což odpovídá chování statických tříd v .NET.

3.3.6 Koncept dirty instancí

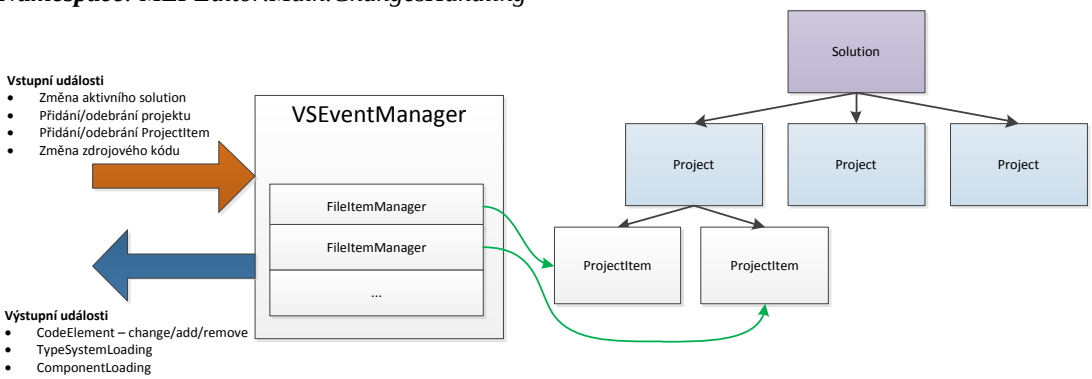
Vzhledem k tomu, že korektní reprezentace všech objektů použitých při interpretaci *composition point* by byla výpočetně velmi náročná, umožníme *instancím* nacházet se v dirty stavu. *Dirty instance* mohou vzniknout jako výsledek volání metody, jejíž instrukce jsou označené dirty příznakem. Tento příznak 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

***Namespace**: MEFEditor.Main.ChangesHandling*



Obrázek 3.4-1 Práce modulu ChangesHandling je založená na obsluze událostí získaných od Visual Studia. Tyto události následně upravuje pro použití v ostatních modulech.

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`, jehož základem je statická třída `VSEventManager`, která je znázorněna na obrázku 3.4-1.

K událostem, jenž jsou sledovány, patří otevření/zavření solution a přidávání/odebírání projektů. Obě tyto události způsobí znovunahrátí typového systému. 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 obstarává `VSEventManager`.

3.4.1 Změna aktivního solution

Změny aktivního solution jsou indikovány událostmi `Opened` a `AfterClosing` objektu `SolutionEvents`. Při otevření nového solution musí editor načíst celý typový systém. Z projektů v solution nejprve vytvoří patřičné `UsrAssembly` 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 umožněno 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 `AfterClosing` je nejprve zkontrolováno, zda editor stále nenačítá předchozí solution. Pokud ano, tak je načítání přerušeno. Poté dojde k uvolnění všech dat, která editor pro solution používal, z paměti. Od této doby je editor neaktivní, dokud není otevřeno další solution.

3.4.2 Změny v projektech aktivního solution

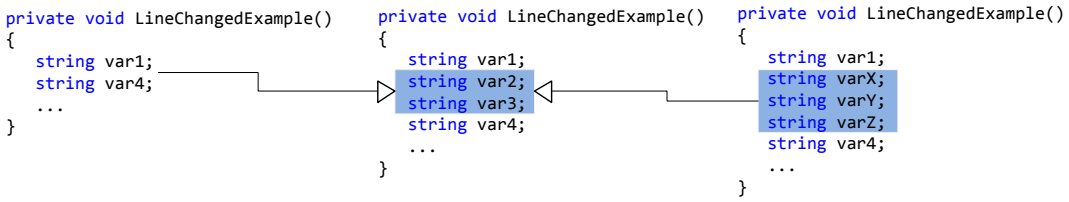
Visual studio umožňuje odebírání/přidávání projektů do aktivního solution. Tyto změny vyvolají události `ProjectAdded/ProjectRemoved` objektu `SolutionEvents`. Pokud k nim dojde, je znovunačten celý typový systém, tak jako by bylo zavřeno a znovu otevřeno aktivní solution. Vzhledem k tomu, že projekty nejsou měněny často, je toto řešení výhodnější, než složité simulace přidávání a odebírání patřičných `UsrAssembly` 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 musí 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 seznamy referencovaných knihoven 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 označeného textu po provedení změny. Abychom zjistili, jak dlouhý úsek původního textu změně odpovídá, musíme si pro každý soubor pamatovat celkovou délku před změnou. Proč nestačí pouhá znalost konce a začátku změny demonstruje následující obrázek.



Obrázek 3.4-2 Uprostřed vidíme metodu, kterou vytvoříme dvěma různými způsoby. Vložením řádků do metody vlevo nebo přepsáním řádků metody vpravo. Událost LineChanged je pro obě změny stejná. Rozdíl mezi nimi však zjistíme z rozdílů celkových délek zdrojových souborů.

Získané 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.

Ze změn zdrojových kódů získáváme následující události:

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*.

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 `OnRemove` na statické třídě `DependencyManager` prohlášeny za odebrané. Editor má díky tomu dostatečné informace pro odebírání *typových definic* z typového systému.

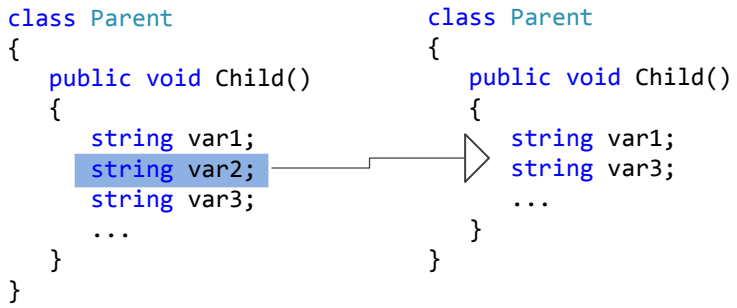
Přejmenování CodeElement

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.

Změna těla CodeElement

Pro události o 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 uvnitř nějakého `CodeElement`. Taková změna například pro aktuálně zobrazený *composition point* znamená, že musíme aktualizovat schéma kompozice.

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

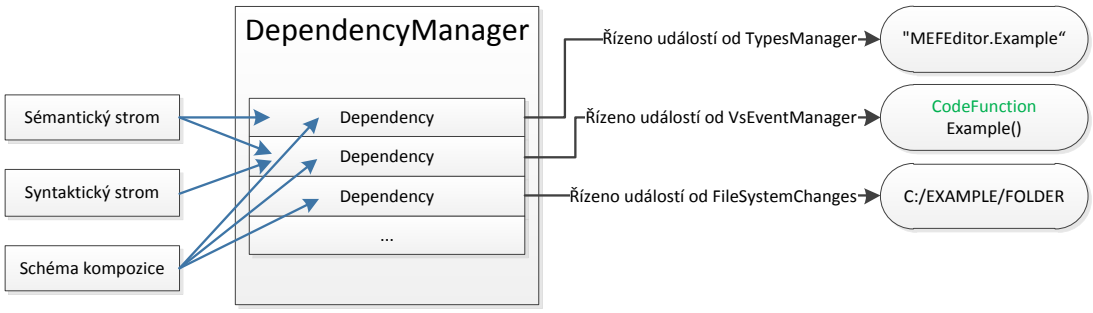


Obrázek 3.4-3 Uvedená změna je považována za změnu těla funkce Child. Třída Parent změněná není, neboť se celá změna odehrává uvnitř jejího potomka.

`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

***Namespace**: MEFEditor.Main.DependencyHandling*



Obrázek 3.5-1 V modulu DependencyHandling je implementován DependencyManager, využívaný pro správu závislostí

S využitím událostí, které nám poskytuje modul `ChangesHandling`, implementujeme závislosti na *typových definicích*, zdrojových kódech metod a dalších cílových objektech. Princip závislostí prezentuje obrázek 3.5-1. 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 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, spouštěné při jejich změnách.

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.

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 patřičné výsledky parserů. 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*.

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 používá. Pro názvy těchto typů pak vytváříme závislosti, které výsledek parsování zneplatní.

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

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 k tomuto účelu využívá systémové prostředky [15].

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 kódů, 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

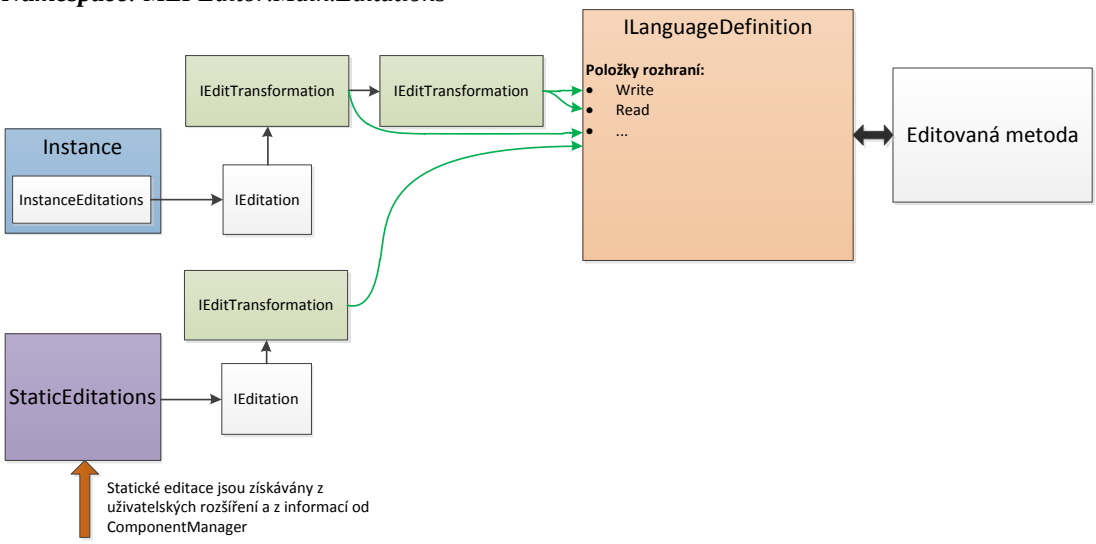
Assemblies získané z projektů aktivního solution žádné závislosti nedefinují. Pokud dojde k nějaké změně na projektu, je znovunačten celý typový systém. `MSILAssemblies` jsou však závislé na souboru, ze kterého byly nahrány a také na souborech referencovaných knihoven. Tyto závislosti jsou pak využívány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

3.6 Editace

***Namespace**: MEFEditor.Main.Editations*



Obrázek 3.6-1 Systém statických editací a editací poskytovaných instancí je založen na transformacích kódu, které své změny zapisují přes objekt jazykové definice.

V editoru jsou editace řešeny s ohledem na nezávislost na konkrétním jazyku. To je umožněno díky jazykovým definicím, splňujícím `ILanguageDefinition` rozhraní, které nám dává základní operace pro práci se zdrojovými kódy. Princip tvorby editací pomocí jazykových definic je uveden na obrázku 3.6-1.

Samotná editace 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 `ComposeParts` voláním na `CompositionContainer` objektu.

V prostředí editoru 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. Pomocí `ILanguageDefinition` můžeme vytvářet volání metod, properties, deklarace proměnných a další operace, které se nám hodí pro zápis editací do zdrojových kódů.

Editor je rozšiřitelný o objekty `ILanguageDefinition`, proto je snadné přidat podporu editací libovolného jazyka, který vyhovuje uvedenému rozhraní. Podrobnosti o způsobu použití jazykových definic jsou v kapitole 4.1.3.

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`** – umožňuje 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 `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 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 4.1.4. V rámci editoru rozlišujeme následující druhy editací:

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 *fullname* 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 jeho 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*.

Write editace

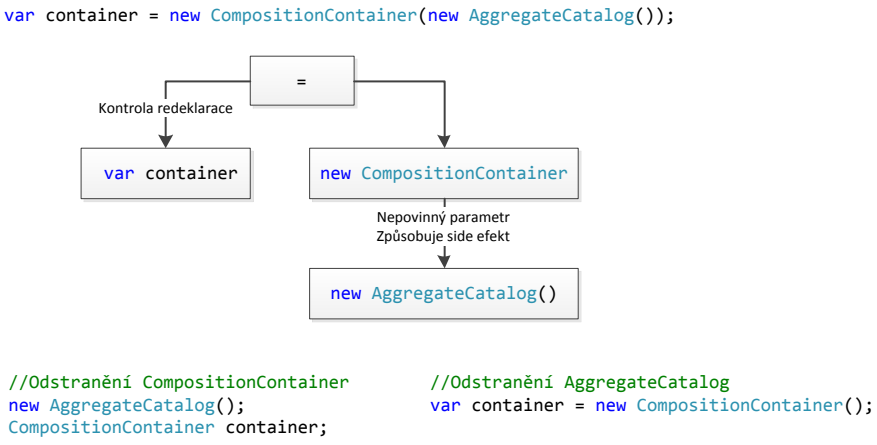
Zapisování argumentů do existujících volání je prováděno pomocí `IWriter` objektu získaného voláním `ICallEditations.ArgumentWriter`. Volání je zadán index argumentu, který chceme zapisovat. Pokud je index větší než index posledního argumentu, je do volání korektně přidán nový argument. V opačném případě je přepsán argument na pozici zadané indexem.

Ze získaného `IWriter` je možné vytvářet editace, které způsobí změnu/přidání argumentu na základě dialogů zobrazovaných uživateli. Příkladem může být `DirectoryCatalog`, který s dostupnými *doporučenými rozšířeními* nabízí editaci na změnu vzoru pro vyhledávání knihoven. Tato editace je vytvořena na základě `IWriter` objektu pro patřičný argument.

Remove editace

Pro odstraňování částí zdrojového kódu slouží v editoru objekty splňující rozhraní `IRemoveProvider`. Každý takový objekt nese pozici ve zdrojovém kódu, kterou může odstranit. Dále pak obsahuje pozici, kterou můžeme využít pro přesunutí částí zdrojového kódu z odstraňovaného místa, například kvůli zachování nějakého side efektu. `IRemoveProvider` může obsahovat několik potomků a nejvýše jednoho rodiče. Tato hierarchie je spolu s příznakem `IsOptional` a `HasSideEffect` využívána pro odstraňování nepovinných argumentů. Odstraníme-li totiž povinný argument z funkce, musíme spolu s ním odstranit i celé volání funkce. Pokud však nastavíme argumentu příznak `IsOptional`, je z funkce odstraněn pouze tento argument.

`IRemoveProvider` není využíván pouze pro odstraňování funkcí a argumentů, ale také pro odstraňování celých instancí. *Instance* je odstraněna tak, že odstraníme místo jejího vzniku a všechna volání, která na ní byla provedena. Při tomto procesu však může dojít ke smazání deklarace nějaké proměnné. `IRemoveProvider` tedy obsahuje položku `AfterRemove`, která obsahuje rutiny provedené po smazání všech `IRemoveProvider` objektů spojených s odstraňovanou *instancí*. V rámci těchto rutin je možné provést redeklarování proměnné, pokud je to potřeba. Konkrétní využití je znázorněno na obrázku:



Obrázek 3.6-2 Příklad hierarchie IRemoveProvider objektů a jejich příznaků. Volání konstruktoru AggregateCatalog má příznak HasSideEffect, neboť vytváří nový objekt. V rámci volání konstruktoru CompositionContainer je však nepovinným parametrem.

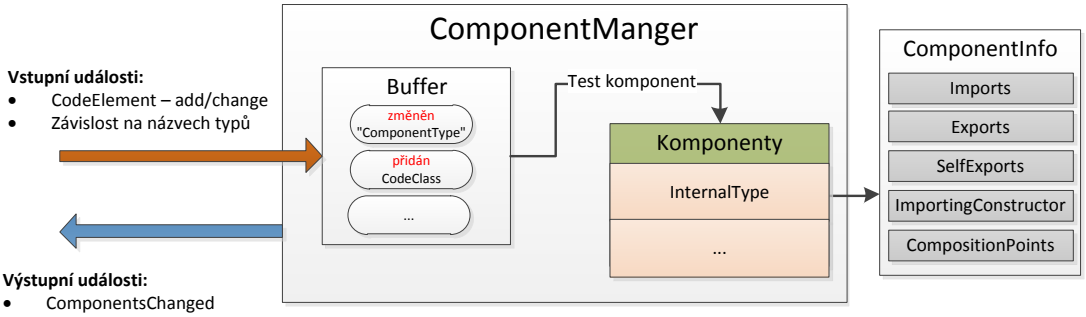
3.6.4 Statické editace

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.

Statické editace jsou zprostředkovány třídou `StaticEdtations`. Zde jsou drženy veškeré dostupné *statické editace* a také je zde prováděno filtrování těchto editací. Například editace pro vytvoření komponenty v nějakém *composition point* je možná pouze tehdy, pokud komponenta pochází ze stejné assembly jako com*position point*, nebo je v knihovně referencované touto assembly a je navíc označena jako `public`. Další filtrování pak probíhá na základě dostupnosti bezparametrického konstruktoru.

3.7 Komponentový model

***Namespace**: MEFEditor.Main.ComponentModel*



Obrázek 3.7-1 Správu komponent provádí třída ComponentManager, která využívá události o změnách typových definic

Informace o komponentách editor získává z *typových definic*. Z těchto informací je vytvořen objekt splňující rozhraní `IComponentInfo`, které zpřístupňuje údaje o atributech importů, exportů, dostupných *composition point* a další údaje využitelné pro simulaci kompozice.

Objevování komponent ve zdrojových kódech je znázorněno na obrázku 3.7-1. Správu komponent zajišťuje třída `ComponentManager`. Ta registruje změny na *typových definicích*. Při každé zjištěné změně spustí test `IsComponent`, objektu `ComponentTools`, na změněnou *typovou definici*. Dle výsledku test se rozhodne zda byla přidána/odebrána komponenta. Na základě toho upraví seznam dostupných komponent, zobrazovaný v uživatelském rozhraní.

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 rozšiřujícími *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 položkách komponenty
- **`ImportingConstructor`** – konstruktor, který má být použit pokud je potřeba zkonstruovat komponentu.
- **`CompositionPoints`** – metody ozačené patřičným atributem jako *composition point*

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

3.7.2 Vyhledávání komponent

O vyhledávání komponent ve zdrojových kódech otevřeného solution se stará třída `ComponentManager`. Díky modulu `ChangesHandling` je informována o změnách `CodeElement` *typových definic*. Proto může kontrolovat, zda změněná *typová definice* splňuje podmínky pro komponentu. Pokud je splňuje, je získán *InternalType* z této definice, který je přidán do seznamu dostupných komponent. V opačném případě je zkontrolováno, zda související typ nebyl přidán do komponent již dříve, aby mohl být odstraněn.

Testování komponent může být u velkých projektů zejména při otevření solution časově náročné. Kdyby `ComponentManager` vytvářel typy pro objevené komponenty ještě před načtením celého typového systému, mohlo by se stát že není dostupný předek typu komponenty. Vytvořený typ by proto musel být opraven poté co by byla přidána *typová definice* předka. To by však zbytečně zpomalovalo načítání typového systému. `ComponentManager` proto počká, dokud není typový systém načten celý a až poté je spuštěno testování komponent.

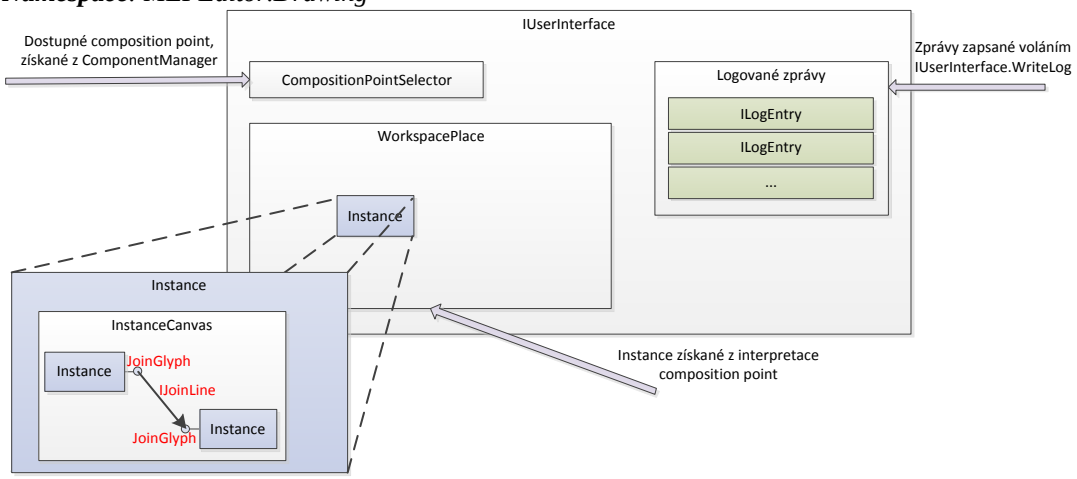
3.7.3 Naplnění importů

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 importovací 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

***Namespace**: MEFEditor.Drawing*



Obrázek 3.8-1 Vykreslování schématu kompozice a zobrazování logovaných zpráv zajišťuje assembly Drawing

Vykreslování je v editoru prováděno přes objekt splňující rozhraní `IUserInterface`. Ten je získán z assembly `MEFEditor` při startu editoru. Význam položek rozhraní je znázorněn na obrázku 3.8-1.

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 *composition point* definován. Tuto *instanci* inicializujeme pomocí patřičného bezparametrického konstruktoru. Následně je na *instanci* zavolána metoda, odpovídající vybranému *composition point*, s případnými argumenty, získanými z `CompositionPoint` atributu.

Výsledkem interpretace je seznam objevených *instancí*, pro které byla nalezena vhodná *definice zobrazení*. Tyto *instance* jsou zobrazeny ve schématu kompozice. Pokud dojde ke zneplatnění libovolné závislosti objevené během interpretování, je schéma kompozice automaticky překresleno.

3.8.1 Seznam dostupných composition point

V tomto seznamu jsou zobrazovány aktuálně dostupné *composition point* objevené v aktivním solution. Seznam je setříděný tak, aby se explicitní *composition point* zobrazovaly před implicitními. Obě skupiny se navíc zobrazují setříděné podle abecedy. Názvy *composition point* zobrazené uživateli jsou odvozeny od *fullname* příslušné metody. *Fullname* je zkrácen tak, aby byl v seznamu unikátní, ale byl co nejkratší.

Aktuálnost seznamu je zajištěna využitím události `ComponentChange` třídy `ComponentManager`, která je spuštěna při každém přidání nebo odebrání nějakého *composition point*.

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* zobrazíme podle patřičných *definic zobrazení*. Ty mají při vykreslování dostupné prostředky, díky kterým mohou zobrazovat *instance* potomků, importy/exporty a také spojení mezi nimi. Jejich využití je na konkrétním příkladu popsáno v kapitole 4.1.5.

Každá *instance* musí být vykreslena uvnitř prvku `InstanceCanvas` metodou `DrawCanvas`. Ten umožňuje zobrazené *instance* vyjmout nebo akceptovat na základě drag&drop událostí a přítomnosti potřebných editací.

Vykreslení *instance* je prováděno tak, že nejprve získáme kresbu z patřičné *definice zobrazení*. Pomocí této kresby je vytvořen objekt `InstanceThumb`, který mimo vlastního zobrazení kresby implementuje WPF události potřebné pro akce drag&drop zahrnující například zobrazení náhledu při tažení myší. `InstanceThumb` také řeší, zda je možné provést drop do `InstanceCanvas` nad kterým se nachází. Pokud se totiž nepodaří provést transformaci 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*.

Výchozí pozice zobrazených *instancí* je určena tak aby byly řazeny přibližně do čtverců. Díky 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 logickým rodičům.

Importy a exporty komponent jsou kresleny pomocí rutin `IDrawingConnector`. Pro každý vytvořený import/export je zároveň vytvořen objekt `IJoinPoint`, který může být získán jinou definicí zobrazení a použit pro definování spojnice mezi dvěma `IJoinPoint` objekty voláním `AddJoin` na `InstanceCanvas`, které přidá `IJoinLine` objekt do schématu kompozice.

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. Stejně tak není možné zapsat změnu do souboru zdrojového kódu, který je v režimu `ReadOnly`. V obou případech je uživatel informován logovanou zprávou.

4 Rozšiřitelnost editoru

4.1 Uživatelská rozšíření

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.
- **Jazykové definice** – pro zapisování změn do zdrojových kódů je nutná přítomnost objektu, který splňuje `ILanguageDefinition` rozhraní. Rozšiřitelnost o tyto objekty umožňuje vytvářet editace nezávisle na jazyku editovaných metod.
- **Definice typů** – aby bylo možné předefinovat 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 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 referencemi na knihovny *MEFEditor.ExtensionPoints* a *System.ComponentModel.Composition*. Dále předpokládáme importované namespace stejných jmen pompocí `using`. Všechny knihovny, ze kterých mají být získána uživatelská rozšíření musí být umístěné ve složce pro rozšíření uvedené v kapitole 6.2.1.

4.1.1 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 uchovává 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 spočívá v 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, IInvokeInfo)`** – metoda provádějící parsování „jednořádkových" instrukcí, jejichž výsledkem je výraz vracející objekt.
- **`IInvokeInfo ParseMethod(IParsingServices, IInvokeInfo)`** – 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` je zde [16]. `ParseMethod` je využíváno pro zpracování metod získaných ze zdrojových kódů. Instrukce které dostane jsou tedy typu `CodeFunction`. Metoda `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 je popsán zde [17] a pro parametry zde [18].

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`, se kterým jsou metody `ParseMethod` a `ParseInline` volány. Pokud není nějaká služba při parsování dostupná, její příslušející property 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 ovlivnily 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 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 pozici ve zdrojovém kódu, která s ohlášenou chybou souvisí.

4.1.2 Uživatelské interpretery

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

- **`string[] Languages`** – obsahuje seznam jazyků, které může interpreter interpretovat
- **`IInstance InterpretInline(IInvokeInfo,IExecInfo)`** – spustí interpretaci „jednořádkových“ instrukcí. Vrací instanci, kterou měli 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 dostupné služby využívány. Tyto služby jsou přístupné přes rozhraní `IExecInfo`, 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 property 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 jejich pozici. 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 volání `PushBlock` a `PopBlock` objektu `IAnalyzing`. Například u příkazu `if` to bude celý 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 a proměnnými se pracovalo. Díky tomu můžeme při editacích rozhodnout, zda přehazování řádků nezpůsobí nechtěné změny v sémantice metody. Toto označování se provádí pomocí `IAnalyzing.AddBorder`. Volání předáme libovolný objekt, který zabrání přesunutí přes řádek, se stejným objektem.

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.

4.1.3 Uživatelské jazykové definice

Zapisování editací probíhá přes objekty splňující `ILanguageDefinition`. Toto rozhraní definuje metody pro vytváření obvyklých primitiv, která můžeme v editacích použít. Také obsahuje rutiny, které slouží pro navigaci ve zdrojových kódech. Nakonec obsahuje metody pro bufferovaný zápis a čtení bufferovaných změn.

Nyní si popíšeme vytvoření části jazykové definice pro jazyk C#. Implementaci začneme vytvořením třídy exportované dle rozhraní `ILanguageDefinition`. Zároveň uvedeme *ID* jazyka C# v položce `Languages`.

```
[Export(typeof(ILanguageDefinition))]
class CSharpDefinition:ILanguageDefinition
{
    /// <summary>
    /// ID používané pro C# ve Visual Studiu
    /// </summary>
    public const string CSharpID = "{B5E9BD34-6D3E-4B5D-925E-8A43B79820B4}";
    /// <summary>
    /// Jazyky kompatibilní s touto ILanguageDefinition
    /// </summary>
    public string[] Languages{get { return new string[] { CSharpID }; }}
```

Příprava pro implementaci jazykové definice pro C#

Navigace pro C# je odvozena od `CodeElement` určených v předané pozici. Metoda `CanNavigate`, která rozhodne zda můžeme na pozici navigovat vrátí `true`, pouze pokud je v `IPosition.Source` nějaký `CodeElement`. K vlastní navigaci použijeme metody nabízené Visual Studiem. Implementace je následovná:

```
public bool CanNavigate(IPosition position)
{
    var el = position.Source as CodeElement;
    return el != null;
}
```

```
public void Navigate(IPosition position)
{
    var el = position.Source as CodeElement;
    //ProjectItem, ve které je element umístěn, musí být otevřena
    el.ProjectItem.Open();
    var doc = el.ProjectItem.Document;
    if (doc == null)
        //nepodařilo se nám obdržet dokument, ve kterém se nachází element
        return;
    //aktivujeme dokument, aby se uživateli zobrazilo okno editoru
    doc.Activate();

    //part určuje část CodeElement, od níž počítáme offset pro navigaci
    vsCMPart part;
    if (el.Kind == vsCMElement.vsCMElementFunction)
        //pro funkci navigujeme od začátku jejího těla
        part= vsCMPart.vsCMPartBody;
    else
        //jinak od začátku elementu
        part = vsCMPart.vsCMPartWholeWithAttributes;

    //start je pozice pro offset 0
    var start = el.GetStartPoint(part);
    //získáme objekt pro kurzor, který budeme přesunovat
    var sel = doc.Selection as TextSelection;
    //vlastní přesunutí kurzoru
    sel.MoveToAbsoluteOffset(start.AbsoluteCharOffset + position.Offset);
}
```

Vzorová ukázka implementace metod pro navigaci

Implementaci budeme pokračovat metodami pro vytvoření primitiv, která mohou být zapisována do zdrojového kódu. Tyto metody jsou uvedeny zde:

```
private string prettyTypeName(IType type, IContext context)
{
    if (type.Alias != null)
        //nejlepším jménem typu je alias, pokud je dostupný
        return type.Alias;
    if (context != null)
        //kontext nám vrátí jméno zkrácené dle dostupných namespace
        return context.GetFriendlyTypeName(type);
    //pokud nemáme ani alias ani context, vrátíme plné jméno
    return type.FullName;
}
public object CreateObject(string varName, IType type, object[] arguments, IContext context)
{
    string args;
    if (arguments != null)
        //argumenty v konstruktoru jsou oddělené ","
        args = string.Join(",", arguments);
    else
        //konstruktor nebude mít žádné argumenty
        args = "";
    //vrátíme konstrukci objektu dle syntaxe C#, využívající klíčové slovo var
    return string.Format("var {0} = new {1}({2})", varName,
                prettyTypeName(type,context),args);
}
public object Declaration(string varName, IType type,IContext context)
{
    //deklarace proměnné zadaného názvu a typu
    return String.Format("{0} {1}", prettyTypeName(type,context), varName);
}
```

Vzorová ukázka implementace metod pro vytvoření primitiv na konstrukci objektu a deklaraci proměnné. Obě metody používají názvy typů zkrácené podle kontextu metody, do které budeme zapisovat. Zkrácení provede volání prettyTypeName.

Implementace dalších metod pro vytvoření primitiv je analogická. Na závěr tedy uvedeme metody pro čtení, zapisování, zjišťování délky primitiv a promítnutí změn

do zdrojového kódu využívající třídu `ExtensionPoints.CodeWriter`, která poskytuje bufferované `Write`, `Read` metody. Změnu do zdrojových kódů promítneme voláním `Commit`, které vrátí `null`, pokud se zapsání změn povedlo, jinak vrátí popis chyby, kvůli které nebyla data zapsána. Využijeme je následovně:

```
public void Write(object data, IPosition position, int length)
{
    var toWrite = (string)data;
    //přepíšeme zdrojový kód délky length od position textem toWrite
    CodeWriter.Write(position, length, toWrite);
}
public object Read(IPosition position, int length)
{
    //načteme data začínající na position o délce length
    return CodeWriter.Read(position, length);
}
public int Length(object element)
{
    //v naší reprezentaci odpovídá délka element jeho stringové reprezentaci
    return ((string)element).Length;
}
public bool Flush()
{
    //zápis se povedl, pokud nebyla vrácena žádná chyba
    return CodeWriter.Commit() == null;
}
```

Ukázková implementace metod pro práci se zdrojovým kódem, využívající třídu CodeWriter.

Nyní máme funkční jazykovou definici pro C#, která nám umožní provádět veškeré editace podporované editorem. Celá implementace třídy je uvedena v souboru *LanguageDefinitions/CSharpDefinition.cs* projektu *doporučených rozšíření* v příloze [B].

4.1.4 Uživatelské definice typů

Editace, které objekt nabízí, jsou definovány typovou definicí pro typ 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 upřednostňovány typy z `Runtime`. Díky tomu můžeme například nahradit typové definice získané z MEF knihoven vlastními typovými definicemi, které již umožní potřebné editace.

Typová definice, kterou naimplementujeme v této kapitole, je dostupná v projektu *UserExtensions* v příloze [E]. V projektu je definováno také několik composition point, na kterých je možné funkci typové definice vyzkoušet v prostředí editoru.

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 `RuntimeType Definition`, 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 `RuntimeTypeDefinition`.

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. `RuntimeTypeDefinition` nám umožňuje využít interpreter, který dovede spouštět metody ve tvaru `IInstance method(IExecInfo)`. Instrukce simulovaných metod tedy budou metody naší typové definice. Její členy definujeme následovně:

```
    [Export(typeof(ITypeDefinition))]
    public class DiagnosticDefinition : RuntimeTypeDefinition
    {
        public DiagnosticDefinition()
        {
            //zde bude inicializace rawname a seznamu metod
        }
        private IInstance _ctor(IExecInfo info){
            //instrukce pro konstruktor
        }
        private IInstance _start(IExecInfo info){
            //instrukce pro metodu Start
        }
        private IInstance _stop(IExecInfo info){
            //instrukce pro metodu Stop
        }
        private IInstance _accept(IExecInfo info){
            //instrukce pro metodu Accept
        }
        public override bool PreloadRoutines(ITypeLoadingConnector connector, IAssembly
runtime)
        {
           //rutiny, které provedou test při spuštění editoru
        }

    }
```

Takto vypadá naše typová definice, připravená k implementaci.

Implementaci začneme inicializací `RuntimeTypeDefinition._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 `RuntimeTypeDefinition.Create`, specifikující instrukce metody a poté ji přidáme do seznamu dostupných metod pomocí `RuntimeTypeDefinition.AddMethod`. Ve zdrojovém kódu to vypadá takto:

```
            _rawName = typeof(Diagnostic).FullName;

            var ctor = Create(_ctor, _rawName);
            //Jedná se o konstruktor
            ctor.IsConstructor = true;

            var start = Create(_start, Constants.Type_Void);
            var stop = Create(_stop, Constants.Type_Void);
            var accept = Create(_accept, Constants.Type_Void,"System.Object");
            //Poslední argument je parametrický
            accept.SetParametric();

            AddMethod("Diagnostic", ctor);
            AddMethod("Start",start );
            AddMethod("Stop", stop);
            AddMethod("Accept", accept);
```

Tělo konstruktoru nyní vypadá následovně. Upozorněme na nutnost označit 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 níže uvedených metodách `_start`, `_stop` pouze odsimulujeme zapnutí a vypnutí časovače.

```
 private IInstance _ctor(IExecInfo info){
    //Stopwatch z namespace System.Diagnostic
    info.SetData("watch", new Stopwatch());
    info.SetData("instances", new List<IInstance>());

    //vytvoříme accepter, který přijme instance vytvořením volání metody Accept
    var accepter=new InstanceAccepter(info.This,"Accept");
    //vytvořený accepter vložíme do konstruované instance, tak aby přijímal instance
    //typu System.Object
    info.This.Editations.AddInstanceAccepter("System.Object", accepter);

    //je přirozené z konstruktoru vrátit vytvářenou instanci
    return info.This;
}
private IInstance _start(IExecInfo info){
    //získáme Stopwatch objekt z datových položek instance
    var watch = info.GetData("watch") as Stopwatch;
    //spustíme měření času
    watch.Start();

    //funkce Start je void funkce
    return info.InstanceCreator.Void;
}
private IInstance _stop(IExecInfo info){
    var watch = info.GetData("watch") as Stopwatch;
    //zastavíme měření času
    watch.Stop();

    return info.InstanceCreator.Void;
}
```

Implementace několika metod, z vytvářené typové definice.

Pokračujme implementací metody `_accept`. Před zpracováním argumentů poskytnutých metodě nastavíme akceptování instancí typu `System.Object` na poslední volání metody `Accept`. Díky tomu budou instance přijímány přidáním argumentu do tohoto volání. Akceptování nastavíme následovně:

```
private IInstance _accept(IExecInfo info){
    //zkratka
    var callEdits = info.CallInfo.CallEditations;

    //chceme akceptovat posledním voláním funkce Accept na instanci
    if (callEdits != null) //ostestujeme zda jsou dostupné CallEditations
    {
        //odstraníme předchozí instance accepter
        info.This.Editations.RemoveAccepter("System.Object");
        //vytvoříme nový argument accepter
        var argAccepter=callEdits.ArgumentAccepter();
        //přidáme argument accepter patřičné instanci
        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 instance `Diagnostic`, zároveň umožníme jejich odstranění metodou drag&drop díky poskytnutým `IRemoveProvider`. O samotné zobrazení se však musí postarat definice zobrazení, kterou implementujeme v následující kapitole. Argumenty zpracujeme takto:

```
//získáme již přidané instance
var instances = info.GetData("instances") as List<IInstance>;
//zde budeme udržovat index aktuálně zpracovaného argumentu
int argIndex=0;
//všechy argumenty přidáme do seznamu
foreach (var arg in info.CallInfo.Arguments)
{
    //uložíme instanci
    instances.Add(arg.Instance);
    //otestujeme zda jsou dostupné editace pro toto volání
    if (callEdits != null)
    {
        //IRemoveProvider, kterým odstraníme argument zadaného indexu
        var remover=callEdits.ArgumentRemover(argIndex);
        //nechceme aby odstranění jednoho argumentu smazalo celé volání
        remover.IsOptional = true;
        //nastavíme argument jako logického potomka, které může být odstraněno
        //pomocí remover
        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 takto:

```
        //vrátíme instanci reprezentující počet uložených instancí
        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 následující test do metody `PreloadRoutines`.

```
public override bool PreloadRoutines(ITypeLoadingConnector connector, IAssembly runtime)
{
   if (TestValueField<int>(0, connector))
      //dostupná typová definice ukládá hodnotu do položky Value
      return true;

   //s touto typovou definicí naše rozšíření neumí pracovat
   connector.Log(LogLevels.error, "Missing compatible System.Int32 definition");
   return false;
}
```

TestValueField je metoda zděděná od RuntimeTypeDefinition. Umožňuje otestovat zda typová definice umožňuje nahrání ze zadaných dat a zda je ukládá v položce Value.

Tímto máme implementovanou třídu pro rozšiřující *typovou definici*. V následující kapitole pro ní implementujeme definici zobrazení.

4.1.5 Uživatelské zobrazení instancí

Instance 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.

Definice zobrazení, kterou naimplementujeme v této kapitole, je dostupná v projektu *UserExtensions* v příloze [E]. V projektu je definováno také několik composition point, na kterých je možné funkci definice zobrazení vyzkoušet v prostředí editoru.

Rozhraní, které musí definice zobrazení splňovat je `IDrawingDefinition`. Podle tohoto rozhraní ji také exportujeme pro použití v editoru. Rozhraní obsahuje property `FullName`, která 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 `FrameWorkElement` 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 *System.Xaml*, *PresentationFrameWork*, *PresentationCore* a *WindowsBase*, 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`. Pro vykreslení spojnic mezi importy a exporty budeme navíc potřebovat třídu splňující rozhraní `IJoinLine`. Implementaci zahájíme následovně:

```
//Namespace potřebné pro WPF objekty
using System.Windows.Shapes;
using System.Windows.Controls;
using System.Windows.Media;
using System.Windows;

    class JoinLine : IJoinLine
    {
        Line _drawing;
        public UIElement Drawing { get { return _drawing; } }

        public JoinLine()
        {
            //vytvoříme kresbu spojovací čáry
            _drawing = new Line();
            _drawing.Stroke = Brushes.Red;
            _drawing.StrokeThickness = 1;
        }
        public void SetPoints(Point from, Point to)
        {
            //při přesunutí konektorů upravíme souřadnice
            _drawing.X1 = from.X;
            _drawing.Y1 = from.Y;
            _drawing.X2 = to.X;
            _drawing.Y2 = to.Y;
        }
```

```
    public IJoinLine Clone()
    {
        //Vrátíme nový IJoinLine objekt.
        return new JoinLine();
    }
}
[Export(typeof(IDrawingDefinition))]
public class DiagnosticDrawing : IDrawingDefinition
{
    public string FullName{get{return typeof(Diagnostic).FullName;}}

    public FrameworkElement GetDrawing(IInstance instance, IDrawingConnector connector)
    {
      //zde vytvoříme kresbu, zobrazovanou ve schématu kompozice
    }
}
```

Třída JoinLine bude použita pro vykreslení spojnice mezi importy a exporty. DiagnosticDrawing je třída připravená pro implementaci metody GetDrawing, která vykreslí zadanou instanci.

Pro vytvoření WPF kresby máme k dispozici `IDrawingConnector`, který obsahuje metody pro vytvoření import/export konektoru. 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. Inicializaci tedy provedeme takto:

```
//drawing bude obsahovat objekt, vrácený jako kresba instance
var drawing = new StackPanel();
drawing.Background = Brushes.LightGreen;

//získáme data z kreslené instance
var watch = instance.GetData("watch") as Stopwatch;
var instances = instance.GetData("instances") as List<IInstance>;

//výstup ze StopWatch
var caption = new TextBlock();
caption.Text = string.Format("{0}ms",watch.ElapsedMilliseconds.ToString());

//canvas, který bude využíván pro kreslení logických potomků
var canvas = connector.CreateInstanceCanvas();
canvas.MinHeight = 100;
canvas.MinWidth = 100;

//definuje rozvržení kresby
drawing.Children.Add(caption);
drawing.Children.Add(canvas);

//sesbírané import/export IJoinPoint
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 voláním `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 pomocí objektu `connector` tímto způsobem:

```
        foreach (var inst in instances)
            {
                //vykreslí každou instanci v kolekci podle správné definice zobrazení
                canvas.DrawInstance(inst);
                var componentInfo = inst.CreationType.ComponentInfo;

                if (componentInfo == null)
                    //není dostupné žádné component info
                    continue;

                //sesbíráme importy a exporty
                foreach (var imp in componentInfo.Imports)
                    imps.Add(connector.GetJoinPoint(inst, imp));
                foreach (var exp in componentInfo.Exports)
                    exps.Add(connector.GetJoinPoint(inst, exp));
            }


            //vykreslíme spojení mezi importy a exporty
            foreach (var impPoint in imps)
            {
                //zobrazí zprávu v ToolTip
                impPoint.SetMessages("Import error", "Import warning");
                foreach (var expPoint in exps)
                    //nakreslí spojnici mezi importem a exportem
                    canvas.AddJoin(impPoint, expPoint, new JoinLine());
            }

            return drawing;
        }
```

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

Dokončili jsme 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.

4.2 Standardní rozšíření

Pro práci editoru je nezbytná přítomnost některých rozšíření. Tato rozšíření jsou pevně zabudovaná v editoru a není možné je uživatelem odebrat. V rámci práce je nazýváme standardní rozšíření. Editoru poskytují některé *typové definice* a standardní interpreter. Z přítomnosti těchto rozšíření však plynou jistá omezení. Uživatel editoru nemá možnost ovlivnit chování typů, pro které existuje standardní *typové definice*.

4.2.1 Standardní typové definice

Standardní rozšíření obsahují *typové definice* typů, nutných pro vyhledávání komponent. Jedná se o typy těchto atributů.

- Import
- ImportMany
- ImportingConstructor
- Export

- ExportMetadata
- CompositionPoint

Dále jsou zde *typové definice, speciálních typů*.

- System.Array<Type,Dimension>
- System.Proxy<Type>
- System.Void
- System.Null

Jejich význam je odlišný od běžných .NET typů, jak bylo popsáno v kapitole 3.2.3. Tyto typy je možné využívat v uživatelských rozšířeních. *Instance* ze `System.Array` může být vytvořena injektováním z objektu splňující rozhraní `IArrayDefinition` o správném počtu dimenzí. Pole lze také vytvořit voláním konstruktoru, který očekává instance typu `int`, udávající velikost každé dimenze pole. Datová položka `Constants.DataField_WrappedValue` pro *instanci* pole obsahuje `IArrayDefinition`, která pole reprezentuje.

Instance typu `System.Proxy<Type>` vytváříme přímo, z delegátu typu `ProxyMethodCall`. Tento delegát pak bude volán místo každé metody zavolené na *instanci*. Poslední standardní *typovou definicí* je definice pro `System.Object`, která je nezbytná pro korektní chování uvedených typů.

4.2.2 Standardní interpreter

Pro potřeby převodu členských proměnných na patřičné setter a getter metody, využívá editor `NativeMethodInterpreter`. Stejný interpreter je pak využíván pro interpretaci instrukcí v *typových definicích* odvozených od `RuntimeTypeDefinition`. Jelikož instrukcemi pro uvedený interpreter jsou .NET delegáty, není třeba provádět jejich parsování. Vlastní interpretace je pak pouhé zavolání delegáta.

4.3 Doporučená rozšíření

V rámci této práce byla pro editor implementována *doporučená rozšíření*, která umožňují nasazení editoru v projektech psaných jazykem C#. Součástí těchto rozšíření je také podpora pro zpracování atributů z metadat referencovaných MSIL knihoven.

Pro testování *doporučených rozšíření* je v příloze [E] projekt *ExtensionsTests*. Jsou zde příklady zdrojových kódů pro „záludné“ situace, které jsou řešeny implementovanými parsery, interpretery a typovými definicemi. V následujících kapitolách si popíšeme vlastnosti *doporučených rozšíření*.

4.3.1 Rozšíření pro parsování

Parsování je řešeno ve dvou fázích. V první fázi jsou rozparsovány instrukce zadaného `CodeElement` pomocí objektu třídy `SyntaxParser`. Výsledkem první fáze je syntaktický strom, který nevyužívá žádné informace z typového

systému editoru. Je tedy závislý pouze na změně zdrojového kódu parsovaného `CodeElement` objektu.

Druhá fáze je prováděna semantickým parserem `SemanticParser`. Ten ze syntaktického stromu vytvoří strom sémantický, který již obsahuje typové informace o proměnných, voláních metod,… Sémantický strom také obsahuje *instance* primitivních typů. Nepředpokládá se totiž, že by v průběhu interpretace docházelo ke změnám vnitřního stavu těchto *instancí*. Mohou být tedy editorem cachovány, což výrazně zrychluje vlastní interpretaci sémantického stromu.

Dohromady umožňují parsery rozparsovat obvyklé konstrukce jazyka C#. Jmenovitě jsou to:

- deklarace proměnných a jejich přiřazování
- parsování literálů pro `string`, `char`, `int`, `bool`
- aritmetické výrazy s binárními, prefixovými i postfixovými operátory
- volání negenerických metod
- konstrukce objektů pomocí new
- přetypování
- inicializátory a indexery polí
- blokové příkazy `if`, `while`, `for`, `do`, `switch`

Pokud editor narazí na konstruci, které nerozumí nebo ji považuje za syntaktickou chybu ohlásí tuto chybu pomocí uživatelského rozhraní editoru spolu s možností navigovat na místo ve zdrojovém kódu, kde k chybě došlo.

4.3.2 Rozšíření pro interpretaci

Interpretace sémantického stromu je prováděna objektem třídy `SemanticInterpreter`. Podporuje interpretaci všech konstrukcí vyjmenovaných v předchozí kapitole. V průběhu interpretace poskytuje editoru údaje, využitelné pro editace. Jedná se především o seznam všech deklarovaných proměnných, který zabrání duplicitní deklaraci proměnné v patřičných editacích. Dále interpreter poskytuje informace o rozsahu platnosti proměnných, který je využíván například při drag&drop editacích na přijmutí instance. Volání, která editor vykonává mají k dispozici veškeré služby definované rozhraním `ICallEditations`. Volané objekty mohou na jejich základě přidat libovolnou editaci.

Druhý implementovaný interpreter je `MSILInterpreter`, který však podporuje pouze `InterpretInline` interpretaci využívanou pro vytváření instancí z atributů v MSIL assemblies.

4.3.3 Jazyková definice pro C#

Aby bylo možné zapisovat do zdrojových kódů napsaných jazykem C#, je součástí doporučených rozšíření třída `CSharpDefinition`, implementující všechny služby definované rozhraním `ILanguageDefinition`. Podporuje zápis/čtení zdrojového kódu metod, instanciování primitivních typů z literálů zdrojového kódu pro `int`, `string`, `bool`, `char`. Dále podporuje navigaci do libovolného `CodeElement` ve zdrojovém kódu.

Zápisy prováděné metodou `ILanguageDefinition.Write` jsou navíc cachovány, dokud není zavolána `ILanguageDefinition.Flush`, která provede všechny zápisy najednou. Díky tomu je možné tyto zápisy sjednotit do jediné položky v *Undo* menu Visual Studia, což usnadňuje práci s editorem.

4.3.4 Rozšiřující typové definice

Editace nabízené editorem závisí na dostupných rozšiřujících *typových definicích*. Z tohoto důvodu obsahují doporučená rozšíření následující *typové definice*, zaměřené na práci s MEF. Zde je uveden jejich výčet:

Typová definice	**Implementované vlastnosti**
AggregateCatalog	• ***Accept/Remove*** editace na `ComposablePartCatalog`
TypeCatalog	• ***Add component type*** – přidání typu ze seznamu typů dostupných komponent • ***Exclude from TypeCatalog*** – odstranění typu přítomného v katalogu
DirectoryCatalog	• ***Set path*** – nastavení výchozí cesty pro vyhledání knihoven • ***Set pattern*** – nastavení vyhledávacího vzoru • ***Open folder*** – otevře aktuálně nastavenou cestu pro vyhledávání knihoven • ***Sledování změn ve složce*** – projevuje se překreslením schématu při změně složky pro vyhledání komponent
CompositionContainer	• ***Accept/Remove*** editace na komponenty • ***Accept/Remove*** editace na `ComposablePartCatalog` • Simulace MEF kompozice, na jejímž základě zobrazuje případné chyby, které byly při kompozici objeveny. Pokud se chyby nevyskytují, zobrazí vztahy mezi importy a exporty. Také provede naplnění importů z dostupných exportů
AssemblyCatalog	• ***Set path*** **–** nastavení výchozí cesty pro načtení assembly • Zobrazuje informace o načtené assembly

Tabulka implementovaných typových definic důležitých pro MEF

Všechny tyto *typové definice* navíc přidávají *statickou editaci* na vytvoření objektu patřičného typu do kontextového menu *statických editací*. Objekty jsou vytvořeny pomocí bezparametrických konstruktorů, případně pomocí konstruktorů s průvodcem pro nastavení jejich parametrů.

Doporučená rozšíření přidávají další *typové definice* nutné pro fungování výše uvedených *typových definic*. Všechny implementované *typové definice* jsou

v projektu `Recommended_extensions`, ve složce *ObjectModel*. Projekt je dostupný v příloze [B].

4.3.5 Rozšíření pro vykreslování schématu kompozice

Aby bylo možné zobrazovat *instance* vytvořené z typů *typových definic* uvedených v předchozí tabulce, jsou v *doporučených rozšířeních* implementovány jejich definice zobrazení. Tyto definice využívají standardní rozhraní editoru určené pro vykreslování schématu kompozice. Poskytují tedy drag&drop editace na *instancích,* pro které jsou dostupné. Stejně tak zachovávají jejich zobrazení, určené dostupnými *typovými definicemi*.

Pro vykreslování komponent je implementována definice zobrazení pro `System.Object`, využívající standardní rozhraní pro vytváření zobrazení importů a exportů. Spojení mezi těmito *instancemi* proto může ovlivňovat libovolná rozšiřující definice zobrazení.

Všechny definice zobrazení zobrazují textový popis chyby, které se vyskytly v průběhu interpretace. Příkladem může být `DirectoryCatalog`, jemuž byla zadána neexistující složka pro vyhledávání knihoven. Tuto skutečnost ohlásí výpisem příslušné chybové hlášky v těle zobrazeného katalogu.

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 kompozice.
- 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 *Recommended_extensions.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.

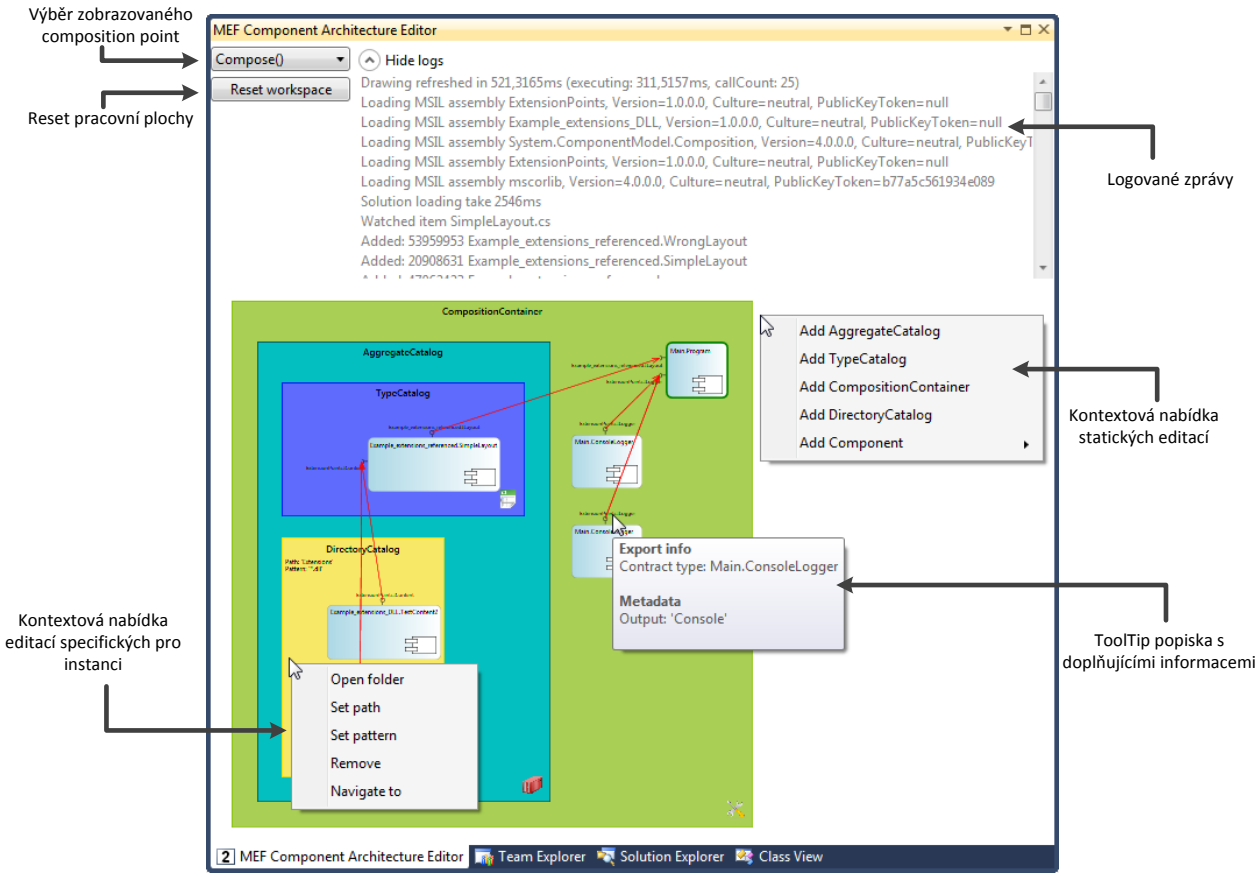
Umožňují práci a zobrazování schémat kompozice nad projekty psanými jazykem C#. Obsažená rozšíření objektového modelu jsou schopná simulovat průběh MEF kompozice a díky tomu dokáží zobrazit vztahy mezi komponentami, případně chyby, které se při kompozici objevily.

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.

Další vývoj editoru by mohl být směřován na implementaci nových rozšíření. Tato rozšíření by měla umožnit analýzu kompozice přímo z binárních souborů MSIL knihoven, případně z projektů napsaných i jinými jazyky, než je C#.

6 Uživatelská příručka

6.1 Uživatelské rozhraní editoru



Obrázek 6.1-1 Uživatelské rozhraní editoru, s popsanými ovládacími prvky

Editor zobrazuje schéma kompozice na základě právě vybraného composition point. Po vybrání se zobrazí patřičné schéma kompozice nebo bude vypsána chyba, ke které došlo v průběhu parsování/interpretování a kvůli které nebylo možné schéma kompozice vykreslit. Kontextová nabídka u vypsané chyby obsahuje příkaz pro zkopírování textové reprezentace chyby do schránky, případně může také obsahovat příkaz pro navigaci na místo ve zdrojovém kódu, kde k chybě došlo.

V zobrazeném schématu kompozice jsou editace prováděny na základě kontextových nabídek zobrazených instancí. Jiným způsobem jak lze editace provádět je drag&drop akce, kdy požadovanou instanci zkusíme přesunout do prostoru nějakého kontejneru. Pokud je editace možná, dojde po drop akci k zapsání změn do zdrojového kódu. Statické editace jsou přístupné v kontextové nabídce volné plochy shématu kompozice. Pomocí rolování kolečka myši je možné měnit velikost zobrazovaných instancí ve schématu kompozice. Celé schéma je také možné libovolně posunovat, tažením za volnou plochu schématu kompozice.

V průběhu analýzy editoru vzniká množství zpráv, které mohou objasnit případné nečekané chování editoru. Může se jednat například o chybějící interpretery/parsery, syntaktické chyby, postup nahrávání knihoven, čas průběhu

jednotlivých operací a další. Posledních několik zpráv je možné prohlížet v poli logovaných zpráv, kde jsou jednotlivé zprávy barevně odlišeny podle důležitosti. Některé zprávy také umožňují navigovat na místo, kterého se týkají. Navigace se provede kliknutím na logovanou zprávu.

6.2 Použití editoru

V následujících kapitolách si popíšeme, jakým způsobem přidáme editor do prostředí Microsoft Visual Studia 2010. Dále si na konkrétním příkladu předvedeme základní principy práce s editorem.

6.2.1 Instalace a spuštění

Po spuštění souboru *MEF_Component_Architecture_Editor.vsix* z přílohy [C] se zobrazí dialogové okno, které nás provede přidáním editoru do prostředí Visual Studia 2010, jež musí být na cílovém počítači nainstalované. Po provedení instalace spustíme Visual Studio a položkou v menu *View* > *Other Windows* > *MEF Component Architecture Editor* spustíme editor. Při jeho prvním spuštění je vytvořena složka pro rozšíření editoru:

Dokumenty/Visual Studio 2010/MEF Component Architecture Editor/

Do složky je následně nahrána knihovna doporučených rozšíření, která umožní využít editor v projektech napsaných jazykem C#. Pokud nechceme, aby byla doporučená rozšíření v editoru dostupná, stačí tuto knihovnu smazat.

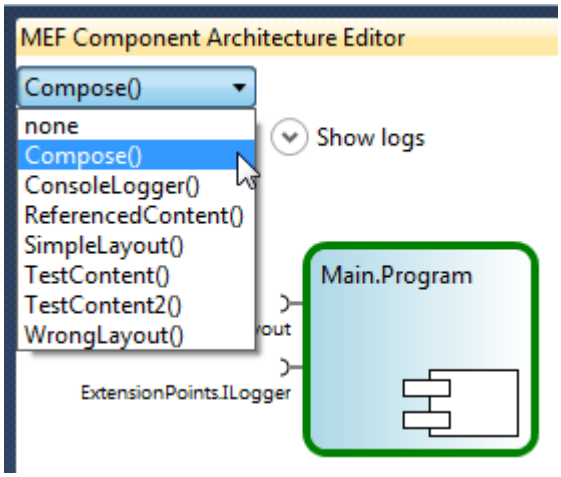
Pro přidání uživatelských rozšíření zkopírujeme knihovnu, ve které jsou implementována, do výše uvedené složky. Rozšíření budou nahrána při příštím spuštení editoru. Seznam nahraných rozšíření, případně chyby objevené při jejich nahrávání jsou zobrazovány v logovaných zprávách.

6.2.2 Příklad použití na konkrétním projektu

Předpokládejme, že máme spuštěný editor a otevřené solution *ExampleMEFEditor* z přílohy [E]. Struktura solution má reprezentovat situaci, kdy chceme do assembly `Main` nahrát komponenty z knihoven přítomných ve složce *Extensions*. Tato složka obsahuje zkompilovanou assembly `Example_extensions_DLL`. Do kompozice chceme dále přidat komponentu `SimpleLayout` z referencované assembly `Example_extensions_referenced`. Výsledek kompozice pak použijeme v metodě `Program.Main` na poskytování html stránky, jejíž vzhled je definován komponentami. Použití editoru závisí na dostupných rozšíření, která jsou k dispozici. Tento návod počítá s tím, že jsou v editoru nahrána *doporučená rozšíření*.

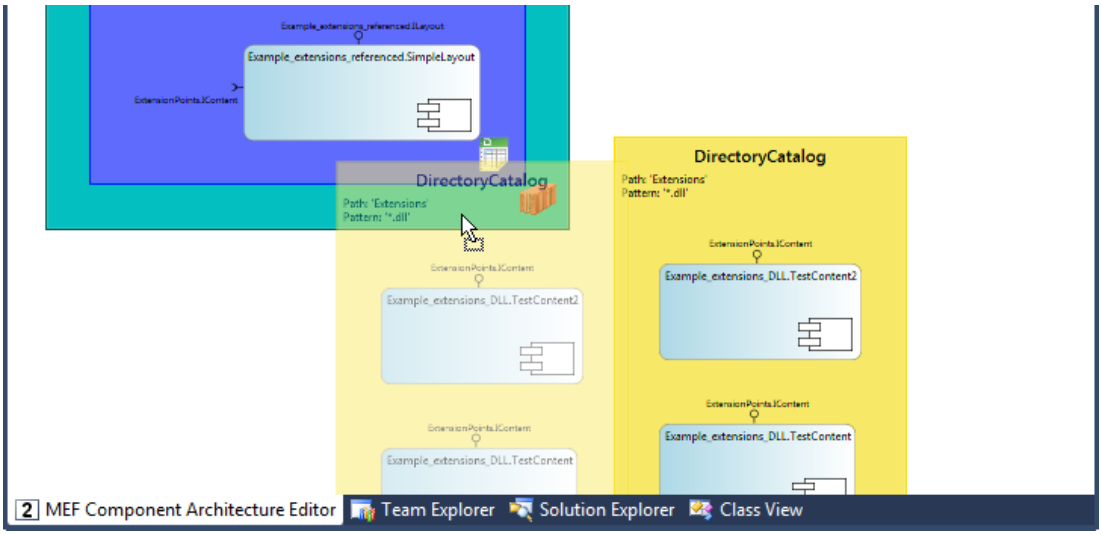
Kompozici chceme provést v metodě `Program.Compose`, označme ji tedy atributem `CompositionPoint`. V seznamu pro výběr composition point se nám objeví právě přidaná metoda. Jejím zvolením se nám zobrazí informace o komponentě `Program`, tak jak je uvedeno na následujícím obrázku:

```
class Program
{
    [Import]
    ILayout compositionResult=null;

    [ImportMany]
    ILogger[] loggers=null;

    [CompositionPoint]
    void Compose()
    {

    }
```



Obrázek 6.2-1 Takto vypadá prostředí Visual Studia po označení metody atributem CompositionPoint a jejím vybrání v editoru.

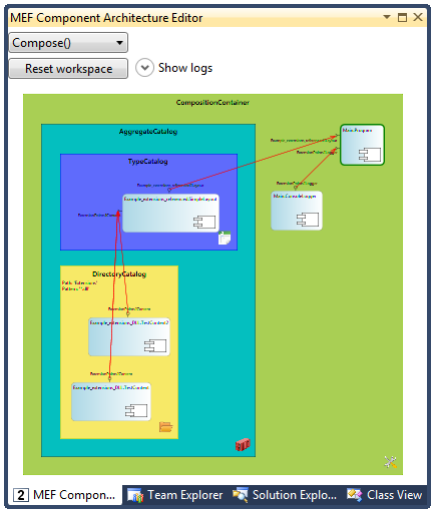
V kontextové nabídce statických editací přidáme `DirectoryCatalog` pro složku *Debug/Extensions*. Stejným způsobem pak vytvoříme `TypeCatalog` a pomocí jeho specifické editace *Add component type* přidáme `SimpleLayout` komponentu. Pro kompozici však potřebujeme všechny katalogy přesunout do jediného katalogu. K tomuto účelu slouží `AggregateCatalog`. Opět ho vytvoříme z menu statických editací a myší přesuneme `TypeCatalog` a `DirectoryCatlog` do vytvořeného katalogu. Přesunování `DirectoryCatalog` je znázorněno zde:



Obrázek 6.2-2 Postup přesunování katalogu s komponentami do AggregateCatalog

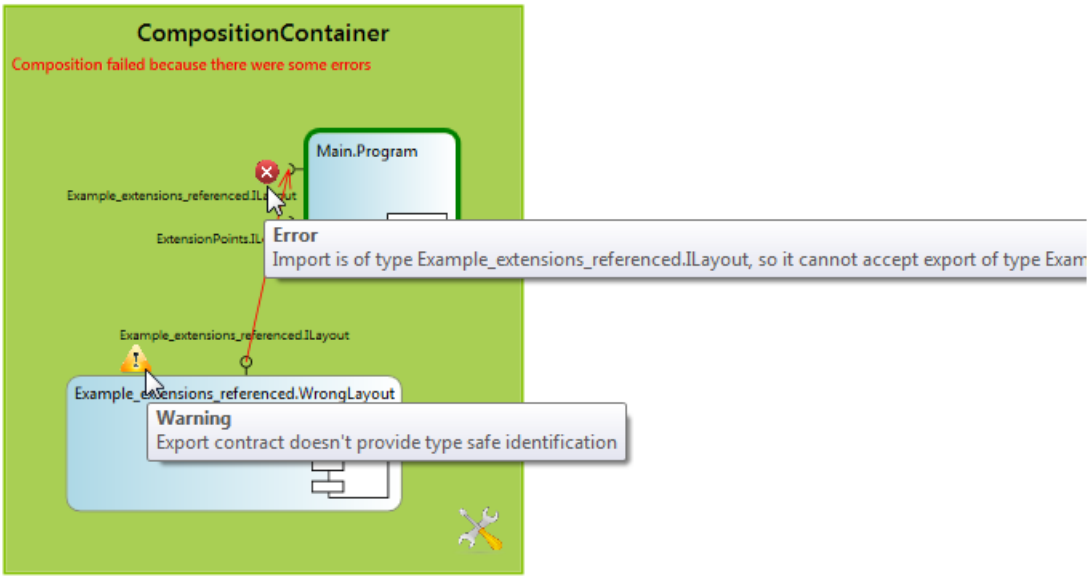
Abychom dokončili kompozici, vytvoříme `CompositionContainer`, který najdeme v nabídce statických editací. Přesuneme do něj `AggregateCatalog` a následně komponentu `Program`, která bude přidána voláním `ComposeParts` na `CompositionContainer` a způsobí tak kompozici. Do `CompositionContainer` ještě přidáme komponentu `ConsoleLogger` vytvořenou pomocí statické editace *Add Component*. Tím dostaneme požadované schéma kompozice. Ze spojnic znázorněných editorem vidíme, kterými exporty jsou jednotlivé importy naplněny.

```
class Program
{
   [Import]
   ILayout compositionResult=null;

   [ImportMany]
   ILogger[] loggers=null;

   [CompositionPoint]
   void Compose()
   {
      var consolelogger = new ConsoleLogger();
      var consolelog = new ConsoleLogger();
      var typecatalog = new TypeCatalog(typeof(SimpleLayout));
      var directorycatalog = new DirectoryCatalog(@"Extensions");
      var aggregatecatalog = new AggregateCatalog();
      var compositioncontainer = new CompositionContainer(aggregatecatalog);
      aggregatecatalog.Catalogs.Add(typecatalog);
      aggregatecatalog.Catalogs.Add(directorycatalog);
      compositioncontainer.ComposeParts(consolelog, consolelogger, this);
   }
```



Obrázek 6.2-3 Konečná podoba schématu kompozice vytvořená v rámci ukázkového příkladu. Zdrojový kód v metodě Compose byl vygenerován editorem na základě prováděných editací.

Spuštěním aplikace získáme http server, poskytující stránku vygenerovanou komponentami dostupnými při kompozici znázorněné na obrázku 6.2-3. Definovali jsme tedy kompozici ukázkové aplikace. Pomocí *doporučených rozšíření* můžeme navíc detekovat některé chyby, které mohou při kompozici vznikout. Jejich zobrazení je znázorněno zde:



Obrázek 6.2-4 Ukázka způsobu hlášení chyb odhalených v kompozici. Zde například třída WrongLayout nesplňuje rozhraní ILayout slibované kontraktem.

Použití editoru nemusí však vycházet pouze z nabízených editací. Editor reaguje i na změny zdrojového kódu, provedené přímo uživatelem. Stejně tak je schéma kompozice překresleno například při změně ve složce sledované `DirectoryCatalogem`. Díky tomu může sloužit pro kontrolu aktuálního schématu kompozice na základě úprav zdrojového kódu prováděných uživatelem.

Pokud dojde k neočekávaným úpravám zdrojového kódu editorem, je možné jednotlivé editace vrátit pomocí standardního příkazu *Undo* v prostředí Visual Studia. Editace jsou v *Undo seznamu* označené jako *MEF Component Architecture Editor change*.

7 Seznam použitých zdrojů

[1] Rozšiřitelnost aplikací pomocí MAF, MSDN, webová adresa:
http://msdn.microsoft.com/en-us/library/bb384200.aspx

[2] Úvod do technologie MEF, MSDN, webová adresa:
http://msdn.microsoft.com/en-us/library/dd460648.aspx

[3] Co je to CIL/MSIL, Wikipedia, webová adresa:
http://en.wikipedia.org/wiki/Common_Intermediate_Language

[4] Nástroj Mefx pro ladění kompozice, MSDN, webová adresa:
http://msdn.microsoft.com/en-us/library/ff576068.aspx

[5] Stránka projektu Visual MEFX, webová prezentace:
http://xamlcoder.com/blog/2010/04/10/updated-visual-mefx/

[6] Stránka projektu MEF Visualizer Tool, Codeplex, webová prezentace:
http://mefvisualizer.codeplex.com/

[7] Popis atributů využívaných v MEF, MSDN, webová adresa :
http://msdn.microsoft.com/en-us/library/ee155691.aspx

[8] Tabulka pro převod operátoru na název funkce, MSDN, webová adresa:
http://msdn.microsoft.com/en-us/library/ms229032.aspx

[9] Teorie o neřešitelnosti halting problem, Wikipedia , webová adresa :
http://en.wikipedia.org/wiki/Halting_problem

[10] Tvorba pluginu pomocí VsPackage, MSDN, webová adresa :
http://msdn.microsoft.com/en-us/library/cc138589.aspx

[11] Rozhraní EnvDTE.DTE, MSDN, webová adresa :
http://msdn.microsoft.com/en-us/library/envdte.dte.aspx

[12] Použití Code Model, MSDN, webová adresa :
http://msdn.microsoft.com/en-us/library/ms228763.aspx

[13] Popis knihovny Mono.Cecil, oficiální stránka projektu Mono, webová adresa:
http://www.mono-project.com/Cecil

[14] Pojem garbage collector, Wikipedia , webová adresa:
http://en.wikipedia.org/wiki/Garbage_collection_(computer_science)

[15] Sledování změn složek a souborů, MSDN, webová adresa:
http://msdn.microsoft.com/en-us/library/system.io.filesystemwatcher.aspx

[16] Rozhraní CodeElement, MSDN, webová adresa:
http://msdn.microsoft.com/en-us/library/envdte.codeelement.aspx

[17] Popis formátu CodeVariable.InitExpression, MSDN, webová adresa:
http://msdn.microsoft.com/en-us/library/envdte.codevariable.initexpression.aspx

[18] Popis formátu CodeParameter2.DefaultValue, MSDN, webová adresa:
http://msdn.microsoft.com/en-us/library/envdte80.codeparameter2.defaultvalue.aspx

[19] Oficiální stránka projektu Sandcastle, CodePlex, webová adresa:
http://sandcastle.codeplex.com/

[20] Common Language Specification, MSDN, webová adresa:
http://msdn.microsoft.com/en-us/library/12a7a7h3(v=vs.100).aspx

8 Přílohy

Přílohy umístěné na přiloženém CD:

A. Implementace MEF Component Architecture Editor
Příloha je umístěna ve složce *Implementace/Editor*

Obsahuje projekt pro Visual Studio 2010, ve kterém byl implementován a zkompilován náš editor. Součástí projektu jsou okomentované zdrojové kódy.

B. Implementace doporučených rozšíření
Příloha je umístěna ve složce *Implementace/DoporucenaRozsireni*

Obsahuje projekt pro Visual Studio 2010, ve kterém byla naimplementována a zkompilována doporučená rozšíření pro náš editor. Součástí projektu jsou okomentované zdrojové kódy.

C. Soubory pro instalaci editoru
Příloha je umístěna ve složce *Instalace*

Obsahuje soubor *MEF_Component_Architecture_Editor.vsix*, určený pro instalaci editoru formou pluginu do Visual Studia 2010. Dále obsahuje knihovnu *Recommended_Extensions.dll* doporučených rozšíření a knihovnu *MEFEditor.ExtensionPoints.dll*, která je nutná pro psaní uživatelských rozšíření.

D. Dokumentace vygenerovaná ze zdrojových kódů
Příloha je umístěna ve složce *Dokumentace*

Obsahuje automaticky generovanou dokumentaci získanou ze zdrojových kódů nástrojem Sandcastle [19].

Dokumentace se skládá z následujících souborů:

- *MEF_Component_Architecture_Editor_Documentation.chm*
  Dokumentace typů použitých v implementaci editoru.
- *Recommended_Extensions_Documentation.chm*
  Dokumentace typů použitých v implementaci doporučených rozšíření.
- *ExtensionPoints_Documentation.chm*
  Dokumentace typů využívaných pro psaní uživatelských rozšíření.

E. Ukázkové projekty na použití editoru
Příloha je umístěna ve složce *Příklady*

Obsahuje projekt pro Visual Studio 2010, který byl použit v kapitole 6.2.2 jako ukázkový příklad, dále projekt s testovacími daty doporučených rozšíření a projekt implementovaných uživatelských rozšíření z kapitol 4.1.4 a 4.1.5.

F. Elektronická verze této práce
Příloha je umístěna ve složce *Dokumentace*

Obsahuje soubor *bakalářská_práce.pdf*, který je elektronickou verzí této práce.