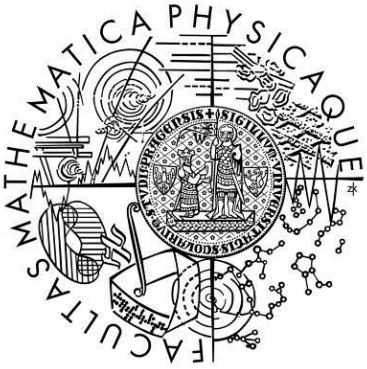
Univerzita Karlova v Praze

Matematicko-fyzikální fakulta

# DIPLOMOVÁ PRÁCE




Miroslav Vodolán

## Rozšířený editor komponentových architektur pro MEF

Katedra distribuovaných a spolehlivých systémů

Vedoucí diplomové práce: Mgr. Pavel Ježek, Ph.D.

Studijní program: Informatika

Studijní obor: Softwarové systémy

Praha 2014

# Poděkování

Rád bych poděkoval vedoucímu práce Mgr. Pavlu Ježkovi, Ph.D. 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 diplomovou 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..... Miroslav Vodolán

Název práce: Rozšířený editor komponentových architektur pro MEF

Autor: Miroslav Vodolán

Katedra / Ústav: Katedra distribuovaných a spolehlivých systémů

Vedoucí diplomové práce: Mgr. Pavel Ježek, Ph.D., Katedra distribuovaných a spolehlivých systémů

Abstrakt: Managed Extensibility Framework umožňuje vývoj komponentových aplikací v .NET. Vztahy mezi komponentami však mohou být složité. Z těchto důvodů byl v minulosti, jako součást autorovy bakalářské práce, vytvořen MEF Editor, který dokáže na základě analýzy zdrojového kódu tyto vztahy zobrazit a umožní jejich editaci. I když jsou možnosti editoru dané uživatelskými rozšířeními, které má k dispozici, existují situace, ve kterých tento editor použít nelze. Proto jsme v rámci této diplomové práce vytvořili novou verzi MEF Editoru, která přidává další možnosti použití. Editor s rozšířeními, která jsme implementovali, dokáže analyzovat nejen kompozice ve zdrojových kódech napsaných jazykem C#, ale také ve zkompilovaných assembly. V těchto aplikacích pomáhá odhalovat chyby kompozice a umožňuje provádět vizuální editace zdrojových kódu definující komponentovou architekturu těchto aplikací.

Klíčová slova: MEF, editor, Visual Studio 2012, komponentové aplikace

Title: Enhanced Editor of MEF Component Architectures

Author: Miroslav Vodolán

Department: Department of Distributed and Dependable Systems

Supervisor: Mgr. Pavel Ježek, Ph.D., 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. Therefore the MEF Editor was implemented in context of author's bachelor thesis, which can visualise the relations according to source code analysis and provide their editing. Although possibilities of the analysis are determined by available user's extensions, in some cases the editor cannot be used. This master thesis provides a solution in form of a new version of the MEF Editor which increases the number of cases it can be used in. As part of this thesis, we implemented the editor with extensions allowing analysis of application projects written in C# language and compiled assemblies. It helps to detect composition errors in these applications and allows visual editing of source code where component architecture of these applications is implemented.

Keywords: MEF, editor, Visual Studio 2012, component-based applications.

# Obsah

# 1 Úvod

Se vzrůstající komplexností aplikací dochází i ke zvyšování nároků na jejich vývoj a údržbu. Musíme totiž čelit problémům, které plynou z velikosti měřitelné například v počtu řádků psaného kódu takových aplikací. Existují situace, kdy by vývoj aplikace nebyl schopen zajistit jediný vývojář. Aplikace může být natolik rozsáhlá, že by byl její vývoj příliš dlouhý, kvůli omezené rychlosti psaní kódu, které je vývojář schopen. Případně může aplikace využívat tolik různých technologií, že je jejich zvládnutí pro jediného vývojáře nereálné.

Pro uvedené problémy vyplývají dvě různá řešení. První je triviální a spočívá v tom, že se vyhneme psaní příliš velkých aplikací. Je ovšem jasné, že v dnešní době je takový požadavek nesplnitelný.

Druhým řešením je rozdělení vývoje mezi větší množství vývojářů. Dohromady jistě dovedou pokrýt větší množství technologií. Chtěli bychom ale také, aby mohli pracovat pokud možno souběžně, čímž bychom zkrátili dobu nutnou pro vývoj aplikace. Z těchto důvodů je potřeba aplikaci rozdělit na části, na kterých mohou jednotliví vývojáři pracovat pokud možno nezávisle na ostatních. Prakticky toho docílíme rozčleněním aplikace na několik funkčních celků, kterým budeme říkat komponenty. Aby nám rozčlenění pomohlo, budeme však po komponentách požadovat několik vlastností.

Předně, komponenta by měla být dostatečně malá (kde velikost můžeme opět měřit v počtu řádků jejího kódu), aby ji mohl vyvíjet a udržovat jeden vývojář. Z toho také plyne požadavek na omezené množství použitých technologií, které bude komponenta využívat. Dalším požadavkem je již zmíněné minimalizování závislosti na jiných komponentách, které nám zajistí vzájemnou nezávislost práce jednotlivých vývojářů.

Typickým principem, který nám umožní tyto požadavky splnit, je zapouzdřenost[1] komponent. Ta nám říká, že komponenta by měla být závislá na okolí pouze v rámci jasně definovaných služeb, které smí využívat. Komponenta také definuje, které služby nabízí. Díky tomu můžeme komponenty mezi sebou spojovat bez ohledu na jejich vnitřní implementaci a tím efektivně rozdělit komplexní aplikaci.

O aplikaci složené z komponent říkáme, že má komponentovou architekturu. S tou souvisí rozšiřitelnost takových aplikací, neboť aplikaci poskládanou z jasně definovaných komponent nebývá problém rozšířit o další komponenty, které dokážou vylepšit nebo pozměnit její funkčnost. Stejně tak můžeme aplikaci vylepšit výměnou některých komponent za jiné, které poskytují stejné služby, ale s lepší implementací.

Komponentová architektura aplikací je v dnešní době často používaná. Ukážeme si proto několik příkladů aplikací s komponentovou architekturou, na kterých můžeme vidět popisované vlastnosti:

- **Microsoft Visual Studio [3]** – Samotné *Visual Studio* neumožnuje zpracování žádných jazyků ani projektů. Prostředí každého podporovaného jazyka je nahráváno pomocí komponent ve formě pluginů, které definují zvýraznění syntaxe jazyků, prostředky nutné pro kompilaci, apod. Také je

[1] Zapouzdřenost je běžně používaný koncept objektově orientovaného programování popsaný například na blogu MSDN [2]

možné nahráním pluginů přidat specializované nástroje pro analýzu kódu jako jsou například *Code Contracts [4]*.

- **Google Chrome [5]** – U internetových prohlížečů je komponentová architektura běžná. Výjimkou není ani *Google Chrome*. Komponenty v podobě rozšíření dovolují přidat nové funkce jako je třeba přehrávání speciálních formátů videa. Stejně tak je možné upravit vestavěné funkce jako například správa navštívených stránek.
- **Operační systém** – I operační systémy často mají komponentovou architekturu. U systému *Windows* ji můžeme vidět ve využití ovladačů, ze kterých se skládá funkční systém podle zařízení, na kterém je spuštěn. U Linuxových systémů můžeme rozšiřovat funkčnost pomocí samostatných kernelových modulů a tak například přidat podporu nových souborových systémů. Zvláštním případem je pak systém *HelenOS* [6]. Ten je vytvořen s důrazem rozčlenit i nejzákladnější části systému do nezávislých komponent.

Způsobů jak psát komponentovou aplikaci je několik. Nejjednodušší 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.

### Komponentové systémy v .NET

Jiným možným způsobem je publikování 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)* [11] a *Managed Extensibilty Framework (MEF)* [12].

MAF je určen pro skládání aplikací z izolovaných celků, které jsou nazývané *Addiny*. Izolace je zde na takové úrovni, že pád jednoho *Addinu* nemusí ovlivnit běh zbylých částí aplikace. Pro každý *Addin* je také možné určit rozdílná oprávnění. Implementace těchto možností v MAF si však žádá relativně velkou režii při komunikaci s *Addinem*. Proto není vhodné rozkládat aplikaci na velké množství *Addinů*. *Addin* tedy příliš nevyhovuje vlastnostem, které po komponentě požadujeme.

MEF je také určen pro skládání aplikace z několika celků. Proti MAF se ale nesnaží tyto celky vzájemně izolovat. Naopak jejich skládání probíhá na úrovni jednotlivých objektů, takže jejich vzájemná komunikace není zatížena žádnou dodatečnou režií.

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.

## 1.1 Základní principy 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 nějaké rozhraní. Na druhé straně, exportem může být jak datová položka, tak celá třída, z níž je získán objekt použitelný pro naplnění nějakého importu.

Na následujícím obrázku si ukážeme příklad, jak vypadá definice importů a exportů ve zdrojovém kódu. Všimněme si atributu `Export` na třídě `NormLayout`, který označuje, že je třída exportovaná. Atributy `Import` a `ImportMany` naopak označují položky třídy, do kterých bude při kompozici přiřazen odpovídající exportovaný objekt.

```
using System.ComponentModel.Composition;
namespace Extensions
{
    [Export(typeof(ILayout))]
    public class NormLayout:ILayout
    {
        [Import]
        public ILogger Logger;

        [ImportMany(typeof(IContent))]
        public IEnumerable<IContent> Contents;
    }
}
```

*1-1 Příklad způsobu definice komponenty pomocí MEF. Označení importů a exportů se provádí pomocí atributů.*

Tento příklad slouží pro ilustraci způsobu, jak se v MEF komponenty definují. Podrobný popis principů a konceptů, které MEF nabízí, je dále v kapitole 1.2.

### Vyhledávání komponent

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. Pro objevování komponent v MEF slouží takzvané katalogy. K dispozici máme několik připravených katalogů, které zajišťují vyhledávání komponent v souborech knihoven nebo v již spuštěných assembly.

Díky tomu, že si aplikace vytváří katalogy stejně jako běžné objekty až za běhu, může si nahrávání komponent snadno přizpůsobit podle uživatelských nastavení, nebo podle zrovna prováděných akcí.

### Kompozice v MEF

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 jí 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

výjimky. Ve složitějších aplikacích navíc můžeme ztratit přehled o celkové architektuře, což negativně ovlivňuje další vývoj.

Také provádění úprav ve zdrojovém kódu zajišťujícím kompozici může být nepřehledné. Definice kompozice totiž může obsahovat netriviální množství zdrojového kódu, ve kterém není snadné se zorientovat.

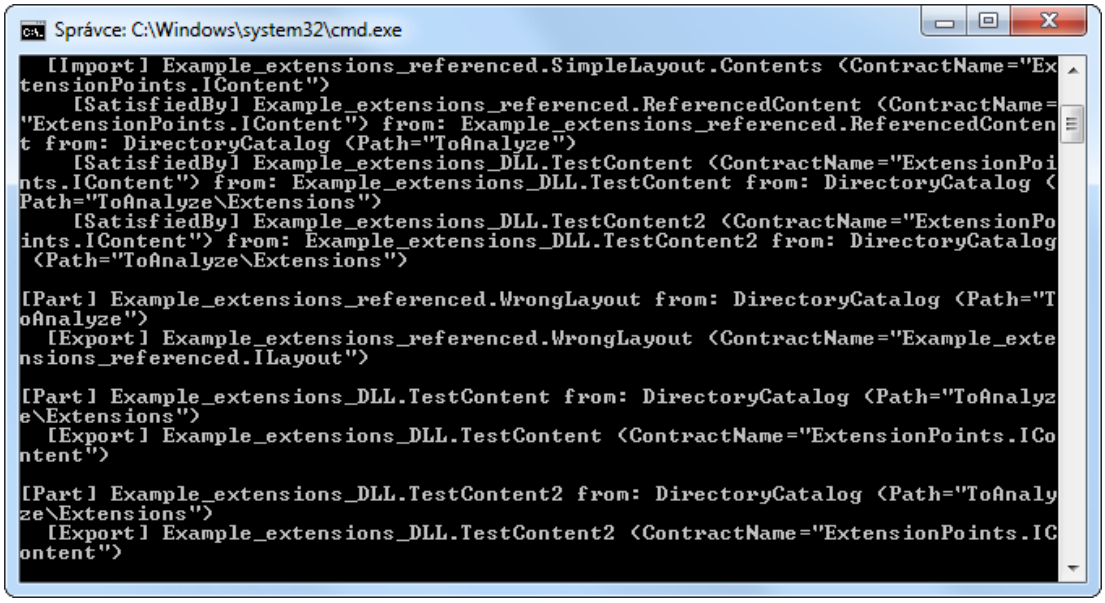
### Nástroje usnadňující použití MEF

Za účelem ladění MEF kompozice již bylo vyvinuto několik nástrojů. Nejznámějšími jsou *Mefx* [14] a jeho vizuální podoba *Visual MEFX* [13]. Pomocí těchto nástrojů si můžeme nechat zobrazit definované komponenty a možné chyby vzniklé při kompozici. Tyto nástroje neposkytují přímý náhled na kompozici, kterou však dokáže zobrazit *MEF Visualizer Tool* [15] a nebo *MEF Editor* [1], jenž je výsledkem bakalářské práce stejného autora jako tato diplomová práce.

Vývoj aplikací využívajících MEF usnadní nejen nástroje pro ladění kompozice. Kvůli množství zdrojového kódu, který definuje kompozici, oceníme také nástroj, jenž editace tohoto kódu zjednoduší. Jediným nástrojem, který v době zadání této práce dokázal provádět editace kompozice pomocí změn zdrojového kódu, byl *MEF Editor*.

Na následujících příkladech si ukážeme princip použití jednotlivých nástrojů a také jejich omezení.

### Mefx




*1-2 Konzolový výstup nástroje Mefx.*

Nástroj *Mefx* slouží pro zjištění chyb v kompozicích zkompilovaných assembly. Ovládá se pomocí příkazové řádky a dokáže poskytnout informace o existujících komponentách, jejich importech a exportech. Díky tomu se může pokusit zjistit, zda je možné provést kompozici všech objevených komponent. Tím dokáže odhalit některé chyby, které mohou při kompozici vzniknout.

Na druhou stranu mohou být výsledky analýzy matoucí, neboť vůbec nezkoumá kompoziční algoritmus analyzované aplikace. Kvůli tomu může do kompozice zahrnout i komponenty, které se do ní dle kompozičního algoritmu nedostanou.

Další nevýhodou je obtížná použitelnost v rozpracovaných projektech. Vzhledem k tomu, že nástroj pracuje pouze se zkompilovanými assembly, není

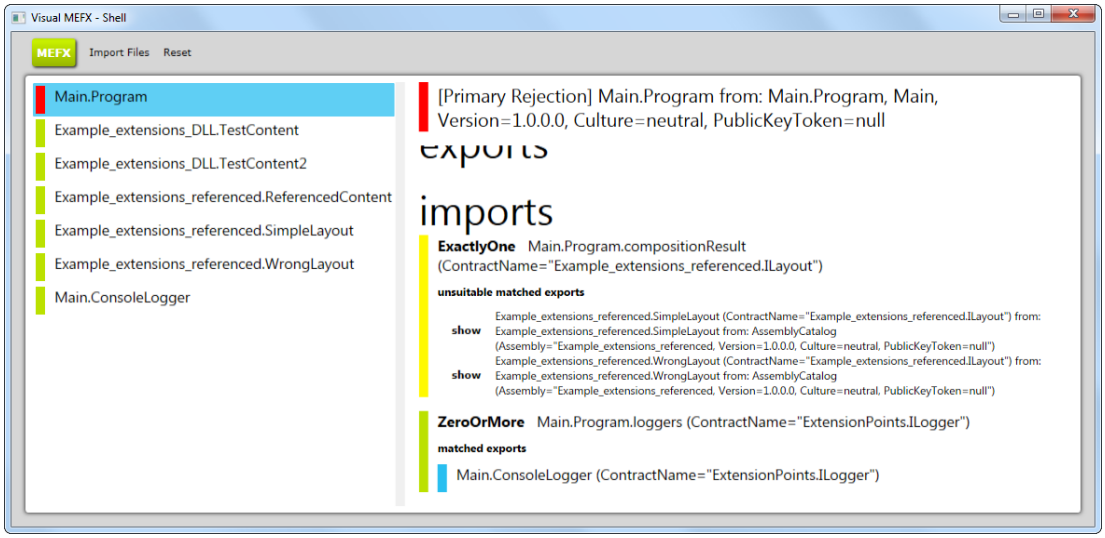
pomocí něj možné odhalit chyby kompozice v průběhu vývoje aplikace, kdy aplikace nemusí být ještě natolik dokončená, aby ji bylo možné zkompilovat.

Absence grafického rozhraní také negativně ovlivňuje přehlednost a použitelnost *Mefx*. Z tohoto důvodu bylo vytvořeno vizuální rozšíření *Visual MEFX*, které si klade za cíl zpřehlednit výsledky analýzy.

**Shrnutí vlastností:**

- ✗ Nemá grafické rozhraní
- ✗ Neanalyzuje kompoziční algoritmus
- ✗ Vyžaduje zkompilované assembly
- ✗ Nepodporuje editace kompozice

## Visual MEFX




*1-3 Ukázka výstupu nástroje Visual MEFX*

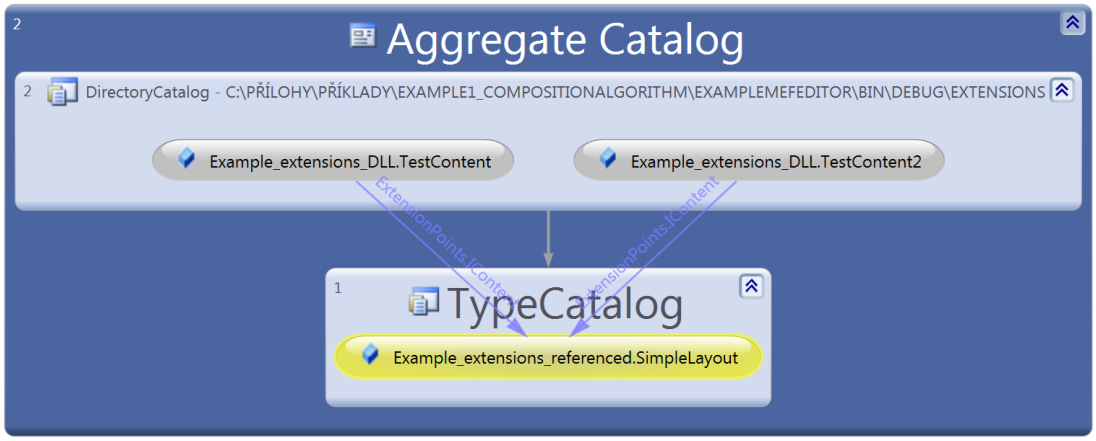
*Visual MEFX* pro analýzu využívá nástroj *Mefx*, platí proto pro něj stejné možnosti využití i stejná omezení. I *Visual MEFX* tedy umí pracovat pouze se zkompilovanými assembly. Poskytovat dokáže stejné informace jako *Mefx*, ale ve výrazně přehlednější grafické formě.

Z nástroje se však nedozvíme žádné podrobnosti o skutečné kompozici, neboť *Mefx* takové informace neposkytuje. Tento problém řeší *MEF Visualizer Tool*, který dokáže zkoumat skutečnou kompozici.

**Shrnutí vlastností:**

- ✓ Má grafické rozhraní
- ✗ Neanalyzuje kompoziční algoritmus – sdílí s *Mefx*
- ✗ Vyžaduje zkompilované assembly – sdílí s *Mefx*
- ✗ Nepodporuje editace kompozice

### MEF Visualizer Tool




*1-4 Použití MEF Visualizer Tool vyžaduje změny ve zdrojovém kódu, následnou kompilaci a spuštění aplikace.*

Zobrazení schématu kompozice dokáže obstarat nástroj *MEF Visualizer Tool*. Na rozdíl od *Mefx* a *Visual MEFX* však nedokáže pracovat s libovolnými zkompilovanými assembly. Pro použití tohoto nástroje je totiž nutné upravit zdrojový kód aplikace. Ta totiž musí za pomoci metod poskytovaných *MEF Visualizer Tool* vypsat graf kompozice ve formátu DGML[2] [7]. Ten je následně možné zobrazit jako přehledný graf, kde vidíme vztah mezi importy a exporty.

Použití tohoto nástroje je pro vývoj ještě komplikovanější, než u *Mefx* nebo *Visual MEFX*, neboť analýza nejenže vyžaduje spustitelnou aplikace, kterou je nutné při analýze spouštět, ale navíc nutí vývojáře upravovat zdrojový kód.

**Shrnutí vlastností:**

- ✓ Má grafické rozhraní
- ✓ Analyzuje kompoziční algoritmus
- ✗ Vyžaduje zkompilované assembly
- ✗ Nepodporuje editace kompozice
- ✗ Vyžaduje úpravy zdrojových kódů

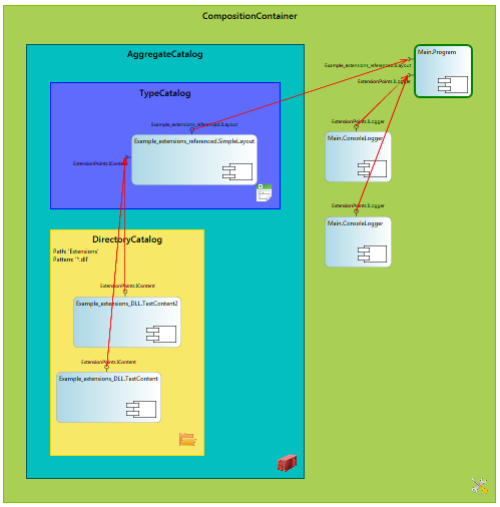
[2] Directed Graph Markup Language – Jedná se o jazyk určený pro definici zobrazení orientovaných grafů.

### MEF Editor (předchozí bakalářská práce autora této diplomové práce)

```
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);
    }
```




*1-5 Metoda Compose provádí kompozici, která je přehledně znázorněna na schématu vpravo.*

S cílem poskytnout nástroj pro analýzu MEF kompozice, který by nebyl omezen pouze na zkompilované assembly, byl v minulosti vyvinut *MEF Editor* v rámci bakalářské práce stejného autora, jako je tato diplomová práce. Tento editor lze nainstalovat do *Microsoft Visual Studia 2010* jako plugin, který dokáže analyzovat zdrojové kódy rozpracovaných projektů. Na základě analýzy pak dokáže zobrazit schéma kompozice a případné chyby. Nad schématem kompozice také umožňuje provádět vizuální změny, které se okamžitě projeví patřičnou editací zdrojových kódů.

I tento nástroj má nedostatky, které znemožňují jeho použití v některých projektech. V rámci této práce se je pokusíme odstranit a rozšířit tak možnosti použití *MEF Editoru*. Abychom však pochopili jeho omezení, musíme si v úvodních kapitolách podrobněji představit vlastnosti a použití MEF. Poté si ukážeme situace, se kterými se můžeme při vývoji aplikací využívajících MEF setkat. Podle nich poznáme, jak nám při vývoji mohou pomoct editace kompozice. V rámci připomenutí schopností *MEF Editoru* a našich zkušeností z vývoje editoru si pak ukážeme jeho limity. Na jejich základě také sestavíme cíle pro tuto práci.

**Shrnutí vlastností:**

- ✓ Má grafické rozhraní
- ✓ Analyzuje kompoziční algoritmus
- ✓ Nevyžaduje zkompilované assembly
- ✓ Podporuje editace kompozice
- ✓ Nevyžaduje úpravy zdrojových kódů
- ✗ Nelze použít v některých projektech

## 1.2 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.

Vzhledem k množství těchto konceptů je rozdělíme na hlavní a vedlejší. V rámci této práci se budeme soustředit na podporu analýzy všech hlavních konceptů. Analýzu však budeme navrhovat i s ohledem na možnou rozšiřitelnost o podporu konceptů vedlejších.

### Hlavní koncepty v MEF

- *Naplnění Importů*

  Z úvodní kapitoly 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 vlastnosti. 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 výjimky.

- *Importující konstruktor*

  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 takový konstruktor zavolán s argumenty, získanými z dostupných exportů. V kódu ho určíme následovně:

```
class NormLayout
{
   [ImportingConstructor]
   public NormLayout(ILogger logger, [ImportMany]IContent[] contents)
   {
      ...
   }
}
```

*1-6 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.

- *Import několika exportů*

  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;
   ...
}
```

*1-7 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*

- *Export a import metadat*

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.

- *Export s dědičností*

Abychom nemuseli označovat exportem stejné položky pro každého potomka odvozeného od nějaké třídy, můžeme využít atribut `InheritedExport`. Pokud dědíme od takto označené třídy, nemusíme u odvozené třídy `Export` použít.

- *Katalogy komponent*

Pro vyhledávání komponent slouží katalogy, kterých MEF nabízí několik druhů. Zde je jejich výčet:

- `DirectoryCatalog` – vyhledává assembly podle zadané cesty a vzoru, poté poskytuje komponenty všech nalezených assembly,
- `AssemblyCatalog` – poskytuje komponenty nahrané assembly, nebo assembly na zadané cestě,
- `TypeCatalog` – poskytuje komponenty, které mají zadaný typ,
- `AggregateCatalog` – sdružuje komponenty všech katalogů, které do něj přidáme.

- *Kompoziční kontejner*

Naplnění importů z dostupných exportů zajišťuje třída `CompositionContainer`. Komponenty dokáže načítat ze zadaného katalogu. Kompozici může ovlivnit objekt typu `CompositionBatch`, který dokáže do kompozice přidat, nebo z ní vyřadit konkrétní komponenty.

Kompoziční kontejner nabízí několik způsobů jak kompozici provést. Uvedeme hlavní z nich:

- `Compose` – provede kompozici, za použití dostupného katalogu a zadaného `CompositionBatch`,
- `ComposeParts` – zkomponuje zadané komponenty, spolu s komponentami v dostupném katalogu,
- `SatisfyImportsOnce` – naplní importy zadané komponenty z exportů v dostupném katalogu.

### Vedlejší koncepty

- *Export factory*

  Tovární třída `ExportFactory` umožňuje lépe kontrolovat objekty vytvářené pro export. Pokud nám nevyhovuje způsob, jak je vytváří MEF, můžeme s použitím této třídy vytváření řídit sami.

- *Komponenty podle konvencí*

  Za pomoci třídy `RegistrationBuilder` můžeme do kompozice přidávat komponenty, aniž by musely být označené příslušnými atributy. Jejich importy a exporty tedy můžeme definovat podle konvencí, které se nám v aplikaci hodí.

- *Vlastní atribut pro metadata*

  MEF umožňuje definovat vlastní atribut pro export metadat. Vytvářený atribut musíme označit pomocí `MetadataAttributeAttribute`.

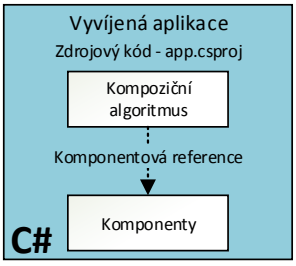
## 1.3 Vývoj aplikací s použitím MEF

Při vyvíjení komponentové aplikace s použitím MEF můžeme rozlišit několik vývojových konfigurací podle vztahů vyvíjené aplikace s referencovanými knihovnami, umístěním kompozičního algoritmu a definic komponent. Každou vývojovou konfiguraci je možné zařadit do jedné ze čtyř kategorií, které si představíme.

V konfiguracích rozlišujeme dva různé typy referencí. Prvním typem jsou běžně používané reference na assembly, které referencující assembly umožní využívat třídy referencované assembly. Druhý typ referencí definujeme pro potřeby této práce. Jedná se o *komponentové reference* vyjadřující vztah mezi kompozičním algoritmem a komponentami, které jsou tímto algoritmem komponovány. Poznamenejme, že *komponentové reference* jsou kompozičním algoritmem objevovány až za běhu aplikace.

Následuje seznam kategorií pro vývojové konfigurace. U každé kategorie uvedeme ilustrační obrázek. Na každém ilustračním obrázku bude zobrazena modrou barvou vyvíjená aplikace, od níž máme zdrojové kódy. Žlutě pak zobrazujeme zkompilované assembly, které jsou dostupné v podobě instrukčního jazyka CIL [8]. Seznam je uspořádán vzestupně se vzrůstající komplexností. Proto kategorie s vyšším číslem může obsahovat konstrukce z nižších kategorií.

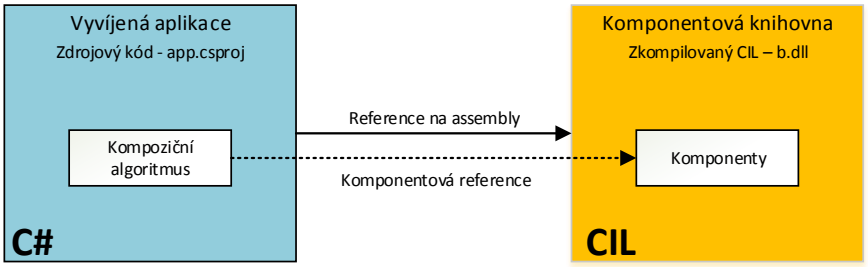
1. Nejjednoduší konfiguraci představuje vývoj aplikace obsahující všechny použité komponenty spolu s kompozičním algoritmem. Tato situace je obvyklá při vývoji nové rozšiřitelné aplikace s předpřipravenými nebo standardními pluginy. Následuje obrázek 1-8 s příkladem takové aplikace.




*1-8 Vyvíjená aplikace obsahující komponenty i kompoziční algoritmus*

Na tomto obrázku vidíme, že všechna kompoziční primitiva jsou dostupná ve zdrojových kódech vyvíjené aplikace.

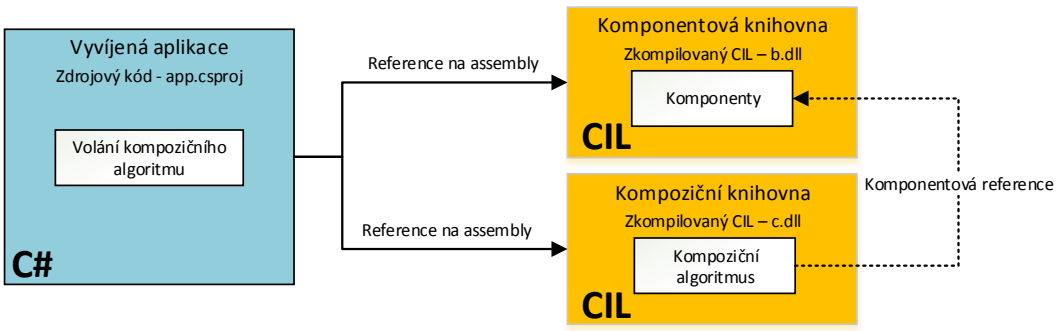
2. Další konfiguraci můžeme pozorovat u aplikace, která definuje kompozici, ale používá komponenty z již zkompilovaných referencovaných assembly. Toto je typické pro aplikace vyvíjené z již implementovaných funkčních bloků definovaných v knihovnách. Na následujícím obrázku 1-9 můžeme vidět příklad uvedené konfigurace.




*1-9 Vyvíjená aplikace neimplementuje komponenty, které používá*

Z tohoto obrázku je patrné, že pouze kompoziční algoritmus je dostupný ve zdrojových kódech. Použité komponenty jsou implementované ve zkompilované knihovně *b.dll*.

3. Vyjmutím kompozičního algoritmu z vyvíjené aplikace do nové referencované knihovny *c.dll* dostaneme další konfiguraci. Ta je typická pro případy kdy máme knihovnu definující kompoziční algoritmy. Příkladem takového algoritmu může být vyhledání pluginů ve všech referencovaných assembly. Tento algoritmus pak může být snadno zavolán z vyvíjené aplikace. Na následujícím obrázku 1-10 můžeme vidět popsanou konfiguraci s knihovnou *b.dll* obsahující komponenty.

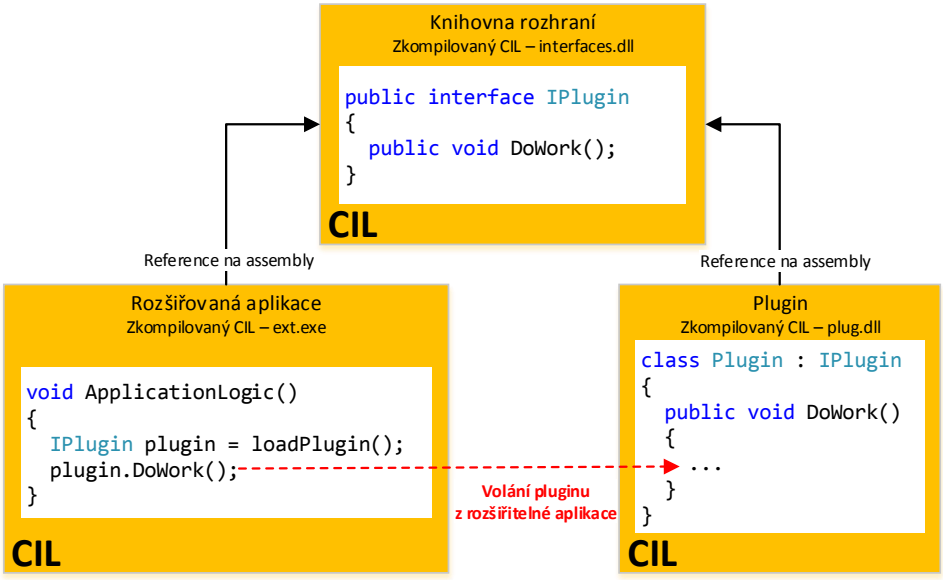



*1-10 Vyvíjená aplikace bez implementovaného kompozičního algoritmu*

Z tohoto obrázku vidíme, že ve zdrojových kódech již nejsou dostupná žádná kompoziční primitiva. Nicméně kompozici můžeme stále detekovat díky volání kompozičního algoritmu ze zdrojových kódů.

4. Poslední a nejkomplexnější konfiguraci budeme v rámci práce nazývat *Rozšiřování Existující Aplikace* (REA). Od předcházející konfigurace se liší v tom, že assembly implementující kompoziční algoritmus nemusí být vůbec referencovaná. Typickou situací, kde se můžeme s REA setkat, je psaní pluginu pro rozšiřitelnou aplikaci. Plugin v takovém případě obvykle nemá referenci na rozšiřovanou aplikaci. Přesto však musí referencovat nějakou knihovnu se

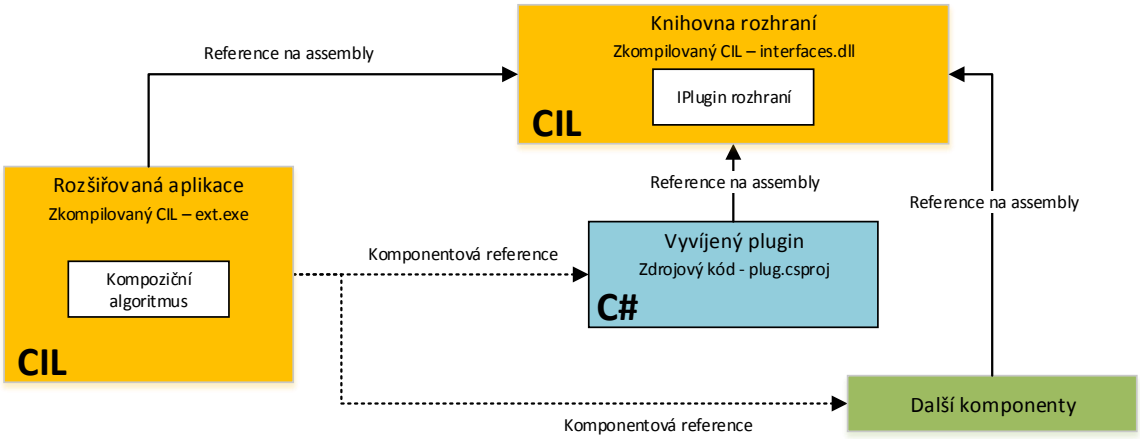
společným rozhraním, přes které ho rozšiřovaná aplikace dokáže využívat. Z těchto důvodů rozšiřitelné aplikace definují knihovny rozhraní, které musí každý rozšiřující plugin referencovat. Na následujícím obrázku si ukážeme příklad takové konfigurace. Všimněme si rozšiřované aplikace *ext.exe*, která využívá služby pluginu díky rozhraní `IPlugin`, které je díky referencím na knihovnu *interfaces.dll* známé jak rozšiřované aplikaci, tak pluginu *plug.dll*.




*1-11 Ukázka závislostí mezi rozšiřovanou aplikací, pluginem a společnou knihovnou rozhraní*

V průběhu vývoje pluginu je tedy z celé rozšiřované aplikace známá pouze knihovna *interfaces.dll*, neboť je pluginem referencována. Nicméně tato knihovna typicky neobsahuje žádnou informaci o tom, jak bude rozšiřovaná aplikace *ext.exe* komponovaná. V následujícím obrázku 1-12 si probíranou konfiguraci rozebereme z pohledu kompozice při vývoji pluginu.

Ze zobrazených referencí můžeme vidět, že ze zdrojových kódů pluginu nedokážeme vyčíst závislost mezi pluginem a *ext.exe*. Jediný vztah, který spojuje tyto assembly, je daný kompozičním algoritmem v *ext.exe* a je v obrázku znázorněn *komponentovými referencemi*.



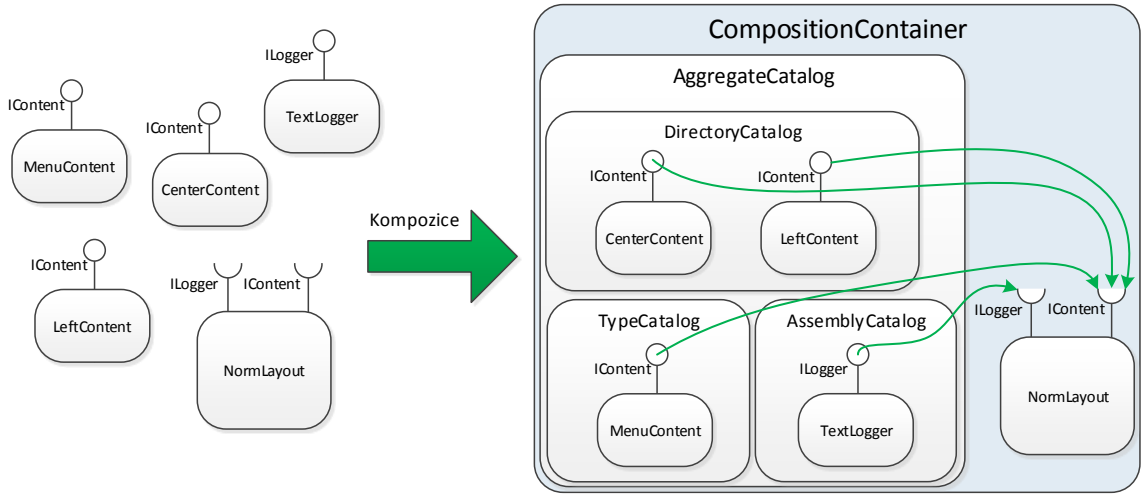

*1-12 Rozšiřovaná aplikace neumožňuje vyvíjenému pluginu zjistit kompozici*

Jak můžeme na tomto obrázku vidět, tak ze zdrojových kódů nezjistíme nic o kompozičním algoritmu ani zúčastněných komponentách. Zobrazené *komponentové reference* pouze vizualizují abstrakci poskytovanou kompozičním algoritmem.

## 1.4 Editace kompozice

V předchozí kapitole jsme zjistili, že při vývoji aplikace s použitím MEF můžeme rozlišovat několik vývojových konfigurací. Z hlediska nabízení editací jsou však významné pouze konfigurace 1. a 2., neboť jen u nich dochází k vývoji kompozičního algoritmu, který chceme editovat.

Při vývoji kompozičního algoritmu řešíme úlohu, kdy máme nalézt a zkomponovat všechny potřebné komponenty. Tuto situaci ilustruje následující obrázek:




*1-13 Při kompozici aplikace musí uživatel zajistit načtení komponent z různých katalogů. Pro jejich použití v CompositionContainer navíc musí všechny katalogy vložit do AggregateCatalog.*

Jak jsme již popisovali v předchozích kapitolách, vyhledání komponent provádíme v MEF pomocí katalogů, které můžeme vidět na obrázku. Samotnou kompozici pak provede `CompositionContainer`, který však vyžaduje sdružení katalogů do jednoho společného. Zdrojový kód takové kompozice může vypadat následovně:

```
public void Compose()
{
  //abychom nalezli potřebné komponenty, vytvoříme
  //patřičné katalogy
  var catalog1 = new DirectoryCatalog("./Extensions");
  var catalog2 = new TypeCatalog(typeof(MenuContent));
  var catalog3 = new AssemblyCatalog("Logging.dll");

  //pro použití v kompozičním kontejneru musíme
  //všechny katalogy sdružit do jednoho katalogu
  var commonCatalog = new AggregateCatalog();
  commonCatalog.Catalogs.Add(catalog1);
  commonCatalog.Catalogs.Add(catalog2);
  commonCatalog.Catalogs.Add(catalog3);

  //vytvoříme kompoziční kontejner
  var container = new CompositionContainer(commonCatalog);

  //a provedeme kompozici aplikace
  var layout = new NormLayout();
  container.ComposeParts(layout);
}
```

*1-14 Kód kompozice zachycené na obrázku 1-13.*

Všimněme si, že algoritmus kompozice obsahuje množství opakujícího se kódu. Příkladem může být postupné vytváření jednotlivých katalogů nebo jejich přidávání

do společného katalogu opakovaným voláním metody `AggregateCatalog.Catalogs.Add`. Proto by bylo vhodné mít nástroj, který by vývojáře ušetřil od nutnosti takový kód psát ručně.

Jiný typ editací zase spočívá ve změnách vlastností jednotlivých katalogů. Pokud například chceme změnit cestu pro načítání rozšíření, musíme nejprve nalézt patřičný `DirectoryCatalog`, a poté ručně přepsat cestu v textovém řetězci. I zde by se proto hodila asistence nějakého nástroje.

## 1.5 Připomenutí předchozí verze

Předchozí verze *MEF editoru* byla implementována v rámci bakalářské práce [1] stejným autorem jako tato diplomová práce. Spolu s *doporučenými rozšířeními* umožňuje analýzu a editace obvyklých konstrukcí MEF v projektech psaných jazykem C#. Editor byl však koncipován s ohledem na snadnou rozšiřitelnost o další .NET jazyky.

### Podpora Visual Studia

Editor je použitelný ve formě pluginu pro *Microsoft Visual Studio 2010*. Ve verzi *Microsoft Visual Studio 2012*, která byla aktuální v době vypsání této diplomové práce, však editor použít nelze. Důvodem je nutnost změny konfigurace při kompilaci pluginu tak, aby plugin bylo možné nainstalovat i do verze *Microsoft Visual Studio 2012*. Nicméně při vývoji editoru nebyla funkčnost v této verzi *Visual Studia* testována a není proto zajištěna bezproblémová použitelnost.

### Princip fungování analýzy

Způsob, jakým předchozí verze analyzuje zdrojové kódy, je založen na sběru dat v průběhu interpretování metod důležitých z hlediska MEF kompozice. Interpretování se provádí nad tzv. sémantickými stromy, které jsou získávány ze syntaktických stromů doplněním informací o typech a metodách. Syntaktické stromy pak dostaneme jako výsledek parsování.

V průběhu interpretace musí interpret explicitně udávat informace, které umožní zjistit schéma kompozice a které budou použity pro tvorbu případných editací. Příkladem takových informací může být informace o tom, že byla vytvořena komponenta, nebo informace o umístění volání ve zdrojovém kódu, které komponentu přidalo do kontejneru.

### Podpora analýzy vývojových konfigurací

V následujícím seznamu si připomeneme schopnosti předchozí verze editoru analyzovat vývojové konfigurace z kapitoly 1.3:

- *Konfigurace 1. typu – Podporována*

Součástí implementace předchozí verze je syntaktický a sémantický parser pro C#. Implementován je také interpret, který dokáže zpracovávat sémantické stromy. Poznamenejme, že tento interpret může být použit i na další jazyky, které mají podobnou sémantiku jako C#.

S těmito moduly je možné analyzovat vývojové konfigurace 1. typu. Stačí rozparsovat metody týkající se kompozičního algoritmu a následně na něm spustit interpretaci. Ze získaných výsledků je editor schopen zobrazit schéma kompozice a provádět nad ním editace.

- *Konfigurace 2. typu – Podporována*

  Analýza konfigurací 2. typu je také možná a to díky přítomnosti modulů, které zajišťují načítání meta informací z referencovaných assembly. Součástí těchto meta informací jsou totiž i informace o definicích komponent. Díky tomu je možné jejich zobrazení ve schématu kompozice.

- *Konfigurace 3. typu – Možné přidat podporu*

  U konfigurací 3. typu je nutné analyzovat kompoziční algoritmus, který je dostupný ve formě zkompilovaného CIL kódu. Aby byl předchozí editor schopen této analýzy, bylo by nutné implementovat nový interpret, který by zpracovával instrukce ze zkompilovaného kódu, neboť jeho sémantika je od C# natolik odlišná že není snadné ho převést do tvaru společného sémantického stromu. Připomeňme, že implementace interpretu je kvůli nutnosti poskytovat informace pro analýzu relativně náročná. Každý interpret si navíc musí reprezentovat vlastní běhové prostředí s informacemi o proměnných, zásobníkem volání, inicializovaných statických tříd, apod.

- *Konfigurace 4. typu (REA) – Nepodporována*

  Přidání interpretu by však neřešilo problém s analýzou konfigurací REA, neboť předchozí verze editoru vůbec nepočítá s analýzou assembly, která není z rozpracovaného projektu nijak referencovaná. Taková úprava by vyžadovala zásadní změny v architektuře editoru.

### Vykreslování schématu kompozice

Po provedení analýzy editor zobrazí schéma kompozice, kde může provádět různé editace usnadňující vývoj aplikace. Ve schématu jsou zobrazeny kompoziční kontejnery, katalogy a v nich nalezené komponenty. Schéma zachycuje i vztahy mezi importy a exporty komponent pomocí spojnic.

Editor nabízí základní rozmístění položek zobrazených na schématu kompozice. Uživatel má však možnost jejich rozmístění libovolně upravovat. S tím však souvisí problém s možným vznikem nepřehledných situací, kdy se některé zobrazené položky překrývají.

Druhým problémem, který snižuje přehlednost schématu je množství spojnic mezi importy a exporty komponent, které protínají zobrazované položky. Bylo by proto dobré vylepšit algoritmus vykreslování tak, aby zamezil překrývání položek a jejich protínání spojnicemi.

## 1.6 Zkušenosti z vývoje předchozí verze

Předchozí verze byla vyvíjena pouze jako plugin *Visual Studia*, který nebylo možné spustit samostatně. Tato vlastnost měla negativní dopad na laditelnost a testovatelnost editoru. Nebylo totiž možné použít automatizovaný testovací framework, neboť by každý test vyžadoval spuštění nové instance *Visual Studia*.

### Ladění předchozí verze

Veškeré ladění proto probíhalo pomocí ručně vyvolávaných akcí ve *Visual Studiu*. Tento způsob byl časově velmi náročný, neboť samotná doba spuštění *Visual Studia* a načtení testovacího projektu trvala téměř jednu minutu. Tento problém by řešil lepší počáteční návrh architektury, který by umožňoval spouštění editoru i jiným způsobem, než jako plugin *Visual Studia*.

Ještě složitější bylo ladění sekvencí různých úkonů, neboť je bylo nutné vždy ručně opakovat. Pokud by však architektura editoru umožňovala vytvoření

frameworku pro automatické unit testování, nebylo by ruční opakování sekvencí úkonů vůbec potřebné.

### Vývoj analýzy

Analýza je v předchozí verzi založena na parsování zdrojových kódů, jejich převedení na instrukce nějakého interpretu a jejich následná interpretace. Pokud analýza podávala nesprávné výsledky, bylo nutné hledat chyby v každé fázi analýzy zvlášť.

Hledání chyb v parsování spočívalo ve zkoumání stromových struktur, představujících rozparsovaný zdrojový kód, přes standardní ladící nástroje. Výhodnější by však bylo zobrazení ladícího výpisu tak, aby bylo možné porovnat výsledek parsování s parsovaným zdrojovým kódem.

Pokud chyba nebyla nalezena ve fázi parsování, přichází na řadu zkontrolovat správnost interpretace. Zde je nutné ověřit správnost přiřazovaných hodnot do proměnných, v reprezentaci běhového prostředí. Tento způsob ladění opět není příliš pohodlný. Oba problémy by řešila úprava architektury editoru tak, aby bylo možné vytvářet ladící výpisy z jednotlivých fází analýzy.

### Vykreslování schématu kompozice

Předchozí verze editoru obsahovala algoritmus, který dokázal přehledně uspořádat zobrazované komponenty, katalogy a kontejnery. Ladění tohoto algoritmu bylo opět problematické. Vzhledem k přílišné provázanosti vykreslování schématu s analýzou opět nebylo možné provádět testování vykreslování jinak, než spuštěním editoru v nové instanci *Visual Studia*.

Oddělením vykreslování schématu kompozice od části editoru, která řeší analýzu, by opět pomohlo k lepší testovatelnosti editoru.

## 1.7 Cíle projektu

V době zadání této diplomové práce nebyl dostupný žádný nástroj, který by dokázal přehledně zobrazit a editovat schéma kompozice rozpracovaných projektů s nekompletními zdrojovými kódy, kromě předchozí verze *MEF editoru* implementované autorem této diplomové práce. Předchozí verze dokáže zobrazovat schéma kompozice pouze u aplikací s kompozičním algoritmem definovaným ve zdrojových kódech.

Rozšířením o patřičný interpret by bylo možné přidat podporu zobrazování schématu aplikací s kompozičním algoritmem definovaným ve zkompilovaných projektech. Nicméně toto rozšíření by stejně nevyřešilo problém s podporou analýzy REA. Z tohoto důvodu je nutná reimplementace předchozí verze *MEF editoru,* aby dokázala pokrýt analýzu všech konfigurací vývojového prostředí, které byly uvedeny.

Hlavním cílem této diplomové práce je rozšíření předchozí verze MEF editoru tak, aby dokázala splnit následující funkční požadavky:

1) *Analýza kompozice definovaná ve zkompilovaných assembly*
2) *Schopnost analyzovat aplikaci v konfiguraci REA*
3) *Provádění editací kompozičního algoritmu ve zdrojových kódech*
4) *Vylepšení způsobu vykreslování schématu kompozice*
5) *Integrace editoru do Microsoft Visual Studio 2012*

6) *Poskytování schématu kompozice ze zdrojových kódů aplikace otevřené ve Visual Studiu*

Pokrýt analýzu všech konstrukcí technologií MEF a .NET by bylo nad rámec této práce. Z tohoto důvodu implementujeme podporu pro hlavní koncepty MEF z kapitoly 1.2 a potřebné konstrukce jazyka C#. Ty budou dostupné v knihovnách, které budeme v rámci této práce nazývat *doporučená rozšíření*.

Editor však bude koncipován s ohledem na další snadnou rozšiřitelnost. Podporu dalších konstrukcí a technologií bude uživatel schopen přidat dle ukázkových příkladů.

Z toho plynou další funkční požadavky pro tuto práci:

7) *Podpora hlavních konceptů MEF*
8) *Podpora jazyka C#*
9) *Uživatelská rozšiřitelnost editoru*

---

Ze zkušeností s vývojem předchozí verze editoru jsme zjistili, že pro efektivní a udržitelný vývoj je nutné dbát na dobrou testovatelnost vyvíjeného editoru.

Proto na vyvíjený editor máme ještě tyto kvalitativní požadavky:

A) *Části editoru budou testovatelné pomocí frameworku unit testů*
B) *Pro vývoj rozšíření bude dostupná konzole s přehlednými ladícími výpisy*
C) *Editor bude možné spustit i mimo Microsoft Visual Studio*

## 1.8 Struktura diplomové práce

V úvodu této práce jsme se seznámili s problematikou komponentových architektur aplikací. V kapitole 1.1 jsme uvedli základní principy MEF a představili jsme si nástroje, které nám mohou ulehčit jeho použití. Zjistili jsme, že pouze předchozí verze našeho editoru nabízí možnost editací schématu kompozice. I přesto má nedostatky, které omezují možnosti jeho použití.

Z rozboru těchto nedostatků jsme pak v kapitole 1.7 sestavili cíle pro tuto diplomovou práci. Pro jejich dosažení nejprve provedeme podrobnou analýzu, která začíná kapitolou 2.1 s analýzou typických využití MEF, která nám umožní definovat schéma kompozice a jeho vlastnosti. V navazující kapitole 2.2 pro toto schéma navrhneme editace, které je vhodné uživateli nabídnout.

Práce dále pokračuje kapitolou 2.3 se zamyšlením nad možnostmi analýzy nejsložitější vývojové konfigurace REA. V kapitolách 2.4 a 2.5 budeme zjišťovat možnosti implementace analýzy prováděné naším editorem. Se znalostí principu analýzy pak v kapitole 2.6 navrhneme způsob, jak vytvářet editace schématu kompozice.

V průběhu práce zjistíme, že je nutné navrhnout vlastní instrukční sadu. Její návrh a typový systém jsou popsány v kapitolách 2.7 a 2.8. Nedílnou součástí editoru bude vykreslování schématu kompozice, kterým se zabývá kapitola 2.9.

Ze zkušeností z předchozí verze víme, že je důležité dbát na dobrou testovatelnost editoru. V kapitole 2.10 se proto zamyslíme nad požadovanými

vlastnostmi testovacího frameworku. Analýzu zakončíme kapitolou 2.11, ve které shrneme předchozí závěry, abychom navrhli architekturu našeho editoru. Protože editor bude koncipován jako rozšíření *Microsoft Visual Studia*, shrneme si v kapitole 3 možnosti jeho rozšiřitelnosti.

Klíčovou částí práce potom bude kompletní reimplementace, která je popsána v kapitole 4. Reimplementujeme téměř všechny části předchozí verze editoru tak, aby bylo dosaženo uvedených cílů, zlepšení architektury implementace a s tím souvisejícími možnostmi usnadnit rozšiřitelnost editoru a zvýšit rychlost analýzy.

V kapitole 5 uživatele provedeme použitím editoru na konkrétním projektu. Nakonec představíme nové koncepty vylepšující možnosti rozšiřitelnosti editoru v kapitole 6. Zde také bude implementováno ukázkové rozšíření, představující nové možnosti rozšiřitelnosti v praxi.

Práci zakončíme shrnutím našich výsledků v kapitole 7.

## 1.9 Použité typografické konvence

V textu práce jsou použity následující typografické konvence:

- *Definovaný termín* – Ustálený název nebo termín definovaný v rámci této práce.
- `Zdrojový kód` – Třída, identifikátor, hodnota nebo úryvek ze zdrojového kódu.
- **Nečíslovaná kapitola** – Název kapitoly, která slouží k rozčlenění a strukturalizaci rozsáhlejších textů.
- *Popisek obrázku* – Popisek přibližující situaci zachycenou na obrázku.

Pro označení úseků textu převzatých z bakalářské práce [1] stejného autora, jako tato diplomová práce, používáme postranní čáru vpravo, která je na ukázku použita i u tohoto odstavce.

# 2 Analýza

V následujících kapitolách se zaměříme na analýzu problémů, které musíme vyřešit před implementací editoru. Analýzu začneme zamyšlením nad vlastnostmi schématu kompozice, neboť se jedná o základní koncept, se kterým bude náš editor pracovat.

Nad definovaným schématem kompozice budeme chtít provádět editace, které uživateli ulehčí vývoj komponentových aplikací. Budeme proto zjišťovat, které editace je vhodné uživateli nabídnout.

Aby mohl editor takové editace poskytnout, bude muset provádět analýzu různých zdrojových kódů a instrukcí. Z tohoto důvodu podrobně rozebereme možnosti analýzy konfigurace REA, která je z hlediska analýzy nejkomplikovanější. Ze zjištěných možností budeme vybírat takový způsob analýzy, který bude nejvhodnější pro potřeby našeho editoru. U vybraného způsobu si také podrobně rozebereme možnosti jeho implementace.

Nedílnou součástí editoru bude vykreslování schématu kompozice uživateli. Navrhneme proto vhodný způsob zobrazení. Nakonec si rozmyslíme vhodnou architekturu našeho editoru, po které budeme požadovat snadnou testovatelnost.

## 2.1 Schéma kompozice

Schéma kompozice by mělo přehledně vypovídat o vztazích mezi importy a exporty jednotlivých komponent. Musíme tedy nejprve zjistit, jaké se mohou mezi komponentami objevit závislosti. Podle nich pak budeme schopni formulovat přesné požadavky na formu schématu kompozice.

### Závislosti komponent

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.

Jak jsme psali již v úvodu, je výhodnější omezit závislost pouze na funkčnost, bez vazby na implementaci. Toho můžeme v prostředí MEF dosáhnout tak, že pro každou komponentu definujeme `interface`, který bude zpřístupněn. Ten poté využijeme pro kontrakt v definici importu/exportu. Jelikož `interface` žádnou vazbu na implementaci nemá, budou i komponenty s takovými kontrakty implementačně nezávislé.

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.

### Požadavky na schéma kompozice

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.

## 2.2 Editace schématu kompozice

Schéma popsané v předchozí kapitole přehledně zobrazuje komponentovou architekturu aplikace. Položme si však otázku, jaké editace by uživateli pomohly při práci se schématem kompozice. V kapitole 1.4 jsme zjistili, že základem kompozice je vytváření různých druhů katalogů a jejich interakce s kompozičním kontejnerem. Uživatel tedy potřebuje do schématu vkládat různé druhy MEF katalogů a měnit jim vlastnosti specifické pro konkrétní katalog.

Příkladem může být `DirectoryCatalog`, u nějž uživatel jistě uvítá možnost měnit cestu pro vyhledávání knihoven, aniž by musel ručně přepisovat zdrojový kód. Další potřebnou operací je změna vztahů mezi komponentami, které docílíme přesunováním komponent mezi kontejnery. Pro snazší orientaci ve zdrojových kódech bude také užitečné, aby se mohl uživatel podívat na místo vzniku editovaných objektů.

### Provádění editací

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. Klíčovou 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.

### Editace ve zkompilovaných assembly

Nová verze editoru bude navíc umět lépe spolupracovat s již zkompilovanými assembly. Nabízí se tedy otázka, zda dovolit provádět editace i v nich. Určitě by bylo zajímavé mít možnost změnit například cestu, odkud bude již zkompilovaná aplikace nahrávat svá rozšíření. Na druhou stranu, mohou tyto změny vést k nepředvídatelným následkům, které nelze dopředu odhadnout, kvůli nemožnosti výsledek editace uživatelsky zkontrolovat, tak jak je to možné ve zdrojovém kódu.

Druhou překážkou v editování zkompilovaných assembly je mechanismus podepisování, který chrání assembly před tím, aby mohla být modifikována.

Takovéto assembly potom není možné z principu editovat vůbec. Editor tedy v rámci *doporučených rozšíření* nebude nabízet editace ani v nepodepsaných zkompilovaných assembly, nicméně bude snadné takové editace do editoru přidat.

## 2.3 Rozšiřování existující aplikace (REA)

V úvodní kapitole 1.3 jsme popsali základní vlastnosti 4. vývojové konfigurace, kterou nazýváme REA. Tato konfigurace je z pohledu analýzy kompozice tou nejsložitější. První problém při analýze vyplývá z našeho pozorování, že rozšiřovaná aplikace nemusí být z vyvíjeného pluginu referencovaná. Aby mohl editor aplikaci analyzovat, musí nejdříve znát její umístění. To ale nedokáže bez chybějící reference zjistit. Z tohoto důvodu musí editor umožnit uživateli určit umístění rozšiřované aplikace, která bude definovat schéma kompozice svým kompozičním algoritmem.

### Vyhledávání assembly s komponentami

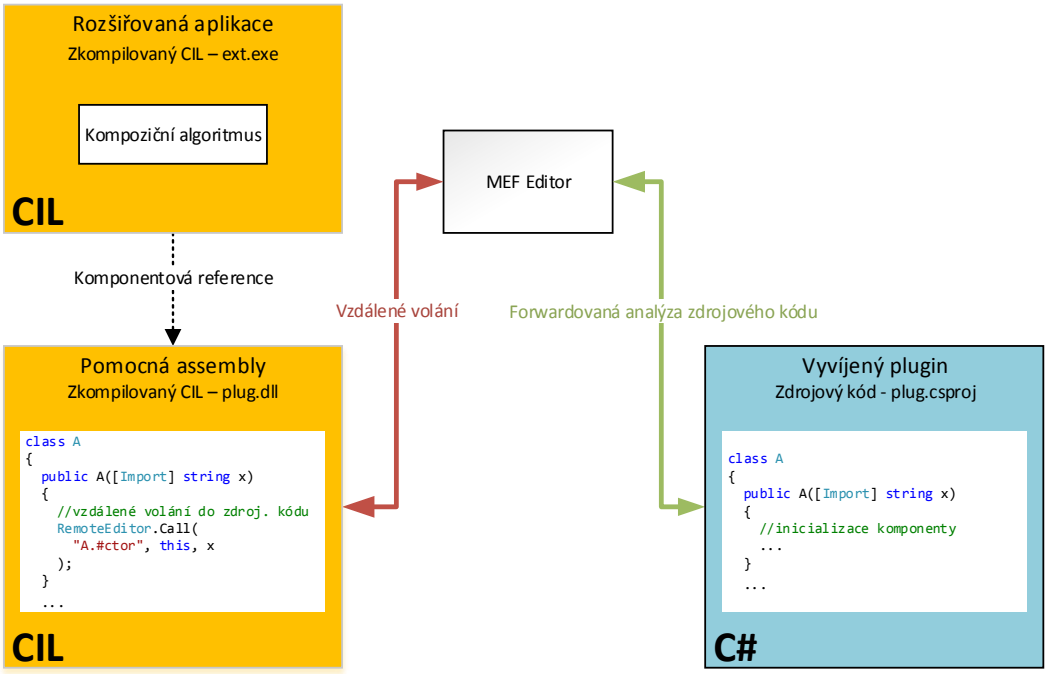
Chybějící reference není však v této konfiguraci jediným problémem. Obvyklý způsob, kterým jsou pluginy v konfiguraci REA do rozšiřované aplikace nahrávány využívá `DirectoryCatalog`. Ten dostane jako argument složku, ve které vyhledá dostupné assembly a z nich jsou pomocí MEF načteny komponenty pluginu. Tuto složku budeme označovat jako složku rozšíření.

Z toho ale vyplývá potřeba umístění assembly s pluginem ve složce rozšíření, kde ji rozšiřovaná aplikace očekává. Nicméně assembly s implementací pluginu nemusí být v době jeho vývoje vůbec dostupná, neboť zdrojové kódy pluginu mohou být nekompletní a proto nekompilovatelné. Z tohoto důvodu musí editor zajistit mechanismus jak nahrát komponenty definované ve zdrojových kódech vyvíjeného pluginu do rozšiřující aplikace.

### 1. možné řešení

Jedno možné řešení je založeno na faktu, že rozšiřovaná aplikace může přes `DirectoryCatalog` přistupovat pouze ke komponentám. Proto jsou z hlediska kompozice ostatní implementované třídy v assembly nepodstatné. Chtěli bychom tedy vytvořit pomocnou assembly, která bude obsahovat pouze třídy odpovídající komponentám pluginu. Bohužel ani třídy, které komponentám odpovídají, nemusí být ve fázi vývoje pluginu kompilovatelné.

Proto do pomocné assembly vložíme pro každou komponentu pouze odpovídající *kostru* komponenty. *Kostrou* zde máme na mysli třídu, která má s komponentou shodné rozhraní, ale obsahuje jinou implementaci metod. Tato implementace musí být uzpůsobena pro potřeby editoru tak, aby dokázal zjistit průběh kompozice, při kterém jsou volány konstruktory, gettery a settery komponent. Toho dosáhneme pomocí vzdálených volání, která dokážou využívat stejných postupů analýzy jako ve zdrojových kódech. V následujícím obrázku 2-1 je znázorněn princip vzdálených volání.



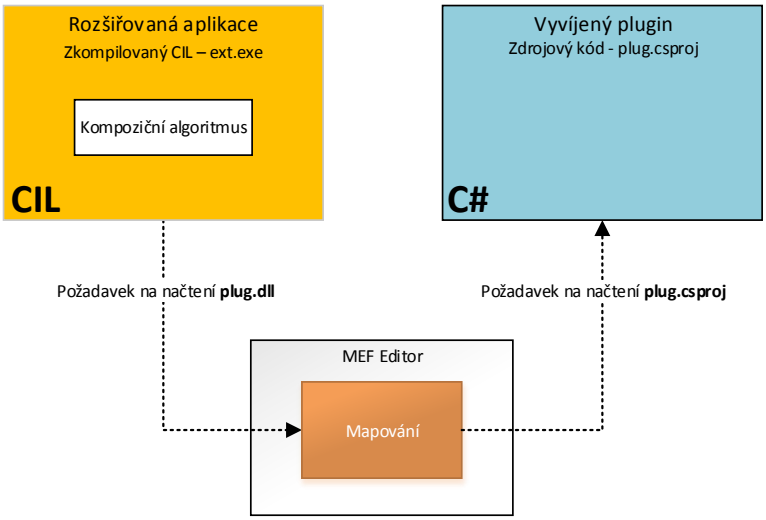

*2-1 Znázornění vzdáleného volání z konstruktoru v pomocné assembly přenesené přes editor na analýzu zdrojových kódů.*

Jak můžeme z tohoto obrázku vidět, vytváření pomocných assembly dokáže rozšiřované aplikaci poskytnout iluzi toho, že má k dispozici assembly se zkompilovaným pluginem. Přesto zde stále přetrvává problém se zjištěním, kde se nachází složka rozšíření, do které bychom mohli pomocnou assembly umístit. Navíc, implementace vzdálených volání by byla poměrně náročná. Z těchto důvodů by uvedené řešení bylo nevýhodné.

### 2. možné řešení

Jiným přístupem, jak by mohl editor poskytnout iluzi zkompilované assembly, je zjišťování pokusů rozšiřované aplikace o načtení assembly a jejich mapování na rozpracovanou assembly ve zdrojových kódech. Tím se také vyřeší problém s neznámým umístěním rozšiřujících assembly.

Tento přístup si ukážeme na následujícím obrázku 2-2. Rozšiřovanou aplikaci sledujeme tak, abychom dokázali zjistit, které assembly se pokouší načíst. Díky tomu dokážeme vytvořit mapování, které umožní nalezení odpovídající assembly i v případě, že je dostupná pouze v nezkompilované podobě.




*2-2 Ukázka principu mapování na základě sledování požadavků o načtení assembly.*

### Shrnutí

Z obrázku 2-2 vidíme, že sledování dotazů na načtení assembly z rozšiřované aplikace nám umožňuje vytvořit mapování i na assembly ve zdrojových kódech. Na druhou stranu editor musí být schopen zařídit spolupráci mezi zkompilovanou aplikací a assembly rozpracovanou ve zdrojových kódech, což může být vyřešeno interpretací, jak bude popsáno v následující kapitole. Z těchto důvodů bude pro implementaci editoru výhodnější 2. řešení.

## 2.4 Analýza kompozice

Základním úkolem našeho editoru je přehledné zobrazení schématu kompozice definovaném v kapitole 2.1 a nabízení jeho editací. Aby toho byl editor schopen, musí nejdříve získat informace o kompozici aplikace. Tyto informace 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.

### Interpretace celé aplikace

Alternativou, která odstraní problém se spouštěním, je přímé interpretování zdroje bez nutnosti jeho kompilace. Díky tomu budeme schopni vyřešit mapování assembly pro analýzu REA popisovanou v kapitole 2.3, neboť v průběhu interpretace můžeme simulovat libovolné běhové prostředí, ať už interpretujeme zdrojové kódy nebo instrukce CIL.

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, která by byla nutná pro spuštění aplikace, ale stále platí, že aplikace nemusí být dostatečně dopracovaná

na to, aby bylo možné ze startovní metody rozpoznat schéma kompozice. Navíc interpretace ze zdrojových kódů bývá řádově pomalejší než spuštění zkompilovaného programu. Interpretaci složitějších aplikací by proto nebylo možné zvládnout v rozumném čase.

### Interpretace composition pointu

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 pointů* 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.

### Hledání composition pointu

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 k nějaké kompozici dochází. 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);
    }
}
```

*2-3 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 pointy*. 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.

Dalším způsobem je označení *composition pointu* pomocí atributu. Ten může navíc obsahovat argumenty použité pro parametry vstupní metody interpretace. Toto řešení je výhodné i vzhledem k tomu, že využívá způsob zápisu, na který jsou uživatelé MEF zvyklí. Navíc jsou atributy zachované v aplikaci i po jejím zkompilování. To přispívá k lepší použitelnosti editoru na zkompilovaných assembly, kdy editor bude moct nabídnout stejný seznam *composition pointů* jako když má k dispozici zdrojové kódy.

### Implicitní composition point

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 pointů* tedy omezíme nutnost uživatelských zásahů do zdrojového kódu na minimum.

### Spouštění composition pointu

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 pointu* 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 pointu* tedy zavoláme statický konstruktor této třídy.

### Shrnutí

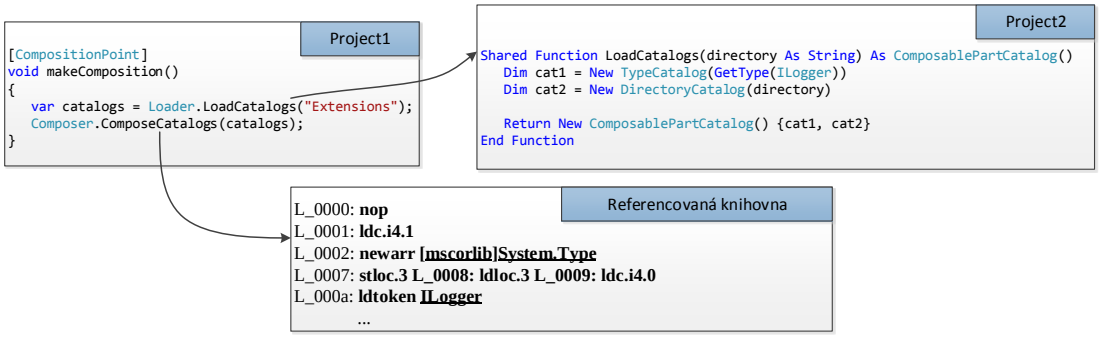
V této kapitole jsme zjistili, že vhodným nástrojem pro analýzu rozpracovaných projektů je jejich interpretace. Nebudeme však interpretovat celou aplikaci, ale pouze metody, které se týkají kompozice. Co všechno taková interpretace vyžaduje, budeme zjišťovat v následující kapitole.

## 2.5 Interpretace metod

Z předchozích kapitol víme, že editor potřebuje získávat informace o kompozici prostřednictvím interpretace *composition pointů*. Z kapitoly 2.4 víme, jak se dá *composition point* v aplikaci nalézt. Nyní se zamyslíme nad způsobem, jak provádět analýzu takového *composition pointu*.

Aby editor dokázal zadaný *composition point* analyzovat, musí nejdříve rozumět významu instrukcí zdroje, ve kterém je dostupný. To mohou být jednak zdrojové kódy v rozpracovaném projektu nebo zkompilované instrukce z referencovaných knihoven či z rozšiřované aplikace v případě konfigurace REA. Navíc typicky nestačí podpora jednoho jazyka zdrojového kódu nebo instrukcí, neboť je běžné že z *composition pointu* jsou volané další metody, které mohou být definované v rozdílných jazycích.

Popsanou situaci nastiňuje následující obrázek 2-4:




*2-4 Z metody makeComposition je volána metoda LoadCatalogs v Project2 psaná ve Visual Basic a ComposeCatalogs ze zkompilované knihovny, dostupné pouze v MSIL instrukcích.*

V předchozí verzi editoru bylo nutné pro jazyky, které si nejsou podobné, implementovat samostatný interpret. Každý takový interpret musel kromě vlastního interpretování řešit rutiny nezbytné pro analýzu. Při podpoře většího množství jazyků je však toto uspořádání nevýhodné. Proto budeme v nové verzi používat abstrakci virtuálního stroje, který bude zpracovávat jednu sadu instrukcí, schopnou interpretace všech .NET konstrukcí. Těmto instrukcím budeme říkat *analyzační instrukce*, jejichž analýza bude tranzitivně zajišťovat i analýzu instrukcí na ně převedených. Instrukce budeme převádět například z instrukcí zkompilovaných assembly nebo ze zdrojových kódů. Souhrnně budeme zdroj pro překlad do *analyzačních instrukcí* nazývat *zdrojové instrukce*.

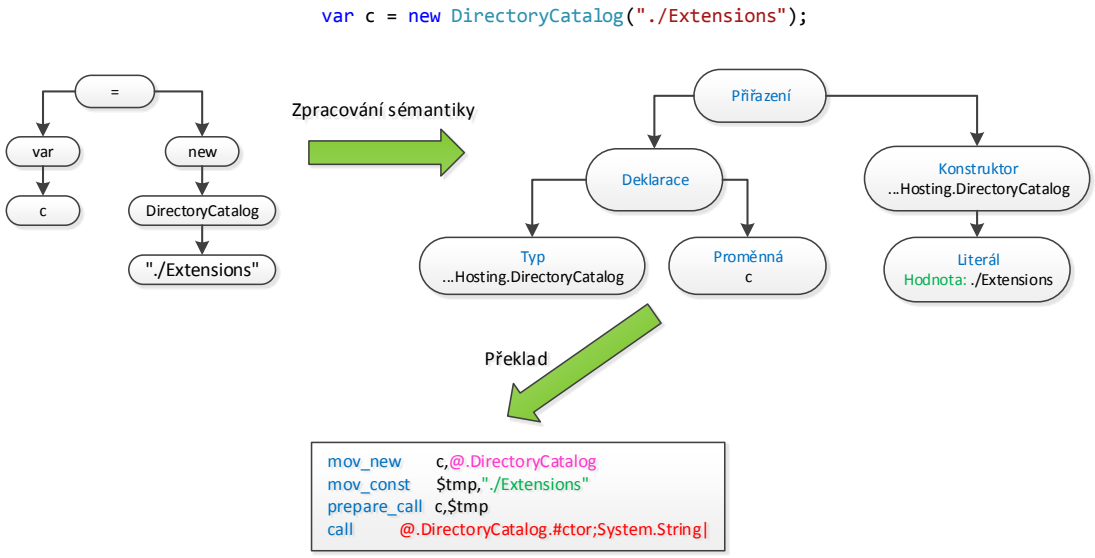
### Převod na jednotné instrukce

Implementace převodu jazyka na jeden společný bude snazší než implementace kompletního interpretu. Hlavním důvodem je fakt, že pro převod instrukcí do jiného jazyka typicky není nutné implementovat způsoby reprezentace interpretačního prostředí, zpracování jednotlivých instrukcí ani analyzační rutiny, tak jak by to bylo nutné při implementaci interpretu

Mohlo by se zdát výhodné jako *analyzační instrukce* použít instrukce některého existujícího jazyka. Přirozenou volbou by mohly být instrukce jazyka CIL, neboť ty podporují všechny konstrukty, které .NET nabízí. Nicméně v kapitole 2.7 si ukážeme, že kvůli stovkám instrukcí, které CIL obsahuje a kvůli architektuře jeho virtuálního stroje se jako *analyzační instrukce* nehodí. Stejně tak je komplikované provádět analýzu nad jazyky vyšší úrovně jako je například C#.

Z tohoto důvodu vyvineme speciální instrukční sadu určenou pro potřeby analýzy v rozsahu, který vyžaduje implementace editoru. Interpretace zdrojových kódů pak bude začínat jejich zpracováním z hlediska syntaxe, kdy získáme jednotlivá slova příkazů. V další fázi určíme syntaktický význam těchto slov, kdy rozhodujeme, která slova označují deklarace, přiřazení proměnných, apod. Tímto procesem získáme syntaktický strom. Ten je následně nutné převést do *analyzačních instrukcí*.

Příklad popsaného procesu je zobrazen na následujícím obrázku 2-5:




*2-5 Ukázka syntaktického (vlevo) a sémantického (vpravo) stromu. Uvedené stromy vznikly parsováním řádku zdrojového kódu (nahoře). Přeložením pak dostáváme analyzační instrukce (dole).*

## Interpretace analyzačních instrukcí metody

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

Jakmile máme k dispozici *analyzační instrukce* metody, můžeme je zpracovat pomocí virtuálního stroje. Ten bude vykonávat *analyzační instrukce* tak, aby co nejvíce simuloval běh .NET aplikace. Díky tomu dostaneme potřebné informace pro vykreslení schématu kompozice a jeho následnou editaci.

## Reprezentace interpretovaných objektů

Samotné vykonávání instrukcí by však nebylo možné, pokud bychom nemohli nějakým způsobem reprezentovat objekty, kterých se týkají. Na první pohled vypadá nejsnazší vytváření stejných .NET objektů, které by se objevily při skutečném spuštění simulovaného kódu. Na to bychom však potřebovali .NET reprezentaci všech typů, se kterými bychom pracovali. Tuto reprezentaci ale obvykle mít nebudeme, neboť budeme pracovat s nezkompilovanými zdrojovými kódy.

Řešením by se mohlo zdát vytváření .NET typů ze zdrojových kódů pomocí *Reflection* [23]. Zde však také narazíme na jistá omezení. Vytvořené typy už bychom nemohli snadno měnit ani odstraňovat, neboť *Reflection* nepodporuje změny na typech nahraných v aplikační doméně. Pro nás to však bude důležitá operace, neboť odpovídá modifikaci třídy ve zdrojovém kódu.

Z těchto důvodů musíme použít vlastní typový systém popsaný v kapitole 2.8. Ten však bude potřebovat speciální reprezentaci objektů, kterým budeme říkat *instance*. Po *instancích* požadujeme, jako po obyčejných objektech, schopnost uchovávat data, abychom byly schopni simulovat datové položky reprezentovaných objektů. Do *instancí* můžeme na rozdíl od nativních objektů ukládat data sesbíraná v průběhu analýzy, která využijeme pro tvorbu editací popsanou v následující

kapitole 2.6. Podle informací, které budeme pro editace potřebovat, pak v kapitole 2.7 definujeme požadavky na *analyzační instrukce*.

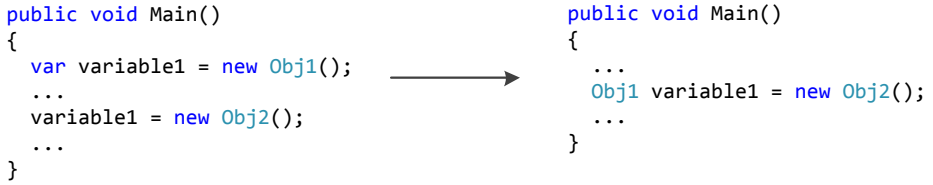
## 2.6 Vytváření editací

Důležitou vlastností editoru bude nabízení editací nad zobrazeným schématem kompozice. V kapitole 2.2 jsme rozmýšleli, které editace by bylo vhodné uživateli nabídnout. Také jsme zjistili, že pro editor bude důležitá podpora editací pouze ve zdrojových kódech. Proto v této kapitole rozebereme podrobně pouze případy týkající se úprav zdrojových kódů. Přesto však uvedené úpravy budeme navrhovat nezávisle na jazyku, aby bylo možné editor rozšířit o podporu editací dalších *zdrojových instrukcí*.

Analýzu vytváření editací začneme uvedením několik modelových editací, o kterých jsme v kapitole 2.2 zjistili, že uživateli pomohou při vývoji komponentové aplikace:

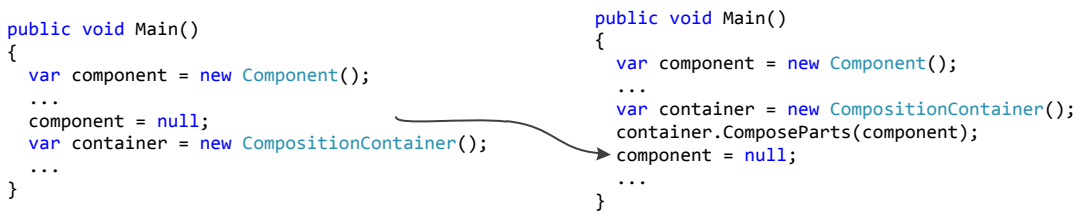
- **Přesunutí katalogu A z katalogu B do katalogu C** – Stačí smazat volání funkce, které přiřazuje A do B a vytvořit volání, které přiřadí A do C.
- **Změna cesty pro `DirectoryCatalog`** – Cesta musí být určena v konstruktoru `DirectoryCatalog`, proto stačí změnit argument na požadovanou cestu.
- **Přidání komponenty do `CompositionContainer`** – Pokud do kontejneru nebyla ještě žádná komponenta 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 2-6:




*2-6 Ukázka případu, kdy je nutné po odstranění objektu Obj1 redeklarovat proměnnou variable1. Za povšimnutí také stojí fakt, že nemůžeme redeklarování 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:




*2-7 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();
}
```

*2-8 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.

Ze zkušeností s používáním předchozí verze editoru vyplynulo, že ne vždy je pro uživatele srozumitelný důvod proč není možné editaci provést. Stejně tak by použití zpříjemnilo lepší informování o editacích, které se editor chystá provést po uvolnění tlačítka myši. Z tohoto důvodu bude nová verze obsahovat kontextové nápovědy, okamžitě informující uživatele o významu a případných problémech s editací.

### Dopad rozšiřitelnosti editoru na editace

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 rozšíření zodpovědné za zpracování této metody postará o provedení požadovaných 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.

Z předcházející analýzy můžeme vidět, že pro podporu editací stačí pouze několik druhů změn ve zdrojových instrukcích:

- Přidání/odebrání volání metody
- Přidání/přepsání/odebrání argumentu
- Konstrukce objektu
- Smazání přiřazení do proměnné
- Redeklarace proměnné
- Změna pořadí řádků

V průběhu analýzy *composition pointu* bude proto nutné, zjistit z *analyzačních instrukcí* informace, které nám umožní tyto editace provádět.

## 2.7 Analyzační instrukce

Z předchozích kapitol již víme, že editor potřebuje analyzovat různé druhy *zdrojových instrukcí*. Ze zkušeností z předchozí verze také víme, že je vhodné *zdrojové instrukce* převést na společné *analyzační instrukce*. Tím se vyhneme nutnosti analyzovat každý jazyk *zdrojových instrukcí* zvlášť. Z uvedených důvodů se v této kapitole zamyslíme nad *analyzačními instrukcemi*, které by byly pro náš editor vhodné.

Abychom mohli navrhnout vhodné instrukce pro náš editor, musíme nejprve zjistit, které konstrukce budeme muset analyzovat. Ze způsobu použití MEF víme, že struktura katalogů je vytvářena pomocí volání metod jako je například `AggregateCatalog.Catalogs.Add`, kterou jsme si ukázali na příkladu kompozice v kapitole 1.4. I vytváření samotných katalogů můžeme chápat jako volání speciální metody – konstruktoru. Proto potřebujeme, aby byly a*nalyzační instrukce* schopné reprezentovat různá volání metod.

### Volání metod

Zamysleme se ale, jakým způsobem bychom volání metod prováděli. Pokud bychom simulovali jejich chování přesně podle .NET, docházelo by k problémům s nežádoucími efekty, jako je například přístup k souborovému systému. To by však bylo z hlediska bezpečnosti použití editoru nežádoucí. Mohlo by se stát, že by analýza editorem například smazala uživatelské soubory. Z těchto důvodů musí existovat možnost, jak určit, která volání budou přesně simulovat běh .NET, a která volání bude editor nahrazovat rutinami podle uživatelských rozšíření.

V předchozí kapitole 2.6 jsme zjistili, které druhy změn ve *zdrojových instrukcích* musí editor podporovat. Některé z těchto změn se týkají volání metod a jejich argumentů. Abychom mohli takové editace v editoru nabízet, potřebujeme do *analyzačních instrukcí* uložit informace, které umožní provést editaci smazání, přepsání či přidání argumentu, případně smazání celého volání.

Další ze změn *zdrojových instrukcí*, které editor potřebuje, je vytváření volání metod a vytváření nových objektů. Zde však nastává problém, kam uložit informace pro vytvoření nějakého konstruktu. Parser *zdrojových instrukcí* nemůže dopředu odhadnout, zda a kde bude nutné vytvářet například volání, protože umístění vytvářeného volání je závislé na uživatelských akcích v editoru. Přidání komponenty do jednoho kontejneru může totiž vytvořit volání jinde, než přidání komponenty do jiného kontejneru. Parser proto musí mít možnost v *analyzačních instrukcích* označit všechna místa, kde lze vytvářet nová volání a objekty.

### Práce s proměnnými

Volání metod a předávání argumentů je v obvyklých programovacích jazycích řešené za pomoci proměnných. Jejich použití však slouží převážně pro účely zpřehlednění zápisu kódu, proto by jejich použití pro potřeby interpretace nebylo nutné. Z hlediska editací ale hrají proměnné důležitou roli. Abychom například mohli vytvořit volání metody na objektu ve zdrojovém kódu v C#, musíme znát název proměnné, ve které je tento objekt přiřazen.

Z těchto důvodů proměnné do *analyzačních instrukcí* zahrneme. Tato volba také ovlivní architekturu virtuálního stroje, který bude analyzační instrukce zpracovávat.

### Architektura virtuálního stroje

Běžně se používají dva druhy virtuálních strojů. Prvním typem jsou zásobníkové virtuální stroje, jejichž typickým zástupcem může být například virtuální stroj pro CIL. V této architektuře je ale místo proměnných používán pro předávání hodnot zásobník.

Proto architektura našeho virtuálního stroje bude vycházet z druhého typu, kterým jsou registrové virtuální stroje. Ty se abstrakcí podobají architektuře běžných procesorů, které si hodnoty předávají pomocí omezeného počtu registrů. V našem případě budeme za registry považovat právě proměnné a z praktických důvodů nebudeme jejich počet explicitně omezovat. To nám umožní do proměnných a instrukcí, které s nimi budou pracovat, uložit informace pro editace týkající se přiřazování do proměnných, jejich mazání a redeklarace.

Poslední typ změn, který budeme potřebovat, je přeházení *zdrojových instrukcí*. V předchozí kapitole jsme zjistili, že tato změna je nutná, abychom dokázali zajistit přítomnost konkrétní *instance* v proměnné, kterou chceme předat jako argument v místě volání. Mohlo by se zdát, že přeházení *zdrojových instrukcí* bude možné provádět v závislosti na prostém přeházení *analyzačních instrukcí*.

Nicméně jedné *zdrojové instrukci*, kterou můžeme chápat například jako jeden řádek v jazyce C#, bude typicky odpovídat více *analyzačních instrukcí*. Příkladem může být přiřazení návratové hodnoty do proměnné. Jak zjistíme v příští kapitole, bude tato operace reprezentována dvěma instrukcemi *analyzačních instrukcí*. Přehozením těchto instrukcí by však došlo ke změně *zdrojové instrukce*, nikoliv k jejímu přehození s jinou *zdrojovou instrukcí*. Z tohoto důvodu je nutné umožnit parserům sdružovat *analyzační instrukce* do bloků, které budou odpovídat *zdrojovým instrukcím*, a proto je bude možné vzájemně přesunovat.

### Podmínkové příkazy

Nyní máme definované požadavky, které od *analyzačních instrukcí* vyžadujeme kvůli možnosti provádět editace. Abychom ale byly schopni simulovat běh programu, musíme také umět interpretovat příkazy pro řízení toku programu. Takovými příkazy jsou například `if`, `while` nebo `goto` z jazyka C#.

### Jazyk analyzačních instrukcí

Pro *analyzační instrukce* bychom rádi využili nějaký již existující jazyk, přirozenou volbou by byl jazyk CIL, neboť dokáže pokrýt veškeré konstrukty .NET. Už ale víme, že architektura jeho virtuálního stroje není vhodná pro práci s proměnnými, kterou budeme požadovat pro editace. Tímto problémem netrpí vyšší jazyky jako je třeba C#. Jejich nevýhodou ale je textová reprezentace, která není pro strojové zpracování příliš efektivní, proto by ji bylo pro potřeby interpretování stejně nutné převést na jinou reprezentaci. Z těchto důvodu pro nás bude výhodnější vytvořit vlastní jazyk, který navrhneme přímo pro potřeby editoru.

Významným kritériem pro nás bude malý počet instrukcí takového jazyka, protože při analýze je bude muset editor po jedné procházet a zjišťovat jejich význam. Čím menší počet instrukcí bude jazyk mít, tím snazší pak bude vlastní analýza i tvorba editací. Z tohoto důvodu také nezahrneme typovost do sémantiky analyzačního jazyka, ale přenecháme řešení typů vyšší vrstvě – typovému systému. Díky tomu bude sémantika instrukční sady jednodušší a také bude obsahovat méně instrukcí, neboť nemusíme například řešit instrukce pro nahrání nebo kontrolu typu. Jak uvidíme v průběhu návrhu instrukcí, nebude mít takové omezení sémantiky vliv na schopnost interpretovat potřebné .NET konstrukce.

## 2.7.1 Návrh analyzačních instrukcí

Nejsložitějším konceptem, se kterým se bude muset instrukční sada vypořádat je volání metod. V .NET existuje několik druhů takových volání, jejichž seznam je uveden zde:

- **virtuální volání** – označuje volání, při kterých zjistíme konkrétní volanou metodu až za běhu programu, na základě typu volaného objektu,
- **nevirtuální volání** – běžné volání, u kterého zjistíme volanou metodu podle typu, který je znám již při překladu,
- **statická volání** – volání, která sdílejí jeden objekt pro každou statickou třídu, vytvořený až těsně před prvním použitím této třídy.

### Instrukce volání metod

Pro reprezentaci volání v *analyzačních instrukcích* budeme používat instrukci *call*. Tato instrukce potřebuje operand, dle kterého budeme schopni volanou metodu nalézt, a také seznam proměnných, ve kterých se nachází argumenty volání. Zamysleme se však, jak by měl identifikátor metody vypadat. Ideální by bylo, kdybychom podle něj dokázali jednoznačně určit volanou metodu.

Nicméně to není možné v případech reprezentace virtuálních volání, jak je ukázáno na následujícím obrázku 2-9:

```
class A {
  //virtuální metoda, která
  //může být přepsána v potomkovi
  public virtual void M1()

  //metoda, která virtuální není
  public void M2()
}
```

```
class B : A {

  //přepíšeme metodu A.M1
  public override void M1()

  //tato metoda NEpřepíše A.M2
  public void M2()
}
```

```
void Test(A obj)
{
  //volaná metoda je určena
  //na základě typu obj
  //v našem případě buď
  //A.M1 nebo B.M1
  obj.M1();
}
public void Usage()
{
  //způsobí vyvolání A.M1
  Test(new A());
  //způsobí vyvolání B.M1
  Test(new B());
}
```

*2-9 Ukázka vlastností virtuálních volání v jazyce C#.*

Všimněme si metody `Usage`, která volá metodu `Test` s různým typem objektů. Kvůli tomu dochází postupně k vyvolání metod `A.M1` a `B.M1`.

Rozlišování metod podle volaného objektu však nelze provádět vždy. Kdybychom v metodě Test použili `obj.M2`, dostali bychom v metodě `Usage` sekvenci volání `A.M2`, `A.M2`, neboť metoda `M2` není virtuální.

Z tohoto důvodu musíme mít možnost identifikátor označit příznakem, zda je nutné metodu vyhledávat na základě jejích argumentů až v průběhu interpretování. Poznamenejme, že objekt, na němž je metoda volána, je pro instrukci *call* také argumentem. Díky tomu dokážeme simulovat virtuální i nevirtuální volání.

### Instrukce pro statická volání

Statická volání jsou prováděna na objektu sdíleném mezi všemi statickými voláními na stejné třídě. Inicializace těchto objektů je prováděna statickým konstruktorem pouze jednou a to těsně před prvním voláním statické metody na třídě. Abychom mohli simulovat i tento druh volání, musíme být schopni zjistit, zda byla *sdílená instance* odpovídající sdílenému objektu již inicializována. Tak zajistíme, že se inicializace neprovede vícekrát. Pro tyto účely bude virtuální stroj nabízet globální proměnné, které budou na rozdíl od běžných proměnných uchovávat *instance* i napříč voláním metod. Díky tomu můžeme zjistit, zda byla *sdílená instance* již vytvořena, nehledě na metodu kde k tomu došlo.

Vlastní vytvoření *sdílené instance* je nutné provést těsně před instrukcí *call*, která reprezentuje nějaké statické volání. Pro tyto účely přidáme do analyzačních instrukcí instrukci *ensure_init*, která má operand určující cílovou globální proměnnou a operand s identifikátorem metody, která provede inicializaci v případě,

že globální proměnná ještě inicializovaná není. Umístěním *ensure_init* před *call* se statickým voláním tedy můžeme simulovat statická volání tak, jak probíhají v .NET.

### Instrukce pro podporu uživatelských rozšíření

V předchozí kapitole jsme zjistili, že je důležité umožnit uživatelským rozšířením změnit chování vybraných metod od přesné simulace .NET. Protože je chování metody určené jejími instrukcemi, potřebujeme tyto instrukce nahradit podle definic v uživatelských rozšíření. Samotné nahrazení je triviální, protože nahrávání analyzačních instrukcí metod bude řídit naše implementace. Problémem ale je způsob jak definovat metody v rámci uživatelských rozšíření. Definice psaná přímo v analyzačních instrukcích by totiž byla pro uživatele vyvíjejícího rozšíření příliš náročná a nepřehledná. Proto dovolíme v rámci analyzačních instrukcí přímo vykonávat uživatelský kód implementovaný nativně v .NET. Spuštění kódu se bude provádět instrukcí *direct_invoke*, jejímž operandem je nativní .NET metoda.

Je evidentní, že editace kódu vykonávaného v rámci *direct_invoke* nebude editor schopen poskytovat, protože vykonávaný kód není zapsán v analyzačních instrukcích. Nicméně použití této instrukce analýze editoru nevadí, protože je určeno pro definice typů uživatelskými rozšířeními, které dokáží sami definovat své editace, pokud je to potřeba. Definice uživatelských rozšíření i tvorba editací bude podrobně vysvětlena v kapitole 6.

Nyní máme definované všechny potřebné instrukce pro reprezentaci volání, jak již ale víme, pouhá volání nám stačit nebudou. Kvůli editacím musíme reprezentovat i přiřazení do proměnných. Zamysleme se tedy, jak by měly *analyzační instrukce* pro přiřazení vypadat.

### Přiřazení v běžných jazycích

Každé přiřazení se musí skládat ze zdroje, který určuje přiřazovanou hodnotu, a z cílového umístění, které určuje, kam bude hodnota přiřazena. Ve vyšších jazycích jako je třeba C# můžeme mít několik cílů pro přiřazení. Takovým cílem může být proměnná, vlastnost nebo indexer, případně datová položka objektu. Různé typy přiřazení můžeme vidět na následujícím obrázku:

```
var variable = 1;
obj.Property = 2;
obj["index"] = 3;
obj.Field = 4;
```

*2-10 Ukázka rozdílných typů přiřazení v jazyce C#*

Všimněme si, že ačkoliv se přiřazení do vlastnosti nebo indexeru objektu zapisuje pomocí operátoru přiřazení, jedná se vlastně o jiný zápis volání metody odpovídající patřičnému setteru. Z tohoto důvodu se takové konstrukty v *analyzačních instrukcích* budou také reprezentovat jako volání. Proto *analyzační instrukce* pro přiřazení nemusí jako cíl přiřazení uvažovat vlastnosti ani indexery objektů.

Jiným případem je přiřazení do datové položky objektu. V tomto případě opravdu dochází k přiřazení hodnoty do paměti. Nicméně v předchozí kapitole jsme zdůvodnili, proč nechceme do *analyzačních instrukcí* vnášet objektovou sémantiku a typovost. Z tohoto důvodu budou přiřazení do datových položek řešena typovým systémem stejným způsobem jako by se jednalo o volání setteru, jak bude podrobněji vysvětleno v kapitole 2.8.

Posledním a jediným cílem, který nám pro přiřazení zbyl, je přiřazení do proměnné. Abychom mohli navrhnout potřebné instrukce pro přiřazení, musíme ale

ještě vědět, z jakých zdrojů budeme přiřazované hodnoty získávat. Ve vyšších jazycích může být zdrojů hodnot opět několik.

Přiřazovat hodnotu můžeme z proměnné, datové položky objektu, vlastnosti, indexeru, výsledku výrazu a obecně z návratové hodnoty volání, kterými většinu z těchto konstruktů reprezentujeme. Jediným z nich, který v *analyzačních instrukcích* není reprezentován voláním je proměnná. Dalším zdrojem hodnoty pro přiřazení může být literál a vytvoření nového objektu, které musíme v *analyzačních instrukcích* také reprezentovat.

### Instrukce přiřazení

Prvním typem přiřazení, které definujeme je přiřazení mezi proměnnými. Zapisovat ho budeme jako instrukci *mov*, jejíž první operand odpovídá cílové proměnné a druhým operandem je proměnná s přiřazovanou hodnotou.

Pro přiřazení literálu neboli konstantní hodnoty reprezentované ve zdrojovém kódu, zavedeme instrukci *mov_const*, jejíž první operand bude opět označovat cílovou proměnnou a druhým operandem bude *instance* odpovídající přiřazovanému literálu.

*Instanci*, nikoliv však konstantní, přiřazujeme také v případě reprezentace přiřazení nového objektu. To je ale sémanticky odlišné od přiřazení literálu, neboť nový objekt nemusí být na rozdíl od literálu konstantní a proto ho nelze sdílet a je nutné při každém přiřazení vytvořit objekt nový. Proto potřebujeme další instrukci, která při každém zpracování vytvoří a přiřadí novou *instanci*. Tato instrukce se bude nazývat *mov_new* a jejím operandem bude proměnná, do které bude přiřazena nově vytvořená *instance*.

Dalším druhem přiřazení, které potřebujeme, je přiřazení návratové hodnoty z volání do proměnné. Pro tento účel definujeme instrukci *mov_return* s jediným operandem, který udává cílovou proměnnou. Tato instrukce do cílové proměnné přiřadí návratovou hodnotu, vrácenou z metody volané nejbližší předcházející instrukcí *call*.

Těmito instrukcemi máme v rámci *analyzačních instrukcí* pokryté všechny případy přiřazení, se kterými se ve vyšších jazycích setkáváme. Nicméně pro potřeby *analyzačních instrukcí* musíme ještě vyřešit přiřazení hodnot z argumentů metody. V běžných jazycích toto přiřazení probíhá „automaticky“ pojmenováním podle parametrů metody.

Nicméně v jazycích nižší úrovně, jako je například CIL, se k argumentům nepřistupuje podle jména, ale podle pořadového čísla argumentu. Tento přístup je efektivnější, neboť není nutné udržovat seznam jmen parametrů. Z tohoto důvodu pro nás bude také výhodnější na úrovni *analyzačních instrukcí* neuvažovat pojmenování parametrů metod, ale raději definujeme instrukci, která nám do proměnné přiřadí hodnotu argumentu dle pořadového čísla. Tuto instrukci budeme označovat jako *mov_arg*, kde prvním operandem bude opět cílová proměnná a druhý operand bude označovat index argumentu s přiřazovanou hodnotou.

### Instrukce řízení interpretace

Poslední skupina instrukcí, kterou zbývá popsat, jsou instrukce pro řízení běhu interpretace. Tyto instrukce jsou nutné, abychom dokázali reprezentovat podmínky, cykly a příkazy skoku jako třeba `return`, `continue` a `goto` z vyšších jazyků.

Obecně můžeme příkazy pro řízení běhu programu rozdělit na dvě skupiny. První skupinu nazýváme nepodmíněné skoky. Pokaždé, když se běh C# programu dostane například na `return`, `continue` nebo `goto`, dojde ke skoku na předem

známou instrukci. Pro reprezentaci těchto příkazů přidáme do *analyzačních instrukcí* instrukci *jmp*, jejímž jediným operandem bude label, který označuje cílovou instrukci pro skok.

Nesmíme však zapomenout na trochu speciální příkaz – `return` s návratovou hodnotou. Zde je nutné kromě skoku také nastavit návratovou hodnotu volání. Definujeme proto instrukci *ret*, jejíž jediný operand je proměnná s návratovou hodnotou. Po nastavení návratové hodnoty dojde navíc ke skoku na konec metody, což zajistí stejné chování, na jaké jsme u `return` zvyklí.

Druhou skupinou jsou příkazy, u kterých dojde ke skoku pouze, pokud je splněna určitá podmínka. Těmto příkazům říkáme podmínkové a jedná se například o `if else` z C#. Podmínkové příkazy dokážeme řešit pomocí instrukce *jmp_if*, jejímž prvním operandem je label cílové instrukce pro skok a druhým operandem je proměnná s podmínkovou hodnotou. Pokud je tato hodnota vyhodnocena jako `true`, ke skoku dojde, jinak se pokračuje ve vykonávání programu bez skoku.

S takto definovanými analyzačními instrukcemi jsme schopni pokrýt veškeré konstrukty důležité z hlediska analýzy. Kvůli zjednodušení práce se skoky dáme ještě možnost překladačům využít operace *nop*, která nemá žádné operandy a neprovádí žádné akce. Nicméně při implementaci překladače může být s výhodou využita jako cílová instrukce pro skok.

Může se zdát, že přidání této „zbytečné“ instrukce porušuje požadavek na co nejmenší instrukční sadu. Tento požadavek ale vyplývá z potřeby malého množství typů instrukcí, které je nutné analyzovat. Operace *nop* však nemá žádnou sémantiku a může být proto v průběhu analýzy ignorována.

### Shrnutí definovaných instrukcí

| | |
|---|---|
| *call* `[MethodID], a, b, c…` | Volání metody určené podle `MethodID` s argumenty zadané seznamem proměnných `a, b, c…` |
| *ensure_init* `a, [InitializerID]` | Inicializuje nový objekt metodou zadanou dle `InitializerID` a uloží ho do globální proměnné `a`, pokud je tato proměnná neinicializovaná. |
| *direct_invoke* `[NativeCode]` | Vyvolá nativní .NET kód zadaný operandem. |
| *mov* `a, b` | Instrukce způsobí přiřazení *instance* v proměnné `b`, do proměnné `a.` |
| *mov_const* `a, [c]` | Přiřadí *instanci* reprezentující literál `c` do proměnné `a`. |
| *mov_new* `a` | Vytvoří novou *instanci* a přiřadí ji do proměnné `a`. |
| *mov_return* `a` | Přiřadí návratovou hodnotu z předcházejícího volání do proměnné `a`. |
| *mov_arg* `a, [index]` | Provede přiřazení argumentu na pozici zadané indexem do proměnné `a`. |
| *jmp* `[label]` | Nepodmíněný skok na zadaný `label`, který se provede vždy. Tuto instrukci můžeme využít například pro skok za podmíněný blok instrukcí. |
| *jmp_if* `[label], a` | Provede skok na zadaný `label` pouze, pokud je hodnota reprezentovaná proměnnou |

| | |
|---|---|
| | `a` vyhodnocena na `true`. |
| *ret* `a` | Ukončí volání funkce a nastaví návratovou hodnotu na *instanci* v proměnné `a`. Připomeňme, že tuto hodnotu je možné po volání funkce využít pomocí popsané instrukce *mov_return*. |
| *nop* | Neprovede žádnou akci. |

## 2.8 Typový systém

Navržené *analyzační instrukce* nemají z důvodu zachování jednoduché sémantiky instrukcí žádnou přímou podporu typů *instancí*. Typovost je však nezbytná pro korektní volání metod. Ta jsou v rámci virtuálního stroje řešena podle identifikátoru metody poskytnutého vyšší vrstvou. Abychom nemuseli řešit mechanismus tvorby identifikátoru metod a jejich dědičnost pro každý překládaný jazyk, potřebujeme vrstvu, která bude poskytovat abstrakci typů známou z .NET.

Jak už jsme nastínili, typy jsou důležité pro rozhodování o konkrétní implementaci metody volané na objektu. Z tohoto důvodu rozlišujeme několik druhů definic metod podle jejich vlastností.

- **Nevirtuální metoda** – Běžná metoda, která je definovaná na třídě. Implementaci metody je možné snadno určit z typu již za překladu.
- **Virtuální metoda** – Metoda, jejíž implementace je závislá na konkrétním typu objektu zjistitelného až za běhu podle dědičné hierarchie.
- **Abstraktní metoda** – Chová se stejným způsobem jako virtuální metoda, nicméně neurčuje defaultní implementaci.
- **Interface metoda** – Implementace je opět závislá na konkrétním typu objektu, nicméně nevyužívá dědičnou hierarchii jako virtuální metody.
- **Statická metoda** – Metoda, která se chová stejným způsobem jako nevirtuální metoda, avšak na objektu sdíleném pro všechna místa volání.
- **Konstruktor** – Další typ metody, jejíž implementaci můžeme obdržet stejným způsobem jako u nevirtuálních metod.

Každou metodu v typovém systému budeme označovat jednoznačným identifikátorem na základě jejího plného jména a typu parametrů. Stojí za povšimnutí, že taková identifikace nám umožní odkazovat se i na metody, které nemusí být zatím nikde definované.

Přes velké množství různých definic metod si vystačíme pouze se dvěma základními principy při hledání odpovídající implementace. Pro všechny druhy metod z uvedeného seznamu, které jsou známé již za překladu, nám pro nalezení implementace metody stačí pouze identifikátor získaný za překladu.

Identifikátor získaný u zbylých metod může označovat metodu definovanou bez implementace například v případě interface metody. Nebo virtuální metodu, která byla předefinována v nějakém potomkovi. Proto potřebujeme navíc procházení hierarchie typů volané *instance*, dokud nenarazíme na implementaci metody se stejným jménem a parametry jaké má metoda s volaným identifikátorem.

### Speciální metody

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* [17]. Tato reprezentace je popsána v tabulce přetížení operátorů [16]. 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 tomto principu funguje i .NET reprezentace konverzních operátorů, kterou můžeme vidět v CIL kódu přeložených aplikací. Obdobou jsou potom vlastnosti objektů, které se skládají z metody pro nastavení hodnoty a metody pro získání hodnoty. Pro vlastnost 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 překladačům budeme datové položky opět reprezentovat jako by se jednalo o vlastnost s getter a setter metodou. Indexery budeme převádět do metod `get_Item` a `set_Item` s příslušným počtem parametrů.

## 2.9 Vykreslování schématu kompozice

V průběhu interpretace *composition pointu*, 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.

### Zobrazování komponent

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

Definovali jsme způsob zobrazení schématu kompozice na základě výsledků získaných interpretací *composition pointu*. Aby schéma kompozice odpovídalo aktuální verzi zdrojových kódů, musíme ho překreslit při každé změně, kterou uživatel provede v metodách použitých při interpretaci. Tímto překreslením však získáme novou sadu *instancí*, která není nijak svázaná s těmi předchozími. Pokud tedy budeme chtít uživateli například umožnit pozicování objektů ve schématu kompozice, potřebujeme identifikovat *instance* napříč spuštěními *composition pointu*. Díky tomu můžeme zobrazit *instance* získané v různých spuštěních *composition pointu* na stejné pozici, pokud logicky odpovídají jednomu objektu.

### Identifikace instancí

Položme si nyní otázku, jakým způsobem zajistit vhodnou identifikaci *instancí*. 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, taková 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.

### Přehlednost schématu kompozice

Z připomenutí předchozí verze v kapitole 1.5 části vykreslování schématu kompozice víme, že původní pozicovací algoritmus mohl v některých případech zobrazovat schémata kompozice nepřehledným způsobem. Tato nepřehlednost byla převážně způsobena možností překrývat přes sebe několik *instancí* zobrazených ve stejném katalogu. Tím mohlo docházet ke „schování“ důležitých komponent.

Druhý typ nepřehledných situací se vyskytoval u kompozic s mnoha spojnicemi mezi komponentami. Protože spojnice procházely napříč komponentami, zhoršovaly čitelnost schématu kompozice.

V nové verzi editoru bychom chtěli těmto problémům předcházet, proto se podíváme na techniky, které nám umožní vykreslovat schéma kompozice přehledněji.

## 2.9.1 Pozicování zobrazených instancí

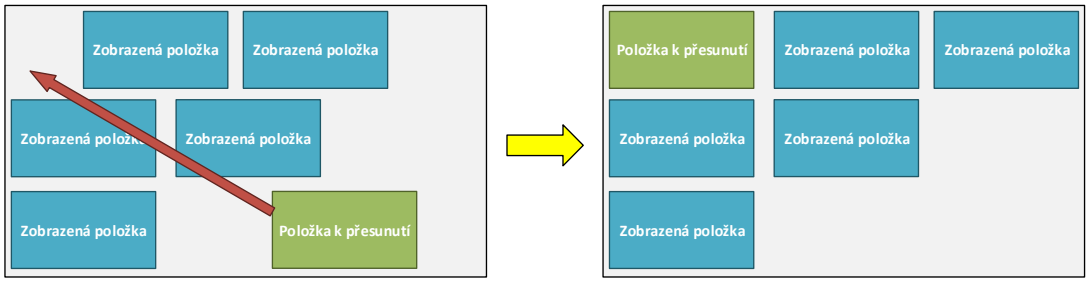
Již víme, že budeme ve schématu kompozice zobrazovat katalogy a v nich vnořené odpovídající komponenty. Komponenty se také budou moci zobrazovat samostatně, pokud nebudou zahrnuté v žádném katalogu. Protože dopředu nedokážeme odhadnout jaké rozložení komponent a katalogů bude uživateli vyhovovat, dovolíme mu pozice těchto položek upravovat.

S tím však přichází problém, kdy si uživatel může zakrýt nějakou zobrazovanou položku jinou položkou, která je na stejné úrovni. Přičemž stejnou úroveň mají dvě položky, pokud se obě nachází ve stejném katalogu nebo žádná z nich není zahrnuta v žádnému katalogu. Abychom tento problém vyřešili, budeme muset mezi položkami udržovat alespoň minimální nenulovou vzdálenost.

### Udržování minimální vzdálenosti

Jedno možné řešení, které by zajistilo korektní splnění podmínky nenulové vzdálenosti, spočívá v zabránění takových úprav, které by dostali položku příliš blízko jiné zobrazené položce. Toto řešení by však nebylo příliš uživatelsky přívětivé, neboť by se mohlo často stávat, že bude požadovaná úprava zamítnuta.

Jiným přístupem je snaha provést požadovanou úpravu vždy a následně upravit pozice ostatních položek tak, aby splňovaly požadovanou podmínku na vzdálenost. Toho můžeme docílit posunutím ostatních položek, které jsou příliš blízko položky, jejíž pozici uživatel určil. Uvědomme si ale, že posunutím položek může řetězově docházet k dalším posunům. Popsanou situaci znázorňuje následující obrázek 2-11:



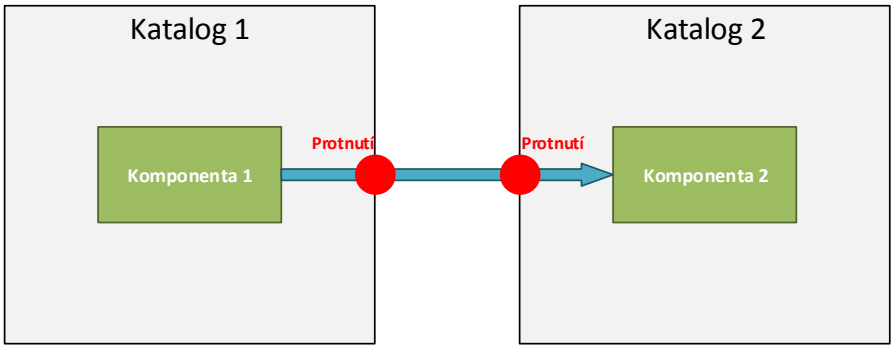

*2-11 Situace, kdy je po přesunutí zelené položky nutné řetězově posunout alespoň dvě modré položky, tak abychom dostali schéma bez překrývání.*

Další situaci, kterou je nutné ošetřit, představuje posunování položky, která je příliš blízko okraje katalogu, ve kterém je vnořena. V takových případech je nutné katalog patřičně zvětšit, aby se do něj zobrazované položky vešly.

### 2.9.2 Hledání tras spojnic komponent

Ze zkušeností s předchozí verze editoru víme, že protínání spojnic se zobrazovanými položkami může zhoršovat čitelnost schématu kompozice. Proto se pokusíme tomuto protínání předcházet. Musíme však podotknout, že existují situace, kdy k protnutí položek docházet musí. Modelovou situaci představují dvě spojené komponenty, přičemž každá z nich je v jiném katalogu. Nehledě na trasu, kterou spojnice povede, musí protnout oba dva katalogy.

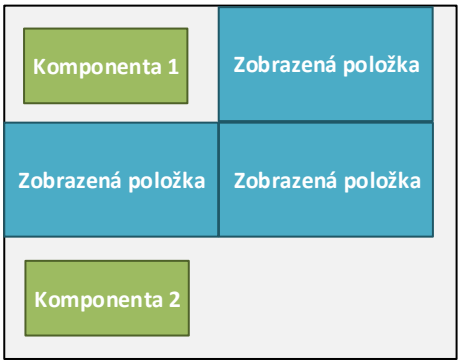
Tuto situaci znázorňuje následující obrázek:




*2-12 Situace, kdy spojnice musí protnout hranu obou katalogů, nehledě na trasu, kterou spojnice povede.*

Pokud však spojnice povede mezi dvěma komponentami, které jsou zobrazené na stejné úrovni, můžeme neprotínající trasu nalézt vždy. Existence trasy je totiž zajištěna garantovanou nenulovou minimální vzdáleností mezi položkami na stejné úrovni, kterou jsme rozebírali v minulé kapitole.

Na druhou stranu, pokud bychom tuto vlastnost zajištěnou neměli, mohli bychom se dostat do situací, kdy zobrazenou položku spojnicí překřížit musíme. Takovou situaci znázorňuje následující obrázek:




*2-13 Pokud dovolíme, aby byly zobrazené položky libovolně blízko u sebe, nemůžeme pro komponenty 1 a 2 nalézt neprotínající spojnici.*

## 2.10 Testování editoru

Předchozí verze editoru je obtížně testovatelná. Tato vlastnost je zapříčiněna úzkou provázaností editoru s *Visual Studiem*. Neboť pro spuštění testu by bylo nutné spouštět i *Visual Studio* s editorem načteným jako plugin, což by bylo nepraktické. Z tohoto důvodu budeme po nové verzi požadovat možnost spuštění editoru i bez *Visual Studia*.

Díky tomu již bude možné nad editorem spouštět automatizované unit testy. Zamysleme se tedy, které funkce editoru a jakým způsobem budeme chtít testovat.

### Interpretace analyzačních instrukcí

*Analyzační instrukce* a jejich interpretace tvoří důležitou část editoru. Je na nich založena celá analýza prováděná editorem. Proto je také důležité, aby jejich funkce byla dobře otestovaná.

Stěžejním úkolem *analyzačních instrukcí* je správná simulace virtuálního stroje, která se projevuje změnami v simulovaném běhovém prostředí. Z tohoto důvodu potřebujeme od testovacího frameworku možnost spouštět testovací programy zapsané v *analyzačních instrukcích*.

Abychom zjistili, že simulace proběhla správně, budeme muset ověřovat správnost hodnot lokálních i globálních proměnných.

### Překladače zdrojových instrukcí

Každý jazyk, jehož analýzu má editor podporovat, musí být překládán do *analyzačních instrukcí*. Takový překlad může být komplikovaný, proto je výhodné mít možnost výsledek překladu otestovat.

Překlad můžeme testovat buď kontrolou vygenerovaných instrukcí. Tento přístup zajistí velmi přesné otestování. Nicméně napsat test, který bude obsahovat všechny kontrolní instrukce, by byl velmi pracný.

Jiným přístupem je testování chování vygenerovaných instrukcí. Správnost překladu *zdrojových instrukcí* pak můžeme testovat podle hodnot uložených v proměnných.

Z tohoto důvodu potřebujeme, aby testovací framework dokázal spouštět překlad a následnou interpretaci vygenerovaných instrukcí.

### Typové definice

Typové definice editor používá pro simulaci chování .NET typů. Jejich prostřednictvím také získává informace o schématu kompozice a editacích, které může uživatel provádět.

Některé typové definice proto mohou mít netriviální implementace a je důležité mít možnost je otestovat. Příkladem takové netriviální typové definice může být kompoziční kontejner. Kromě kompozičního algoritmu, který musí zjišťovat vztahy mezi komponentami a případně generovat chybová hlášení, také řeší editace přidávání a odebírání komponent.

Testovací framework proto bude umožňovat použití typových definic a testování správnosti výsledků analýzy.

### Editace ve zdrojových kódech

Důležitou vlastností editoru je nabízení editací zdrojových kódů. Z kapitoly 2.6 víme, že změny ve zdrojových kódech mohou být v některých situacích velmi komplexní. Editace se tvoří na základě informací získaných v průběhu parsování. Ty

jsou pak předány v průběhu analýzy typovým definicím, které teprve vytvoří patřičné editace.

Uvědomme si, že pro automatické testování editací musíme mít možnost je také automaticky vyvolat. Architektura editoru proto musí zohlednit možnost vyvolávání editací i bez uživatelského zásahu.

Způsob testování editací je poté už přímočarý. Musíme mít možnost editaci spustit na zadaném zdrojovém kódu. Její správnost pak ověříme porovnáním výsledného kódu se vzorovým.

### Vykreslování schématu kompozice

V kapitole 2.9 jsme nastínili nutnost dvou základních požadavků, které budou řídit vykreslování. Prvním požadavkem dostatečná vzdálenosti mezi položkami zobrazenými na stejné úrovni tak, aby nedocházelo k jejich překrývání. Druhým požadavkem je tvorba neprotínajících spojnic.

I v případě vykreslování proto můžeme požadovat možnost testování. Pro rozmístění položek využijeme testování minimální vzdálenosti mezi položkami. U vykreslování spojnic můžeme testovat protínání se zobrazenými položkami.

Od testovacího frameworku budeme proto požadovat možnost rozmístit zobrazené položky na zadané pozice. Následně budeme muset spustit pozicovací algoritmus. Po provedení úprav pozic algoritmem budeme ověřovat minimální vzdálenost zobrazených položek a jejich protínání spojnicemi.

## 2.11 Požadavky na architekturu editoru

Ze zkušeností s vývojem předchozí verze editoru popsaným v kapitole 1.5 víme, že je nutné navrhnout architekturu s ohledem na snadnou testovatelnost. V předchozí kapitole jsme zjistili, které části editoru je nutné testovat. Navrhneme tedy požadavky na architekturu, která takové testování umožní. V této kapitole se proto zamyslíme nad členěním editoru do knihoven a jejich vzájemnými závislostmi.

### Analýza

Klíčovou funkcí editoru je provádění analýzy zadaného *composition pointu.* Aby bylo analýzu možné testovat nezávisle na ostatních funkcích editoru, oddělíme ji do samostatné knihovny *MEFEditor.Analyzing.dll*. Součástí implementace analýzy je interpretace *analyzačních instrukcí* a základní algoritmy pro tvorbu editací, které vycházejí z informací uložených do *analyzačních instrukcí* v době jejich překladu. Můžeme tedy vidět, že analyzační knihovna není závislá na žádných jiných službách editoru a může být proto dobře testována.

### Vykreslování

Další nezávislou funkcí, kterou od editoru požadujeme je vykreslování schématu kompozice. Pro správné vykreslení totiž stačí pouze informace o vztazích a vzhledu zobrazovaných položek. Spolu s informacemi, jaké editace má která položka nabízet, můžeme poskytnout kompletní schéma kompozice. Můžeme tedy vykreslování schématu kompozice implementovat v knihovně *MEFEditor.Drawing.dll*. Podkladem pro vykreslení pak bude definice, jak má schéma kompozice vypadat a jakým způsobem má reagovat na uživatelské vstupy. Tím získáme vykreslování nezávislé na analýze a můžeme ho proto snadno testovat.

### Typový systém

Práce editoru je však komplexní a musí proto využívat služby obou knihoven. Potřebujeme tedy knihovnu, která zajistí jejich propojení. Vhodným místem pro toto propojení je typový systém, neboť pro analýzu zajišťuje abstrakci typového systému .NET popsanou v kapitole 2.8 a zároveň podle typů určuje vzhled zobrazovaných komponent, katalogů a kontejnerů, jak jsme zjistili v kapitole 2.9.

Typový systém tedy bude implementován v knihovně *MEFEditor.TypeSystem.dll*, která bude obsahovat abstrakci typového systému .NET, a podporu pro tvorbu podkladů pro vykreslení schématu kompozice. Knihovna bude závislá na analýze a vykreslování, nicméně obě bude možné používat v testovacím frameworku, proto i typový systém s typovými rozšířeními a editacemi, které poskytují, bude možné testovat.

### Spolupráce s Visual Studiem

Ze zkušeností s vývojem předchozí verze editoru jsme v kapitole 1.6 zjistili, že automatické testování funkcí závislých na *Visual Studiu* je problematické. Z tohoto důvodu se je budeme snažit vyčlenit do samostatné knihovny *MEFEditor.Interoperability.dll*. Pro přílišnou provázanost s *Visual Studiem* však tento kód nebude dobře testovatelný.

### Uživatelská rozšíření

Důležitou vlastností editoru je rozšiřitelnost, protože není možné v rámci této práce pokrýt veškeré konstrukce z MEF a .NET. Z těchto důvodů bude editor nabízet rozšiřitelnost o typové definice a podporu pro analýzu dalších jazyků. Kvůli možnosti přizpůsobit vzhled schématu kompozice bude také možné přidávat definice zobrazení.

Rozšiřitelnost editoru nebude vyžadovat složité oddělení rozšiřujících částí od editoru, tak jak to umožňuje MAF. Kvůli jednoduchosti použití si tedy vystačíme s rozšiřitelností poskytovanou MEF. Editor bude tedy definovat *složku rozšíření*, ve které bude z knihoven načítat komponenty odpovídající rozšířením.

### Plugin do Visual Studia

Aby mohl editor fungovat jako plugin v rámci *Visual Studia*, je nutné vytvořit knihovnu, která provede potřebné propojení. Podrobnější popis způsobu vývoje pluginu je popsán v následující kapitole 3.1.

Knihovna pluginu bude nazvána *MEFEditor.Plugin.dll* a bude obsahovat jednak kód, který zajistí načtení editoru do *Visual Studia*, ale také kód, který definuje uživatelské rozhraní editoru. Tato knihovna tedy bude využívat analýzu, vykreslování, typový systém i knihovnu pro spolupráci s *Visual Studiem*. Kvůli značné provázanosti s *Visual Studiem* tato knihovna nebude dobře testovatelná.

### Konzolová aplikace pro vývoj a ladění

Aby bylo možné editor spouštět i mimo *Visual Studio*, potřebujeme nějakou spustitelnou assembly, ve které bychom ho mohli spouštět. Tuto assembly nazveme *MEFEditor.TestConsole.exe*. Bude využívat knihoven analýzy, vykreslování a typového systému. Díky tomu nebude mít žádnou vazbu na *Visual Studio* a bude moct být spouštěna bez něj. Součástí této assembly budou rutiny pro přehledné vypisování ladících výpisů a testování jednotlivých částí editoru.

### Testovací framework

*Visual Studio* nabízí nástroje pro automatizované testování. Využijeme je tedy pro vývoj testovacího frameworku našeho editoru. Ten bude implementován v knihovně *MEFEditor.UnitTesting.dll*. Součástí knihovny bude sada testů editoru. Aby je bylo možné spouštět, budeme zde také potřebovat nástroje, které umožní splnit požadavky na testování (spouštění testů editací, interpretací, ověřování jejich správnosti, apod.) popsané v předchozí kapitole 2.10.

# 3 Rozšiřitelnost Microsoft Visual Studia 2012

V bakalářské práci [1] stejného autora, jako je tato diplomová práce, jsme rozebírali rozšiřitelnost *Microsoft Visual Studia 2010* vzhledem k potřebám editoru analyzovat zdrojové kódy. Možnosti rozšiřitelnosti jsou pro verzi *Microsoft Visual Studio 2012*, na kterou je editor zaměřen, v aspektech důležitých pro implementaci editoru totožné. Z těchto důvodů je i popis rozšiřitelnosti *Visual Studia* shodný se zmíněnou bakalářskou prací.

## 3.1 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 v návodu na MSDN [18].

## 3.2 EnvDTE.DTE

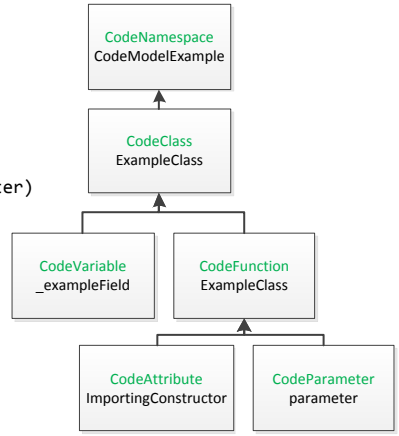
Základní přístup ke službám, které *Visual Studio* nabízí, probíhá přes rozhraní `DTE`. Pro potřeby editoru budeme z `DTE` využívat události týkající se uživatelských akcí a informace o otevřeném solution a jeho projektech. Za aktivní solution budeme z pohledu editoru považovat solution, dostupné v položce `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` lze nalézt na stránce s dokumentací [19].

## 3.3 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* [21], které reprezentují dostupné typové definice, jejich metody, atributy a další elementy zdrojového kódu.

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




*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 výjimku. 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 výjimku. Tato situace nastává, když je mezi předky třídy uveden identifikátor, který nedefinuje žádný dostupný typ. Vyvolávání těchto výjimek pak může zpomalovat prohledávání *Code Model*. Podrobné informace o použití *Code Model* jsou k dispozici v návodu na MSDN [20].

## 3.4 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`.

# 4 Implementace editoru

V následujících kapitolách si popíšeme implementaci editoru ve formě pluginu, která je dostupná v solution *MEFEditor_v2.sln* z přílohy [A]. Kompilace tohoto pluginu probíhala s využitím *Visual Studia 2012* s nainstalovaným *Microsoft Visual Studio 2012 SDK* [22], které je určené pro vývoj rozšíření *Visual Studia*.

Jako součást této práce je dále implementována konzolová aplikace, která usnadňuje vývoj uživatelských rozšíření editoru. Díky této aplikaci můžeme uživatelská rozšíření testovat v prostředí editoru, které však běží samostatně a ne jako plugin *Visual Studia*. Její použití podrobně popíšeme v kapitole 6.6.

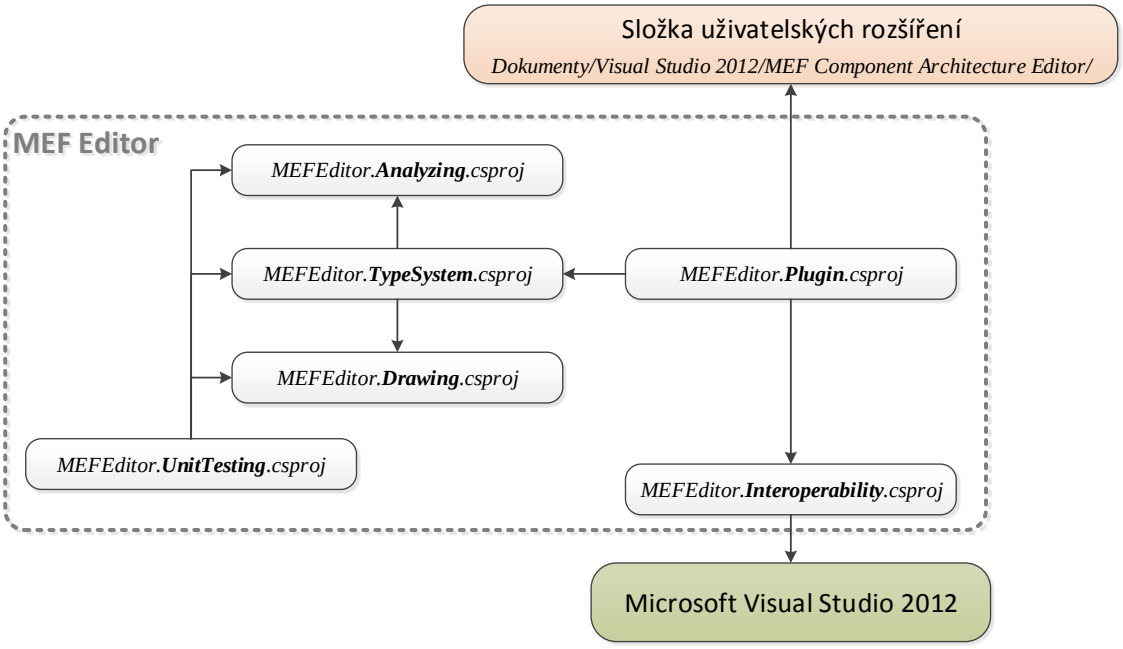
V rámci této práce také implementujeme *doporučená rozšíření* umožňující analýzu základních konstrukcí MEF a poskytující podporu pro zpracování jazyka C# a zkompilovaných assembly. *Doporučená rozšíření* podrobně popíšeme v kapitole 6.7.

Ke všem těmto implementacím existují dokumentace automaticky generované ze zdrojových kódů, které se nachází v příloze [F].

## 4.1 Plugin Visual Studia

***Namespace**: MEFEditor.Plugin*

Z pohledu struktury implementace našeho editoru je nejdůležitější assembly *MEFEditor.Plugin*. Zde dochází k nahrání všech potřebných knihoven a také k nahrání editoru do *Visual Studia*. K pochopení vztahů mezi knihovnami nám pomůže následující obrázek 4-1, zachycující závislosti jednotlivých částí našeho editoru:




*4-1 Celková architektura editoru ukazující vztahy mezi jednotlivými knihovnami.*

Z tohoto obrázku můžeme vidět základní architekturu editoru. Vstupním bodem architektury je *Visual Studio*, které nahraje *VsPackage* assembly implementující

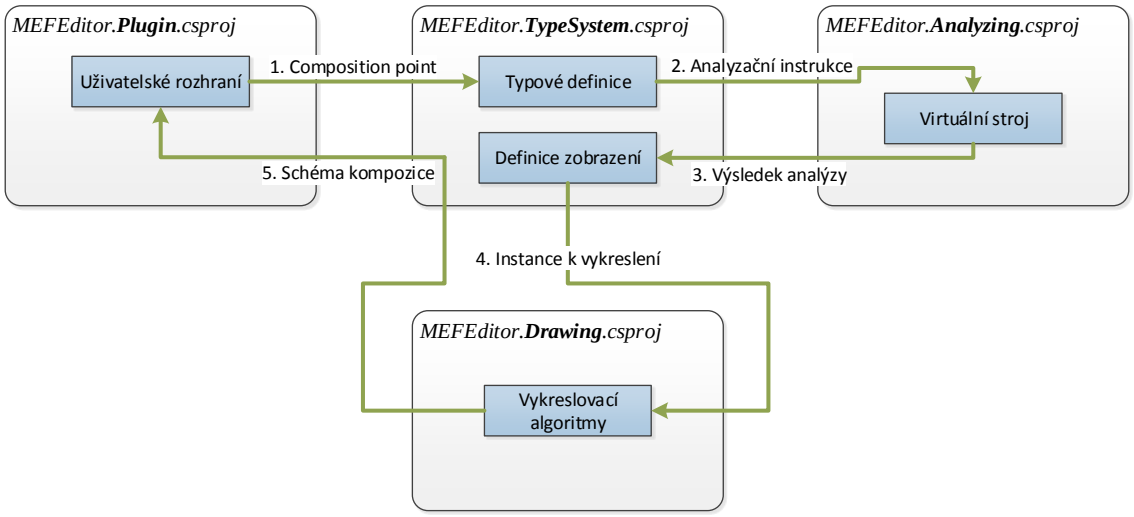
plugin našeho editoru. V této assembly se odehrává inicializace editoru, která začíná vyhledáním dostupných uživatelských rozšíření ze *složky uživatelských rozšíření* popsané později v kapitole 5.1.

Pomocí získaných uživatelských rozšíření je načtena assembly typového systému *MEFEditor.TypeSystem*, která zajišťuje nahrávání instrukcí pro analyzační knihovnu *MEFEditor.Analyzing*. Typový systém také obsahuje definice vykreslení *instancí* jednotlivých typů. Spolu s vykreslovací knihovnou *MEFEditor.Drawing* pak dokáže uživateli zobrazit schéma kompozice.

Rutiny zajišťující propojení editoru a jeho uživatelských rozšíření s *Visual Studiem* jsou vyčleněné do samostatné assembly *MEFEditor.Interoperability*.

Testování editoru implementuje assembly *MEFEditor.UnitTesting*, která dokáže testovat interpretaci analyzační knihovny, služby typového systému i některé části vykreslovací knihovny. Automatické testování není podporováno pro assembly pluginu, ani pro knihovnu *MEFEditor.Interoperability*, neboť závisejí na *Visual Studiu*, které by bylo nutné v rámci testů spouštět. Takový přístup by však byl nepraktický, jak jsme zjistili v kapitole 1.6.

Hlavním úkolem editoru je zobrazení schématu kompozice na základě metody, která je z hlediska kompozice důležitá. Takovou metodu nazýváme *composition point*, který jsme definovali v kapitole 2.4. Jakým způsobem je *composition point* zpracován znázorňuje následující obrázek 4-2:




*4-2 Zpracování composition pointu editorem.*

Z tohoto obrázku vidíme, že nejprve je *composition point* přeložen v assembly typového systému *MEFEditor.TypeSystem* na *analyzační instrukce* definované v kapitole 2.7. Jejich interpretací virtuálním strojem z analyzační knihovny *MEFEditor.Analyzing* získáme informace, ze kterých můžeme vykreslit schéma kompozice a nabídnout jeho editace. Toto schéma nakonec zobrazíme uživateli v uživatelském rozhraní.

Editor automaticky překresluje schéma kompozice, pokud zjistí změnu ve zdrojových kódech nebo knihovnách analyzovaných v rámci analýzy *composition pointu*. Toto překreslování je založeno na notifikacích, které editor dostává z typového systému, který musí sledovat změny v reprezentovaných assembly. Pokud je notifikována změna v metodě, která byla volána v průběhu interpretace zobrazovaného *composition pointu*, obnoví editor schéma kompozice dle pozměněné implementace.

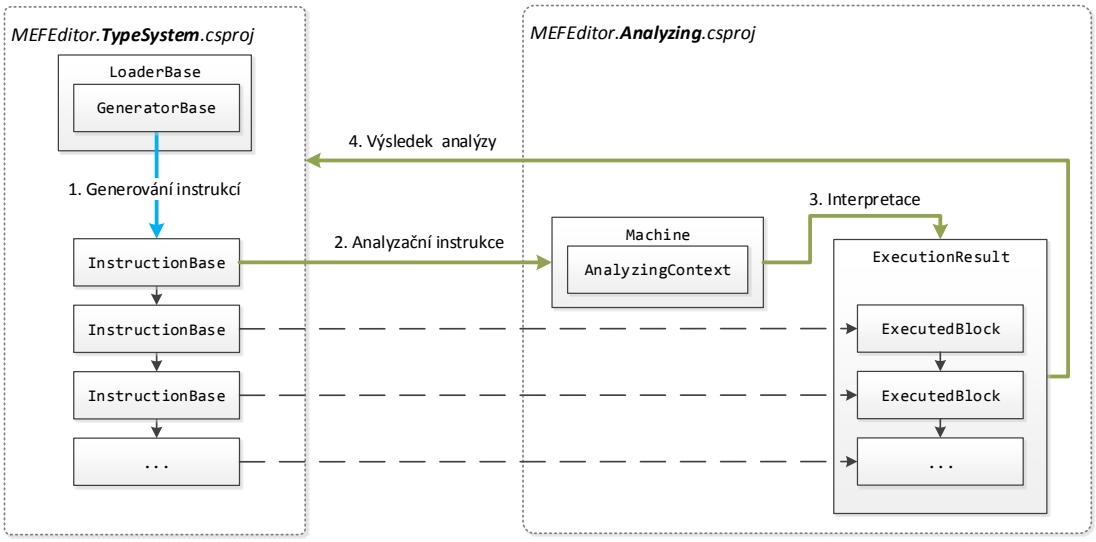
Nyní známe základní strukturu našeho editoru a víme, jakým způsobem spolu jednotlivé části komunikují. V následujících kapitolách si implementaci editoru popíšeme podrobněji. Začneme kapitolou 4.2 popisující analyzační knihovnu, která umožňuje interpretaci *analyzačních instrukcí*. Následovat bude kapitola 4.3, kde se seznámíme s typovým systémem, který *analyzační instrukce* poskytuje. Dále si popíšeme v kapitole 4.4 tvorbu editací a v kapitole 4.5 vykreslování schématu kompozice, využívající výsledek analýzy. Nakonec si ještě v kapitole 4.6 představíme vlastnosti testovacího frameworku našeho editoru.

## 4.2 Analyzační knihovna

***Namespace**: MEFEditor.Analyzing*

Úkolem analyzační knihovny je poskytnutí virtuálního stroje, který dokáže interpretovat *analyzační instrukce*. Jak jsme si popsali v kapitole 2.7, *analyzační instrukce* jsou navržené s ohledem na jednoduchou sémantiku, aby je bylo možné snadno analyzovat. Neřeší proto typovost zpracovávaných *instancí*, ale umožňují tuto sémantiku přidat typovým systémem.

Způsob komunikace s assembly typového systému *MEFEditor.TypeSystem*, a následný průběh zpracování instrukcí virtuálním strojem analyzační knihovny *MEFEditor.Analyzing* je patrný z následujícího obrázku 4-3:




*4-3 Schéma zpracování instrukcí virtuálním strojem v analyzační knihovně.*

V tomto obrázku můžeme vidět základní architekturu analyzační knihovny. Hlavní částí této knihovny je virtuální stroj implementovaný třídou `Machine`, která je podrobně popsaná v kapitole 4.2.1. Třída `Machine` udržuje kontext reprezentovaného prostředí v podobě objektu třídy `AnalyzingContext`. Ten umožňuje *analyzačním instrukcím* měnit stav prostředí, například ukládáním hodnot proměnných nebo voláním metod.

Třída `Machine` také zajišťuje interpretaci *analyzačních instrukcí*, které jsou do analyzační knihovny nahrávány pomocí implementací abstraktní třídy `GeneratorBase` umožňující jejich generování. Každý generátor, jak nazýváme

objekt odvozený od třídy implementující `GeneratorBase`, generuje instrukce pouze jedné konkrétní metody.

Protože v průběhu interpretace potřebujeme generovat instrukce každé volané metody, využíváme pro nahrávání generátorů loader v podobě objektu třídy implementující abstraktní třídu `LoaderBase`. Tento loader je volán, když je potřeba získat instrukce nějaké metody. Pro ni poskytne patřičný generátor, ze kterého jsou instrukce metody generovány.

Výsledkem analýzy *composition pointu* je řetězec objektů třídy `ExecutedBlock`, které reprezentují bloky interpretovaných instrukcí. V těchto blocích jsou uchovány informace, které editor potřebuje pro vykreslení schématu kompozice a jeho editace.

#### Reprezentace objektů

Interpretace virtuálním strojem pracuje s *instancemi*, které simulují chování objektů reprezentovaných v průběhu interpretace. Nicméně analyzační knihovna nerozlišuje typ objektu reprezentovaného *instancí*, ale pracuje pouze s abstrakcí metod a jejich argumentů.

Abstrakci typů totiž musí zajistit vyšší vrstva, která analyzační knihovnu využívá. V případě editoru je touto vrstvou typový systém. Aby dokázal rozlišit typ každé *instance*, využívá třídu `TypeDescriptor`, která implementuje abstraktní třídu `InstanceInfo` ukládané do *instance* při jejím vytváření. Spolu s identifikátorem metody v podobě objektu třídy `MethodID` má typový systém dostatečné prostředky pro poskytnutí požadované abstrakce.

### 4.2.1 Třída Machine

Vstupním bodem do API analyzační knihovny je implementace virtuálního stroje třídou `Machine`. Tato implementace zajišťuje zpracování *analyzačních instrukcí*. Díky jejich malému počtu a jednoduché sémantice v nich můžeme snadno identifikovat důležité MEF konstrukty a umožnit editace nad zdrojovými kódy.

Virtuální stroj dovoluje nastavení některých základních vlastností. Konkrétní nastavení je definováno implementací abstraktní třídy `MachineSettingsBase` předávaného jako argument při konstrukci virtuálního stroje. Nastavením je možné určit jmenné konvence používané pro inicializátory sdílených objektů, způsob vyhodnocování podmínek a také obsahuje pomocné metody, používané při generování instrukcí.

V průběhu interpretace potřebujeme kontrolovat počet vytvořených *instancí*, aby nedocházelo k neomezené spotřebě operační paměti. Toho analyzační knihovna dosahuje tak, že dovolí vytvářet *instance* pouze v rámci konkrétního objektu `Machine`. Virtuální stroj si registruje každou vytvořenou *instanci*, a pokud zjistí překročení zadaných limitů, okamžitě zastaví interpretaci. Tento limit je předáván nastavením jako vlastnost `Machine.InstanceLimit`.

Dalším limitem, který je důležitý z hlediska editoru, je omezení doby výpočtu virtuálního stroje. Z tohoto důvodu je také registrována každá zpracovaná instrukce a při překročení zadaného počtu opět dojde k zastavení interpretace. Tento limit se také předává v nastavení a to v podobě vlastnosti `Machine.ExecutionLimit`.

Průběh interpretace je zahájen načtením instrukcí vstupní metody do kontextu. Virtuální stroj pak instrukce po jedné zpracovává, dokud není dosaženo konce vstupní metody, nebo dokud nedojde k překročení některého z limitu nebo běhové chybě.

Interpretace virtuálním strojem je založena na nahrávání instrukcí a jejich vykonávání. Nahrávání je zajištěno implementací `LoaderBase` objektu, který je

vyžadován při spuštění v `Machine.Run`. Kdykoliv virtuální stroj potřebuje instrukce nějaké metody, nejprve získá z uvedeného loaderu generátor, podle identifikátoru v podobě objektu třídy `MethodID`. Získaný generátor je pak využit ke generování instrukcí metody zadaným emitorem.

Po skončení interpretace virtuální stroj poskytne množinu *instancí*, které byly vytvořeny spolu s řetězcem objektů `ExecutedBlock`, vytvořeného ze sekvence interpretovaných instrukcí. Tyto bloky kromě informací o instrukcích navíc obsahují seznamy *instancí*, které byly instrukcemi zpracovány. Díky tomu může editor poskytnout veškeré potřebné editace ve schématu kompozice.

### 4.2.2 Abstraktní třída LoaderBase

Abychom mohli simulovat volání metod, potřebujeme nejdříve znát jejich instrukce. Bylo by však neefektivní zjišťovat instrukce všech metod, proto využijeme mechanismus nahrávání instrukcí až na vyžádání virtuálním strojem. To je zajištěno konkrétní implementací třídy `LoaderBase`, která je poskytnuta virtuálnímu stroji při volání `Machine.Run`. V případě našeho editoru poskytuje implementaci třídy `LoaderBase` typový systém, který tak může vyhledávat implementace metod v načtených assembly.

Při zpracování instrukce volání je známý pouze identifikátor, v podobě objektu třídy `MethodID`, volané metody a hodnoty jejích argumentů. Tento identifikátor metody je chápán jako pojmenování, dle kterého metodu najdeme bez vzájemné závislosti s jejím skutečným umístěním. To je výhodné proto, že identifikátory a tudíž ani instrukce metod, které tyto identifikátory používají, nejsou ovlivněny změnami ve zdrojích volaných metod. Na druhou stranu je nutné umožnit nahrávání a vyhledávání metod virtuálnímu stroji v průběhu interpretace.

Nahrávání instrukcí metod dělíme do dvou základních kategorií. Statické nahrávání používáme v situacích, kdy přesně známe identifikátor volané metody v době generování instrukcí. To odpovídá případům volání nevirtuálních metod v .NET. Při statickém nahrávání instrukcí můžeme rovnou získat generátor instrukcí voláním `LoaderBase.StaticResolve`.

Druhým případem je dynamické nahrávání instrukcí metod. To je nutné pro případ, kdy v době generování instrukcí není možné přesně určit, kterou metodu bude nutné volat. To se může stát v případě simulace volání virtuálních metod v .NET. Abychom přesně dokázali určit volanou metodu, musíme nejdříve znát konkrétní typ objektu.

Dynamické nahrávání metod se provádí zjištěním konkrétního identifikátoru metody pomocí `LoaderBase.DynamicResolve` s informací o *instancích*, které jsou předávány jako argumenty volání. Dle těchto informací je pak loader schopen nalézt konkrétní metodu dle typu *instance*, na níž je metoda volána. Identifikátor konkrétní metody je vrácen virtuálnímu stroji, který tak už může získat instrukce metody stejně, jako v případě statického nahrávání instrukcí.

Všimněme si, že předávání identifikátoru metody v případě dynamického nahrávání instrukcí je výhodnější, než kdybychom rovnou obdrželi vygenerované instrukce. Instrukce, které získáme pro statický identifikátor metody, jsou tímto identifikátorem jednoznačně určeny a můžeme je při dalším volání využít znovu, bez nutnosti opakovaného generování.

Jakmile virtuální stroj dostane generátor volané metody, je vytvořen objekt třídy `CallEmiter`, který je předán metodě `GeneratorBase.Generate`. Tak je možné využít služby emitoru k vytváření instrukcí, labelů a přídavných informací

využitelných pro tvorbu editací. Po dokončení generování jsou generované instrukce předány kontextu `AnalyzingContext`, který je registruje a přidá do zásobníku volání.

### 4.2.3 Třída AnalyzingContext

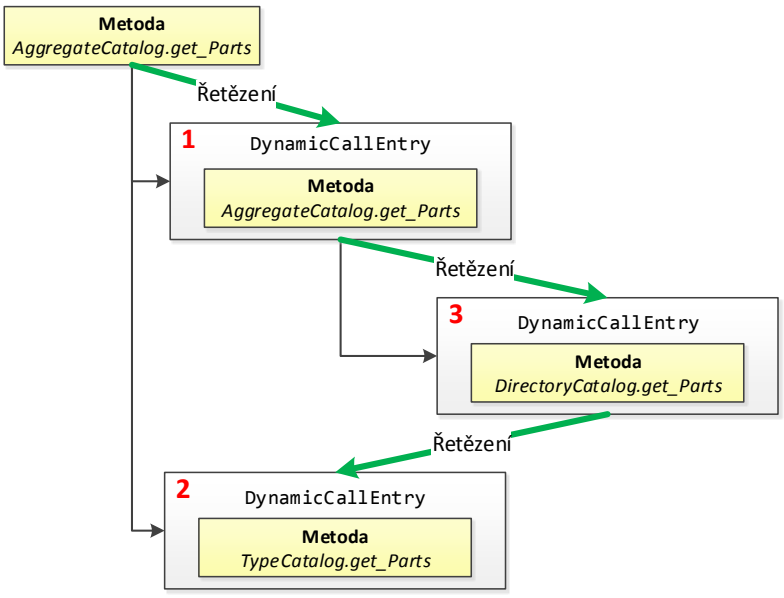
V průběhu interpretace potřebuje virtuální stroj ukládat hodnoty proměnných spolu s dalšími informacemi běhového prostředí, jako jsou například ukazatele vykonávaných instrukcí pro každé volání metody na zásobníku. Tuto funkčnost zajišťuje třída `AnalyzingContext`, simulující chování běhového prostředí interpretované aplikace.

Významnou částí tohoto prostředí je zásobník volání, udržující objekt třídy `CallContext` pro každou úroveň zásobníku. V každém kontextu volání jsou uložené hodnoty lokálních proměnných a instrukce vygenerované pro volání. Pro každé volání je také nutné udržovat ukazatel vykonávané instrukce v podobě jejího indexu. Instrukce skoku pak mohou být jednoduše implementovány změnou tohoto indexu.

Zásobník volání, tak jak jsme ho popsali, se chová běžně, jak je v programovacích jazycích zvykem. Nicméně pro potřeby typového systému budeme vyžadovat některé speciální vlastnosti. Protože virtuální stroj udržuje vlastní zásobník volání, musí být volání metod na *instancích* v uživatelských *typových definicích* psány asynchronním způsobem. To znamená, že metody přidané na zásobník volání virtuálního stroje jsou zavolány až po skončení metody uživatelské *typové definice*.

Z tohoto důvodu virtuální stroj umožňuje vytvářet dynamicky generovaná volání, která mohou být řetězena, což je nezbytné pro korektnost interpretace, pokud těchto dynamicky generovaných metod voláme více.

Pro lepší pochopení si ukážeme příklad, kdy pro získání komponent z `AggregateCatalog` potřebujeme dynamicky volat metodu `get_Parts` všech obsažených katalogů. V `AggregateCatalog` může však být vnořen další `AggregateCatalog`, proto i z dynamicky volané metody může dojít k dalšímu dynamickému volání, jak znázorňuje následující obrázek 4-4:




*4-4 Demonstrace řetězení dynamického volání.*

Z tohoto obrázku vidíme význam řetězení dynamických volání. Uvědomme si totiž, že tato volání jsou asynchronní, tudíž jsou nejprve vytvořena volání metod 1 a 2. Teprve po spuštění metody 1 dojde k vytvoření volání 3. Po skončení metody 1 bude díky řetězení vyvolána metoda 3 a až poté metoda 2.

Další důležitou službou, kterou poskytuje třída `AnalyzingContext` je poskytovatel editací `EditsProvider`. Zde jsou uchovávány všechny dostupné transformace definované nad *analyzačními instrukcemi*. Úkolem kontextu však je tyto transformace propagovat napříč voláními, aby uživatelská rozšíření mohla snadno vytvářet editace v místě, odkud jsou zavolané.

Dosud popsané služby se týkaly pouze lokálního kontextu metod. `AnalyzingContext` ale také zpřístupňuje globální služby reprezentované objekty `Machine` a `MachineSettings`, které mohou být použity v instrukcích nebo v uživatelských rozšířeních. Poslední globální službou nutnou pro virtuální stroj jsou globální proměnné. Měli bychom upozornit na rozdíl mezi globálními proměnnými a proměnnými definovanými ve vstupní metodě. Proměnné z metod totiž nejsou přístupné mezi jednotlivými voláními. Při interpretaci .NET metod však potřebujeme některé informace sdílet ve všech voláních. Toto se týká situací, kdy potřebujeme použít sdílenou *instanci* na reprezentaci statických tříd. Tato *instance* musí být totiž stejná pro všechna volání v rámci jednoho spuštění *composition pointu*. Díky globálním proměnným můžeme sdílenou *instanci* pro statickou třídu snadno nalézt.

### 4.2.4 Abstraktní třída GeneratorBase

Generování instrukcí pro metody je zajišťováno pomocí implementace třídy `GeneratorBase` získané z patřičného loaderu v průběhu interpretace. Tyto generátory jsou v případě našeho editoru získávány z typového systému, který je hledá v `AssemblyProvider` reprezentujícím assembly, kde je hledaná metoda definovaná.

Samotné generování je implementováno v `GeneratorBase.Generate` metodě, která přijímá emitor v podobě objektu třídy `CallEmiter` zpřístupňujícím služby potřebné pro generování instrukcí. Tento emitor obvykle bývá používán parsery nebo překladači pro překlad metod ze *zdrojových instrukcí* do *analyzačních instrukcí*. Protože se však může jednat o relativně výpočetně náročnou činnost, `GeneratorBase` automaticky provádí cachování vygenerovaných instrukcí. Tím se vyhneme opakovanému překládání již zpracovaných metod.

Abstrakce generátorů nám nabízí efektivní způsob jak generovat instrukce na vyžádání až v době, kdy jsou opravdu potřeba s možností cachování. Na druhou stranu s tím přichází nutnost, vytvářet nový generátor při každém pokusu o získání instrukcí metody, jejíž zdrojový kód byl změněn. Nicméně režie spojená s vytvářením objektu generátoru je zanedbatelná v porovnání s výhodami, které nám tento přístup nabízí.

### 4.2.5 Abstraktní třída Instance

Interpretované objekty jsou v editoru representovány implementacemi abstraktní třídy `Instance`. Použití těchto *instancí* v editoru je podobné, jako tomu bylo u předchozí verze. Nicméně na rozdíl od předchozí verze editoru využívá analyzační knihovna dva druhy implementací této třídy.

Bylo by totiž obtížné reprezentovat primitivní datové typy v nějakém speciálním formátu pro virtuální stroj. Místo toho raději zabalíme primitivní objekty v první implementaci *instance,* kterou je třída `DirectInstance`. Tento přístup nám

umožňuje využívat nativní volání na primitivních objektech, což zejména uplatníme při implementaci uživatelských rozšíření. Je zřejmé, že tento přístup nemusíme používat pouze pro primitivní typy. Stejného principu můžeme také využít u objektů, u nichž chceme zachovat nativní chování, čímž můžeme zvýšit rychlost interpretace.

Zabalování nativních objektů však není vždy dostatečné, protože musíme umět pracovat i s typy, které jsou nahrané ze zdrojových kódů, nebo protože chceme chování nativních objektů změnit z důvodu analýzy. K těmto účelům slouží druhá implementace *instance* nazvaná `DataInstance`. Její hlavní funkcí je uchovávání hodnot pro pojmenované položky, které umožní reprezentovat datové položky objektů.

Upozorněme však na fakt, že typový systém tyto položky nezpřístupňuje přímo, ale pro každou datovou položku vytváří getter a setter metodu. Uložené datové položky jsou určeny pouze pro interní implementaci metod. Tento přístup pak zjednodušuje práci parserům, které nemusí řešit datové položky odlišně od způsobu, jakým řeší volání metod a vlastností.

#### *4.2.5.1 Volání metod na instancích*

V předchozí verzi editoru byla každá *instance* pevně svázána se svým typem. Vzhledem k tomu, že v současné verzi provádíme interpretaci nad instrukční sadou, která nevyužívá typovost *instancí*, není takové provázání potřebné. Aby však vyšší vrstvy dokázaly poskytovat abstrakci typů, je možné implementovat třídu `InstanceInfo` s potřebnými informacemi, které budou uloženy do *instance*. V rámci editoru je `InstanceInfo` implementováno typovým systémem jako typový deskriptor třídou `TypeDescriptor`. Ten typovému systému slouží k tomu, aby mohl rozlišit dynamická volání, která závisí na konkrétním objektu. Typové deskriptory však nejsou pevně svázány s konkrétní definicí typu, tudíž změny jako je přidávání nebo ubírání metod typu v analyzovaných zdrojových kódech nemusí zneplatnit výsledek analýzy, jako tomu bylo v předchozí verzi editoru.

Negativum tohoto přístupu však spočívá v nemožnosti volat metody přímo na *instanci*, tak jako tomu bylo v předchozí verzi. Místo toho je nutné využít abstrakci poskytovanou typovým systémem, který zajišťuje že „volaná“ *instance* bude prvním argumentem volané metody.

#### *4.2.5.2 Vytváření instancí*

Abychom mohli pracovat s *instancemi*, musíme být nejdříve schopni je vytvořit. Tuto funkcionalitu zajišťuje virtuální stroj, který tak může registrovat celkový počet vytvořených *instancí* a poskytovat tak omezení bránící přílišné spotřebě paměti při analýze. Už jsme říkali, že pro potřeby editoru jsou dostupné dva druhy *instancí*. Proto také máme dva způsoby, jak *instance* vytvářet.

Prvním způsobem je přímé vytvoření *instance*, které je zajištěno metodou `Machine.CreateDirectInstance`. Tato metoda dostane argumentem nativní objekt, který chceme zabalit do *instance*. Výsledkem volání je objekt `DirectInstance` reprezentující zadaný nativní objekt.

Druhou možnost k vytvoření *instance* dává `Machine.CreateInstance`, která vytvoří prázdnou `DataInstance` a nebo `DirectInstance` obsahující `null`, podle toho, zda typ popsaný typovým deskriptorem dovoluje vytvářet přímé *instance*.

Všimněme si, že pro potřeby vytvoření *instance* není nutné udávat žádný konstruktor jako u předchozí verze editoru. Sémantika konstruktoru musí být zprostředkována vyšší vrstvou, kterou je v případě našeho editoru typový systém.

#### 4.2.5.3 Sdílené instance

V .NET se můžeme setkat s konceptem statických volání, která nepotřebují explicitně uvádět na kterém objektu se má metoda zavolat. Místo něj určíme třídu, která ale ve skutečnosti reprezentuje objekt sdílený pro všechna volání této třídy. Tento objekt je automaticky zkonstruován za pomoci statického konstruktoru těsně před prvním statickým voláním.

Abychom tento koncept dokázali pokrýt i v našem editoru, využívá typový systém globálních proměnných, ve kterých uchovává sdílené *instance*. Pokud hledaná *instance* v globálních proměnných není, je vytvořena a uložena do patřičné proměnné. Při vytváření je také použit statický konstruktor. Tato implementace zajišťuje, že bude pro každou statickou třídu vytvořena pouze jedna sdílená *instance* v rámci spuštění *composition pointu*. Navíc je vytváření prováděno těsně před prvním použitím a proto má stejnou sémantiku jakou můžeme pozorovat v .NET.

#### 4.2.5.4 Dirty instance

Protože není vždy možné přesně interpretovat celý *composition point*, nebo je to příliš výpočetně náročné, nabízí analyzační knihovna koncept *dirty instancí*. Tento koncept byl dostupný již v předchozí verzi editoru. Současná verze ho také podporuje, aby nebylo nutné v průběhu analýzy interpretovat všechny metody.

*Dirty instance* je *instance* označená příznakem *dirty*. Ten značí, že stav *instance* nemusí odpovídat stavu, který by byl dosažen přesnou interpretací všech metod. Typické použití tohoto příznaku je v případech, kdy například kvůli výkonu editoru nechceme načítat implementaci metody z referencované assembly. Místo interpretování metody jsou její argumenty označeny *dirty* příznakem.

Tento příznak se pak šíří při každém dalším volání, do kterého vstupuje nějaká *dirty instance*. Místo samotného volání jsou totiž zbylé *instance* také označeny jako *dirty*. Díky tomu může editor zjistit, že stav některých *instancí* není přesně známý a podle toho například upozorní uživatele.

### 4.2.6 Implementace analyzačních instrukcí

V kapitolách analýzy jsme popsali instrukce, které budeme potřebovat pro simulaci konstruktů .NET. Nyní si ukážeme, jakým způsobem jsou tyto instrukce v editoru implementovány.

Každá *analyzační instrukce* se projevuje určitou změnou provedenou v interpretačním prostředí reprezentovaném objektem třídy `AnalyzingContext`, který je předáván instrukcím v metodě `InstructionBase.Execute`. Díky tomu může konkrétní implementace instrukce provést patřičné změny v prostředí.

Každá instrukce patří do nějakého bloku instrukcí. Ten obvykle odpovídá jedné *zdrojové instrukci*. K bloku *analyzačních instrukcí* můžeme pro ladící účely nastavit dodatečné informace v podobě objektu třídy `InstructionInfo`. Nastavit můžeme například textový komentář jako vlastnost `InstructionInfo.Comment`. Z hlediska nabízení editací je důležitá vlastnost `InstructionInfo.BlockTransformProvider`, která nastavuje poskytovatele transformací pro blok instrukcí. Pokud je tento poskytovatel dostupný, může ho editor využít na změnu pořadí bloků instrukcí, což je výhodné v případech, kdy potřebujeme změnit pořadí řádků zdrojového kódu pro účely editace. Další podporu editací můžeme nalézt u instrukcí přiřazení a volání, díky kterým je možné zdrojové instrukce mazat, nebo manipulovat s argumenty volání.

*Analyzační instrukce* byly navrženy pro potřeby překladu *zdrojových instrukcí* do společného jazyka, který můžeme interpretovat a analyzovat. Z tohoto důvodu předpokládáme, že nejčastěji bude nutné generovat instrukce z parseru nebo překladače. Na tento účel se dobře hodí emitor, na kterém voláme metody pro jednotlivé instrukce v pořadí, v jakém je chceme interpretovat virtuálním strojem.

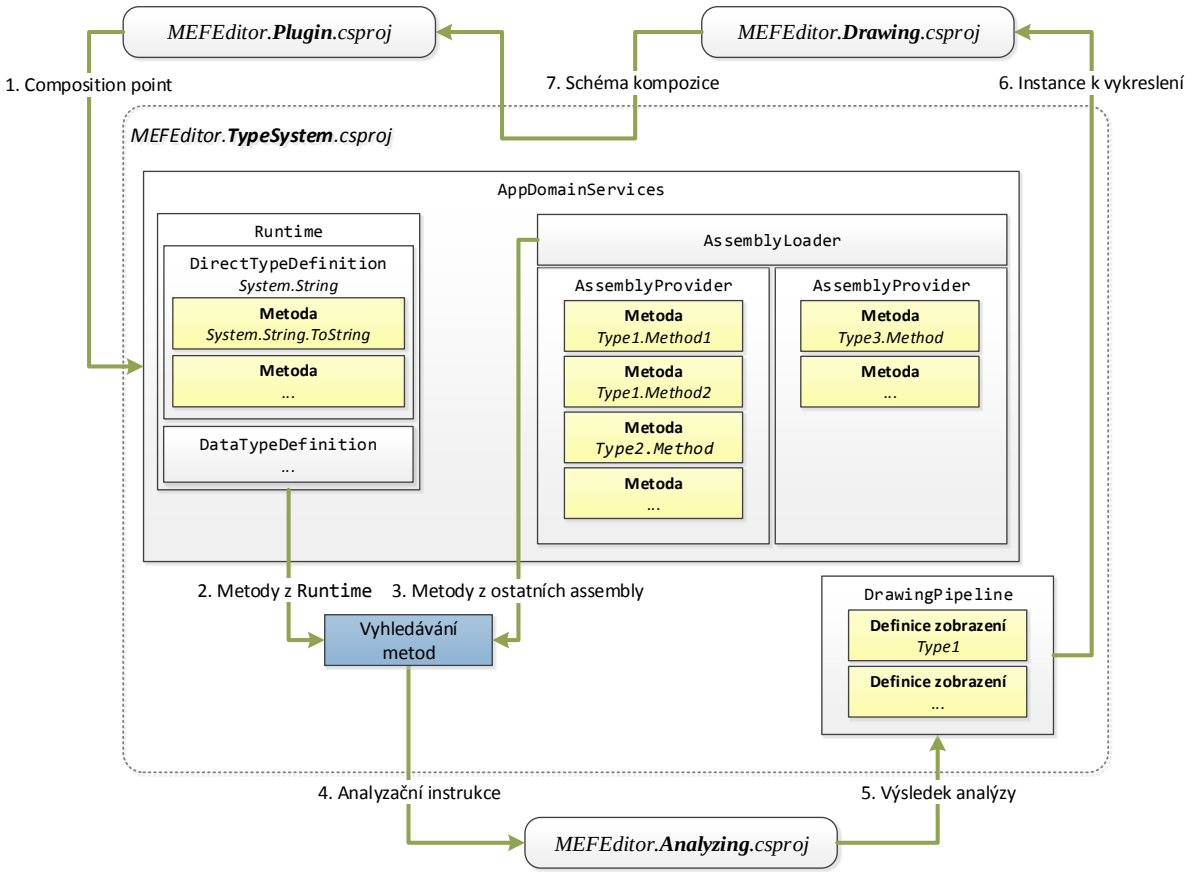
Takový emitor může navíc nabízet pomocné metody, které odstíní některé nízkoúrovňové operace jako je vytváření několika instrukcí pro každé statické volání metody. Stejný přístup můžeme vidět i u generátorů instrukcí CIL, používaných v .NET *Reflection*.

## 4.3 Typový systém

***Namespace**: MEFEditor.TypeSystem*

Hlavní úlohou typového systému v našem editoru je poskytování abstrakce typů nad analyzační knihovnou, abychom mohli simulovat volání tak, jak je známe z .NET. I v rámci našeho editoru proto používáme obdobnou hierarchii typového systému. Nejvýše stojí aplikační doména v podobě třídy `AppDomainServices` popsané v kapitole `4.3.2`, která udržuje aktuálně nahrané assembly, reprezentované konkrétními implementacemi abstraktní třídy `AssemblyProvider`. Ty poskytují informace o typech definovaných v assembly, kterou reprezentují.

Strukturu typového systému s napojením na další knihovny editoru zachycuje následující obrázek 4-5:




*4-5 Struktura typového systému a jeho napojení na další části editoru.*

Z tohoto obrázku můžeme vidět, že napojení na analyzační knihovnu *MEFEditor.Analyzing* spočívá v poskytování *analyzačních instrukcí* metod, které jsou zpracovávány v analyzační knihovně. Toto napojení zajišťuje třída `AssemblyLoader` implementující abstraktní třídu `LoaderBase`, která zprostředkovává mechanismus vyhledávání implementací metod podle jejich identifikátorů v podobě objektů třídy `MethodID`.

Metody jsou vyhledávané v aktuálně nahraných assembly, které jsou reprezentovány objekty třídy `AssemblyProvider`. V našem typovém systému navíc využíváme speciální assembly *Runtime* reprezentovanou třídou `RuntimeAssembly`, blíže popsanou v kapitole 4.3.4, která udržuje uživatelské *typové definice* získané při načtení editoru. Typy definované v *Runtime* mají při vyhledávání metod přednost, a proto je možné uživatelsky pozměnit chování libovolného typu.

Abychom byli schopni řešit i dynamická volání, je nutné uchovávat informaci o typu pro každou *instanci*. Toto zajišťuje třída `TypeDescriptor`, která obsahuje identifikátor reprezentovaného typu. Díky tomu můžeme při vyhledávání dynamicky volané metody zjistit konkrétní implementaci na základě typu volané *instance*.

Typy jsou také využívány pro napojení na vykreslovací knihovnu. Té poskytujeme informace o *instancích* k vykreslení, které jsou získány definicemi zobrazení s využitím třídy `DrawingPipeline`.

### Jména typů

V souvislosti s pojmenováváním typů v rámci implementace typového systému 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, kde druhým generickým parametrem označujeme dimenzi pole.

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. V následujícím seznamu se můžeme podívat na příklady takových jmen:

- ***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`

V současné verzi pracujeme s typy odlišně od předchozí verze editoru. Hlavní změnou je fakt, že editor nikde nevyžaduje kompletní reprezentaci typu. Typový systém je navržen tak, aby dokázal pracovat pouze s jednotlivými metodami, které

typ definuje. Toto by však nestačilo pro zjišťování informací o dědičnosti, proto musíme navíc udržovat informace o předcích každého typu ve formě řetězců objektů třídy `InheritanceChain`. Tento přístup je výhodný zejména z důvodu častých změn, které jsou ve zdrojových kódech prováděny. Při přidání, ubrání nebo změně metody totiž nemusíme přestavovat celou reprezentaci typu tak, jako tomu bylo v předchozí verzi editoru.

### 4.3.1 Třída TypeDescriptor

Základním identifikátorem typu, který je v typovém systému používán je typový deskriptor v podobě třídy `TypeDescriptor`. Ten je přiřazován každé vytvořené *instanci* na základě typu, se kterým je vytvořena. Základní výhodou typového deskriptoru je fakt, že není žádným způsobem odkázán na skutečné implementace typů, tudíž není ovlivněn jejich změnami v průběhu editací. Typový deskriptor je totiž nutné chápat jako pojmenování, dle kterého je konkrétní typ dohledatelný.

Současná implementace typového systému umožňuje vytvářet typový deskriptor z *fullname*, *rawname* i z nativních typů v .NET. Tyto služby pak mohou být použité v implementovaných rozšíření, takže je snadnější dodržet jmenné konvence typového systému. Jména typů, se kterými typový deskriptor pracuje, jsou odvozena od formátu, použitém v C#. Odlišností je, že *fullname* nesmí obsahovat žádné mezery. Generickým argumentem pak může být *fullname*, číselný argument, nebo generický parametr, který je složen z pořadového čísla parametru prefixovaným znakem `@`.

### 4.3.2 Třída AppDomainServices

Aplikační doména reprezentuje kolekci poskytovatelů assembly, která je aktuálně dostupná v interpretačním prostředí. Assembly do aplikační domény můžou být nahrány přes reference z již nahraných assembly a nebo uživatel aplikační domény, kterým je v případě našeho editoru implementace pluginu. Každý druh assembly je reprezentován konkrétní implementací třídy `AssemblyProvider`, která se například specializuje na reprezentaci assembly získané ze zkompilovaných knihoven.

Stejně jako v předchozí verzi, obsahuje interpretační prostředí jednu speciální assembly, která se jmenuje *Runtime*. V této assembly jsou nahrány všechny dostupné uživatelské *typové definice* a jejich definice zobrazení. Při vyhledávání implementace metody pro nějaké volání mají pak implementace v *Runtime* přednost a je proto možné uživatelsky předefinovat chování libovolné metody.

Hlavním úkolem aplikační domény je implementace služeb pro poskytovatele assembly, které potřebují pro vzájemnou komunikaci. Jsou zde tedy řešeny rutiny pro vyhledávání metod v referencovaných assembly a informace o dědičnosti, které využijí především parsery a překladače.

Aplikační doména také zprostředkovává některé běhové služby, jako jsou služby pro zkoumání dědičnosti typů a nahrávání nových assembly využívané například z typové definice pro `DirectoryCatalog`, která potřebuje vyhledávat assembly v zadané složce. Díky tomu může aplikační doména efektivně řešit mapování cest jednotlivých assembly a poskytnout tak prostředí vhodné pro analýzu konfigurace REA.

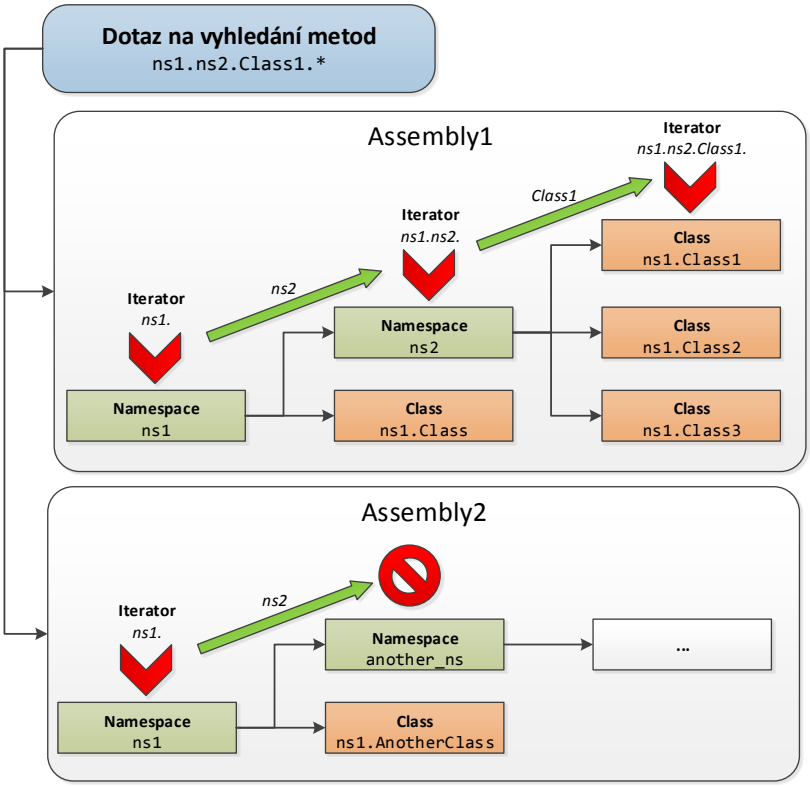
### 4.3.3 Abstraktní třída AssemblyProvider

Typy jsou v .NET organizovány do assembly, ve kterých jsou implementované. Stejně tak typový systém používá reprezentaci assembly pomocí `AssemblyProvider`, ke zpracování knihoven a projektů, ze kterých získává definice typů. V reprezentované assembly pak hlídá změny a notifikuje o nich editor. Ten pak na jejich základě dokáže efektivně překreslovat zobrazené schéma kompozice například na základě změn prováděných uživatelem ve zdrojovém kódu.

Jelikož `AssemblyProvider` má přehled o veškerých typech definovaných v assembly, musí také poskytovat informace o komponentách, které jsou v ní obsaženy. Po nahrání assembly je proto nutné prozkoumat veškeré implementované typy a zjistit, zda se nejedná o komponenty. Všechny objevené komponenty jsou pak nahlášeny editoru, který je může přehledně zobrazit uživateli.

#### *4.3.3.1 Vyhledávání metod*

Ze zkušeností z předchozí verze editoru jsme zjistili, že aktivní poskytování typů a implementací metod ze zkoumaných assembly je zbytečně výpočetně a paměťově náročné. Proto v současné verzi využíváme pasivní přístup k assembly, kdy jsou editoru poskytovány pouze ty metody, o které si v průběhu interpretace explicitně řekne. Tento přístup dovede navíc efektivně využít struktury zkoumané assembly, jak je znázorněno na následujícím obrázku 4-6:




*4-6 Hierarchické vyhledávání metod zefektivňuje procházení assembly.*

V tomto obrázku můžeme vidět, jak funguje hierarchické vyhledávání v assembly se stromovou strukturou, kterou mají například assembly poskytované ze zdrojových kódů. Díky postupnému prohledávání nemusíme procházet například assembly, které vůbec neobsahují potřebný *namespace*. Tuto funkcionalitu zajišťuje

třída `SearchIterator`, která reprezentuje konkrétní pozici v assembly dle aktuálně vyhledávaného jména metody.

Poznamenejme však, že toto vyhledávání je zejména nutné pro vyhledávání informací o metodách při překladu. V průběhu interpretace získáváme implementaci přímo podle konkrétního objektu třídy `MethodID`.

#### *4.3.3.2 InheritanceChain*

V současné verzi editoru nevyužíváme přímou konstrukci typů, abychom se vyhnuli problémům, které souvisí se synchronizací jejich reprezentace s definicemi ve zdrojových kódech. Proto musíme využít jiný mechanismus pro zachycení vztahů mezi typy. K tomuto účelu slouží `InheritanceChain`, poskytovaný objekty `AssemblyProvider`. Každá reprezentovaná assembly je zodpovědná za konstrukci té části řetězce dědičnosti, jejíž typy definuje. Díky tomu není problematické řešit dědičnost napříč různými assembly.

### 4.3.4 Třída RuntimeAssembly

Jak jsme již nastínili, stejně jako v minulé verzi editoru používá současná verze implicitní assembly *Runtime*, která slouží pro uživatelské definice typů. Tato assembly je implementovaná jako třída `RuntimeAssembly` implementující abstraktní třídu `AssemblyProvider`. Chová se tedy podobně jako běžná assembly s tím rozdílem, že při vyhledávání implementací metod dostává přednost před všemi ostatními. Tento mechanismus spolehlivě zajišťuje možnost změnit chování libovolné metody.

Samotné metody, poskytované touto assembly se spouštějí v nativní podobě. To s sebou nese výhodu v nativní rychlosti zpracování metod v průběhu interpretace. U typů reprezentovaných přímou *instancí* to navíc umožňuje volat metody přímo na reprezentovaném objektu.

Takovéto přímé volání je ale možné pouze pokud jsou i všechny argumenty pro volání dostupné ve formě přímých *instancí*. Už před inicializací assembly tedy potřebujeme znát všechny přímé typy, abychom mohli rozhodnout, které metody budou volané v nativní podobě. Tento fakt znemožňuje provádět změny v *Runtime* za běhu, jako je to možné u jiných assembly. Nicméně výhody, které přenáší nativní zpracování, jsou pro nás mnohem důležitější.

Kromě základní práce s typy má *Runtime* na starost tvorbu podkladů pro zobrazení schématu kompozice. Toto architektonické rozhodnutí je zapříčiněno tím, že základem pro tvorbu podkladů jsou metody poskytované uživatelskými definicemi typů. *Runtime* je tedy může pohodlně využít pro získání potřebných údajů.

### 4.3.5 Abstraktní třída RuntimeTypeDefinition

Každá typová definice, která může být přidána do *Runtime*, musí implementovat abstraktní třídu `RuntimeTypeDefinition`. Tato třída poskytuje framework pro snadnou tvorbu uživatelských rozšíření. Znatelnou změnou proti rozšiřitelnosti předchozí verze je podstatně zjednodušený způsob, jak se dají definovat metody rozšiřujících typů. Podrobnější informace o rozšiřitelnosti budou popsané v kapitole 6.3.

Definované metody jsou následně upraveny tak, aby před jejich vyvoláním docházelo ke správnému rozbalování objektů reprezentovaných v přímých

*instancích*. Stejně tak jsou zabalovány případné návratové hodnoty, aby je bylo možné dále použít v analyzační knihovně.

#### 4.3.5.1 Třída DirectTypeDefinition

Třída `DirectTypeDefinition` je předpřipravená třída, pro definici metod přímých *typových definic*. V takto definovaných metodách sice nemáme možnost ukládat informace do datových položek *instance*, ale máme k dispozici konkrétní objekt ve vlastnosti `DirectTypeDefinition.This`. Díky tomu je definice uživatelských metod mnohem přehlednější, než tomu bývalo v předchozí verzi editoru.

#### 4.3.5.2 Třída DataTypeDefinition

Druhou předpřipravenou třídou uživatelských *typových definic* je `DataTypeDefinition`, kterou používáme pro definici metod datových *instancí*. Najdeme zde služby, které umožní ukládat libovolná data do datových položek *instance*. Dále jsou datové definice typů přizpůsobené pro snadnou tvorbu editací. Tento typ definic je využíván pro reprezentaci komplexnějších typů, nebo typů, u nichž si přejeme vytvářet editace.

#### 4.3.5.3 Třída Array

Zpracování polí je v rámci editoru řešeno trochu odlišným způsobem, než se kterým se můžeme setkat v .NET. Pole totiž nemůžeme snadno reprezentovat v nativní podobě, jako například primitivní typy. Je to způsobené tím, že pole mohou mít různé typy, podle toho, které položky do nich chceme ukládat a také podle toho, kolik dimenzí pole má. Museli bychom tedy vytvářet zvláštní typ pole pro každý podporovaný typ a také pro každou dimenzi, což je ale nepraktické.

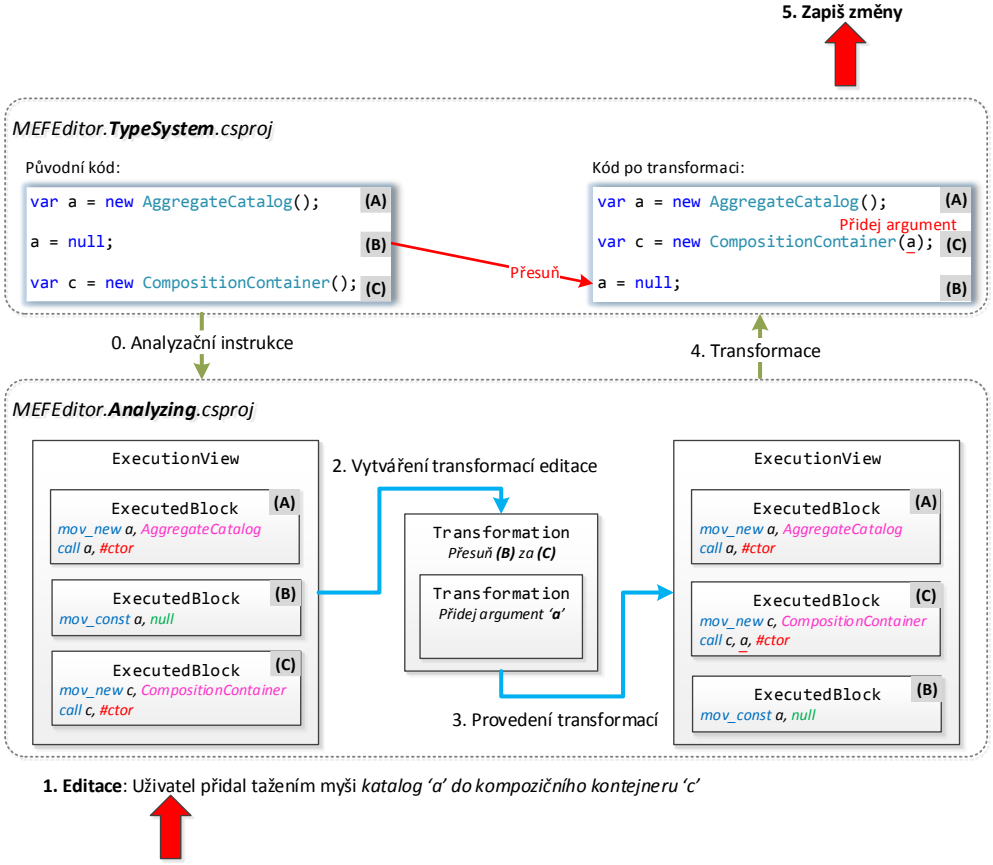
Z tohoto důvodu implementujeme pole jako přímou *instanci* obsahující objekt typu `TypeSystem.Runtime.Array`. Tento objekt může být jednak použit v uživatelských rozšířeních a spolupracovat tak s datovými *instancemi*. Stejně tak ale díky konverzním metodám může být použit jako by se jednalo o přímou *instanci* typu pole použitelnou v nativních voláních.

## 4.4 Editace

***Namespace**: MEFEditor.Analyzing.Editing*

Nabízení editací je důležitou vlastností našeho editoru, proto si jejich implementaci podrobně popíšeme. Zpracování editace začíná jejím vyvoláním, které provede uživatel ve schématu kompozice. V této době tedy již máme k dispozici `ExecutionView`, který je výsledkem analýzy zobrazené kompozice. Editace po svém vyvolání vytvoří patřičné transformace nad `ExecutionView`. Pokud jsou transformace úspěšné, jsou zaslány vyšší vrstvě, kterou je v našem případě typový systém.

Popsaný proces je zachycen na následujícím obrázku 4-7:




*4-7 Průběh zpracování editace přes transformace ExecutionView.*

Na tomto obrázku si všimněme, že editace i transformace pracují pouze s výsledkem analýzy v podobě `ExecutionView`, aniž by potřebovaly znát jazyk *zdrojových instrukcí*. To je zejména nutné, abychom mohli v editoru nabízet uživatelsky přívětivé API pro tvůrce editací v uživatelských rozšířeních. Nemusíme totiž znát syntaktickou stránku editací.

Editací *analyzačních instrukcí* tedy docílíme možnosti nabízet editace jednotně pro různé jazyky *zdrojových instrukcí*. O samotné promítnutí změn do *zdrojových instrukcí* se pak postará mapování transformací na *zdrojové instrukce*, ze kterých jsou *analyzační instrukce* vygenerovány.

Poskytované transformace je potom možné skládat do komplexnějších editací. Takže je snadné spojit například transformaci pro změnu pořadí některých řádků s transformací pro přidání argumentu do volání metody na objektu. Tak získáme editaci, která například přidá katalog do `CompositionContainer`, jak bylo uvedeno na obrázku 4-7.

## Druhy editací

Rozlišujeme několik druhů editací, v závislosti na kontextu, ve kterém mají platnost. Základním kontextem je platnost pro *instanci*. Takové editace jsou nabízeny v rámci editoru, kdykoliv je *instance* uživateli zobrazena. Druhým typem jsou připojené editace, které jsou do *instance* připojovány jinou *instancí*. Typickým příkladem použití je nabízení editace na vyjmutí komponenty z kontejneru. Tuto editaci typicky připojuje kontejner ke komponentě. Platnost připojených editací je

omezena pouze na kontext kontejneru, což znamená, že editace je nabídnuta, pouze pokud bude komponenta v kontejneru opravdu zobrazená.

Posledním typem editací jsou globální editace, které se nevztahují k *instancím*, ale ke *composition pointu*. Typickým příkladem globální editace je vytvoření nového objektu.

### 4.4.1 Koncept pohledů a transformací

V rámci našeho editoru definujeme třídu `ExecutionView` jako pohled na zaznamenaný řetězec objektů třídy `ExecutedBlock`, který získáme jako výsledek interpretace *analyzačních instrukcí*. Transformacemi pak budeme rozumět operace nad tímto pohledem.

Transformace jsou poskytované přes editační informace v *analyzačních instrukcích*, jsou tedy specializované podle konkrétního jazyka ze kterého *analyzační instrukce* pocházejí. Použití transformace na pohled pak spočívá ve vyvolání základních operací podporovaných pohledem, tak aby se v rámci pohledu patřičně změnila struktura řetězce objektů třídy `ExecutedBlock`.

Tímto bychom však nedocílily žádných změn ve *zdrojových instrukcích*. Proto si transformace do pohledu navíc poznamená informace o změnách, které mají být ve *zdrojových instrukcích* provedeny. Tyto změny jsou použity při volání `ExecutionView.Commit`, kdy dojde k jejich zapsání.

Podotkněme, že koncept pohledů umožňuje vyzkoušet, zda je nějakou editaci možné provést, ještě před samotným zapsáním změn do *zdrojových instrukcí*. Díky tomu můžeme například efektivně zjistit, kterou proměnnou je vhodné využít pro vytvoření volání nějaké metody, v případech kdy je jedna *instance* dostupná ve více různých proměnných. K tomu také pomáhá možnost zneplatnit pohled voláním `ExecutionView.Abort`, ve kterém může transformace uvést důvod, proč nemůže být provedena. Toto chybové hlášení editor zobrazuje uživateli.

### 4.4.2 Poskytovatelé transformací

Každá transformace, kterou editor používá, musí být vytvořena některým poskytovatelem transformací, kterého do *analyzačních instrukcí* vloží parser nebo překladač. Rozlišujeme následující druhy poskytovatelů:

- **BlockTransformProvider** – Poskytuje transformaci pro práci s celými bloky instrukcí (které obvykle odpovídají jednomu řádku zdrojového kódu). Umožňuje bloky prohazovat mezi sebou a také přidávat nová volání.
- **CallTransformProvider** – Nabízí transformace pro práci s voláními, jako jsou změny, mazání nebo přidávání argumentů. Také dovoluje nastavit příznaky argumentů, které ovlivní, zda smazání argumentu vynutí i smazání celého volání či nikoliv.
- **RemoveTransformProvider** – Poskytovatel transformací, které dokáží vymazat volání, argument nebo celý blok instrukcí, v závislosti na tom z jakého kontextu jsme poskytovatele získali.

Princip poskytovatelů transformací je výhodný v tom, že dochází k vytváření transformací až v době, kdy je opravdu potřebujeme. Není tedy nutné v průběhu překladu do *analyzačních instrukcí* vytvářet všechny možné transformace. To se projevuje vyšší rychlostí překladu metod.

### 4.4.3 Editace na odstranění instance

Většina editací je vytvářena z transformací *typovými definicemi*, neboť editace, které *instance* nabízí, obvykle záleží na jejím typu. Odstranění *instance* je však odlišné, neboť tuto editaci můžeme požadovat po libovolné *instanci*, nehledě na jejím typu.

Odstraňování *instancí* zajišťuje třída `InstanceRemoveProvider`. *Instanci* odstraňuje tak, že vyhledáme místo, kde byla *instance* vytvořena. V metodě, kde k vytvoření došlo, vyhledáme všechny instrukce, ve kterých byla *instance* přítomna. Tyto instrukce pak jednotlivě odstraníme pomocí transformací, které jsou v nich uloženy. Pokud v popsaném algoritmu narazíme na instrukci, kterou není možné odstranit, editaci nelze provést.

Editor tak může nabízet editaci pro odstranění každé položky zobrazené ve schématu kompozice. Nutnou podmínkou však je dostupnost potřebných transformací v *analyzačních instrukcích*.

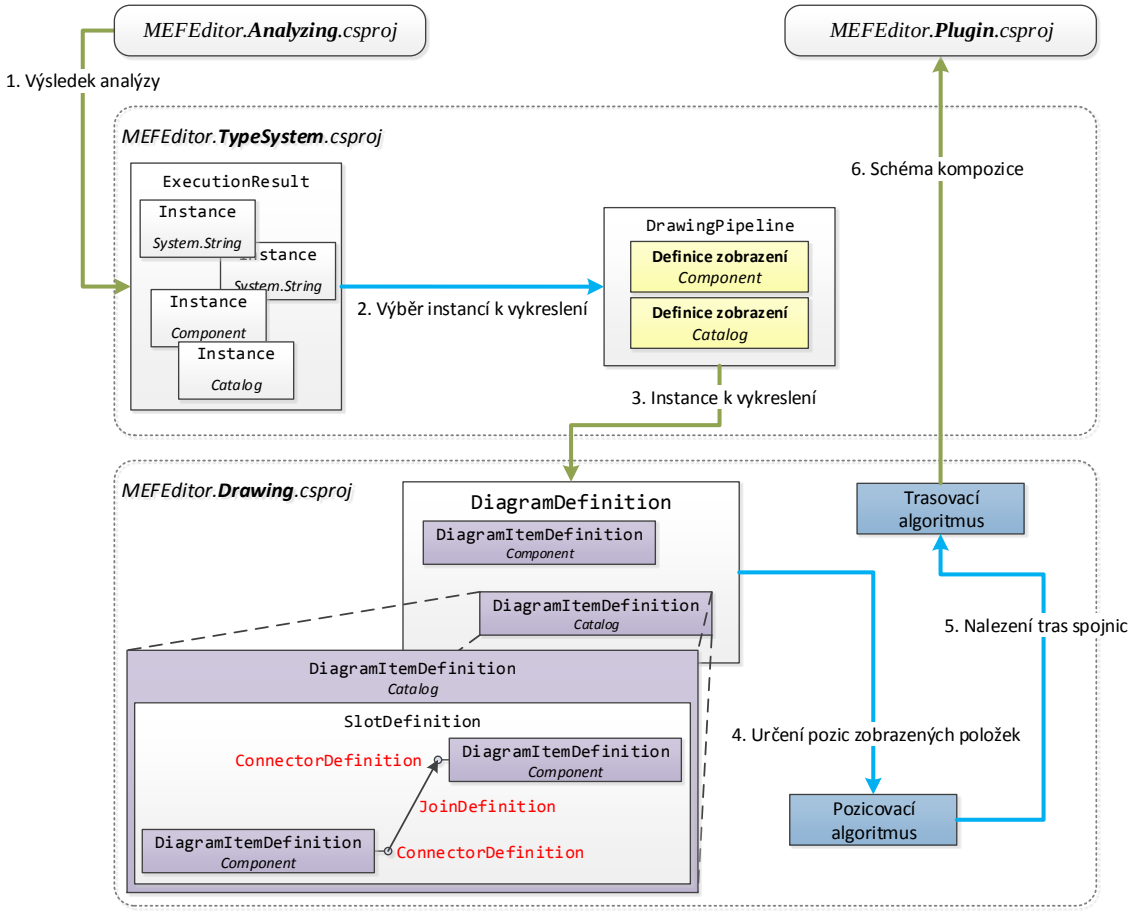
## 4.5 Vykreslování schématu kompozice

***Namespace**: MEFEditor.Drawing*

Proces vykreslování je zahájen vybráním nějakého *composition pointu* v uživatelském rozhraní. To způsobí, že se okamžitě začne interpretovat metoda, která *composition pointu* přísluší. Po dokončení interpretace dostaneme výsledek analýzy, který mimo jiné obsahuje seznam všech vytvořených *instancí*. Z nich musíme vybrat ty, které budeme uživateli zobrazovat ve schématu kompozice. Výběr provedeme na základě typu, dle kterého zjistíme, zda máme definici zobrazení, získanou z uživatelských rozšíření, umožňující *instanci* reprezentující objekt daného typu vykreslit.

To by však nebylo dostatečné pro vykreslování komponent, které obvykle chceme zobrazit nezávisle na jejich typu. Proto *instance*, které jsou komponentami a nemají explicitně určenou definici zobrazení, vykreslíme pomocí obecného vykreslovacího rozšíření.

Než je možné výsledek zobrazit uživateli, je však nutné nejprve upravit pozice zobrazených položek a také nalézt neprotínající trasy pro spojnice. Popsaný proces je znázorněn na následujícím obrázku 4-8:




*4-8 Proces vykreslení schématu kompozice z výsledku analýzy composition pointu.*

Vykreslení *instance* v předchozí verzi editoru vycházelo přímo ze zkoumání stavu této *instance*. To s sebou však neslo problémy, které souvisely s přílišnou provázaností interní implementace definic typů s vykreslovacími rozšířeními. Editor kvůli tomu také nebyl schopen řešit případy s kruhovou závislostí mezi zobrazenými *instancemi*.

Z těchto důvodů je současná implementace navržena mnohem deklarativněji. Z *instancí*, které chceme vykreslit, nejdříve necháme vytvořit objekty třídy `DiagramItemDefinition`, které jsou poskytovány *typovou definicí* každé *instance*. Ve vytvořené definici zobrazení jsou uchovány informace, sloužící jako podklady pro práci vykreslovacího rozšíření. Tento přístup znamená, že schéma kompozice dokážeme vytvořit v implementačně nezávislé formě, ve které například vyřešíme kruhové závislosti, a až na jejím základě sestrojujeme samotné grafické reprezentace *instancí*.

### 4.5.1 Třída DiagramItemDefinition

Základní jednotkou zobrazovanou ve schématu kompozice jsou obrazy *instancí* získaných z analýzy *composition pointu*. Tyto obrazy získáváme od uživatelsky definovaných rozšíření typu `ContentDrawing`, které pro svoji práci potřebují

reprezentace *instancí* v podobě `DiagramItemDefinition`. Tím se efektivně vyhneme nutnosti zkoumat samotnou *instanci* na úrovni interní implementace jejího stavu. Objekty `DiagramItemDefinition` získáváme voláním `RuntimeTypeDefinition.Draw`, které definují uživatelské *typové definice*.

Další výhodou, kterou získáváme použitím `DiagramItemDefinition`, je možnost ovlivnit obsažené informace i z jiných *instancí* než je ta vykreslovaná. Typickým případem může být označení komponovaných *instancí* v `CompositionContainer` příznakem, zda se *instanci* povedlo zkomponovat či nikoliv. Takové příznaky samozřejmě mohou být využity při vykreslování.

Již jsme si popsali principiální použití definic zobrazení, nyní se podrobněji podívejme na informace, které musí být v definicích zobrazení dostupné, abychom mohli úspěšně vykreslit *instanci*. Základním údajem, který bude uživatel používat, bude jistě typ zobrazené *instance*, díky čemuž lze lépe nahlédnout, které komponenty se kompozice zúčastňují. Pro komponenty bude také důležitá schopnost zobrazit konektory pro naznačení importů a exportů. Z tohoto důvodu i *definice zobrazení* musí připojování konektorů podporovat. Posledním významným typem zobrazovaných *instancí* jsou kontejnery, ve kterých se typicky můžou objevovat další vnořené *instance*. Aby náš framework postihl i možnost, kdy jsou vnořené *instance* uvnitř kontejneru uspořádány do nějakých skupin, můžeme tyto *instance* uspořádávat do takzvaných slotů, které je možné vykreslit pomocí `SlotCanvas`.

K těmto obecným informacím ještě musíme přidat údaje specifické pro konkrétní *instanci*. Pro `DirectoryCatalog` nás totiž může zajímat cesta, kde hledáme rozšiřující assembly, takovýto údaj by ale neměl smysl například pro `CompositionContainer`. Z tohoto důvodu přidávají *typové definice* jednotlivých *instancí* speciální vlastnosti, které mohou být následně využity při vykreslení.

### 4.5.2 Třída ContentDrawing

Po zkonstruování `DiagramItemDefinition` všech *instancí* určených k zobrazení můžeme přejít k vytvoření obrazu reprezentované *instance*. Tuto funkcionalitu obstarávají uživatelská rozšíření typu `ContentDrawing`, která představují WPF[3] objekty zobrazované uživateli. Tento obsah je v konečném zobrazení umístěn uvnitř objektu třídy `DiagramItem`, která přidává funkčnost na interakci s uživatelem a dalšími zobrazenými *instancemi*.

Způsob jakým se typy `ContentDrawing` nahrávají do editoru, bude uveden v kapitolách zabývajících se uživatelskými rozšířeními 6.4. Zde také najdeme manuál, jak lze takové rozšíření snadno vytvořit.

### 4.5.3 Třída DiagramItem

Abychom ušetřili uživatele definujícího vzhled nějaké *instance* od nutnosti implementovat také rutiny pro interakci, jsou v knihovně `MEFEditor.Drawing` tyto části oddělené. Vzhled je určen objektem typu `ContentDrawing`, kdežto chování a reakce na uživatelské vstupy definuje `DiagramItem`.

V rámci editoru rozlišujeme následující uživatelské vstupy týkající se *instancí*:

- **Změna pozice** – Systémem drag&drop může uživatel změnit umístění libovolné *instance* ve schématu kompozice. Přesun *instance* je

[3] Framework Windows Presentation Foundation [24] slouží pro implementaci grafických rozhraní aplikací v .NET.

implementován v rámci `DiagramItem`. Po provedení přesunu je navíc nutné upravit schéma, aby vyhovovalo požadovanému formátu, který bude popsán v kapitole 4.5.5. Aby byla pozice zachovaná v rámci několika spuštění *composition pointu*, využívá `DiagramItem` pro počáteční pozici persistentní data založená na identifikátoru *instance*.

- **Přesun mezi katalogy** – Zobrazené *instance* lze také pomocí drag&drop přesunovat mezi různými kontejnery. Takovýto přesun se již projeví změnou provedenou ve *zdrojových instrukcích*. Pokud by však nebylo možné přesun provést, je uživatel patřičně upozorněn kontextovou nápovědou, kterou zobrazuje `DiagramItem`.
- **Vyvolání explicitní editace** – Posledním uživatelským vstupem, který musí `DiagramItem` řešit je vyvolání explicitní editace z kontextové nabídky pro každou *instanci*. Explicitní editace se obvykle týkají změn vlastností zobrazených *instancí*, jako je třeba zdrojová složka pro `DirectoryCatalog`.

Dále `DiagramItem` poskytuje služby, které může `ContentDrawing` využít pro vykreslování. Nejdůležitější takovou službou je `DiagramItem.FillSlot`, která umožňuje zobrazit *instance* uvnitř jiné. V `DiagramItem` jsou také obsažené případné konektory definované komponentami, takže je možné mezi jednotlivými `DiagramItem` objekty vytvořit spojnice. Toho s výhodou využijeme pro zobrazení vztahů mezi komponentami, které určuje `CompositionContainer`.

### 4.5.4 Třída SlotCanvas

Dosud jsme popisovali zobrazení reprezentace samostatné *instance*. V rámci schématu kompozice ale často potřebujeme zobrazit *instance* vnořené do *instance* reprezentující nějaký kontejner. K těmto účelům slouží `SlotCanvas`, který zpřístupňuje služby pro rozvržení a interakci se zobrazenými *instancemi* poskytované knihovnou `MEFEditor.Drawing`.

Obvyklým způsobem použití je vytvoření `SlotCanvas` uvnitř nějakého objektu `ContentDrawing`. Vytvořený slot je poté možné naplnit voláním `DiagramItem.FillSlot`, které zobrazí *instance* definované pomocí `SlotDefinition`. Takto naplněný slot dokáže vyvolávat editace v závislosti na uživatelské interakci a také zajišťuje uspořádání zobrazených *instancí* stejným způsobem jako ve zbytku schématu kompozice. Díky tomu uživatel implementující *definici zobrazení* nemusí řešit způsob zobrazení obsažených *instancí*.

### 4.5.5 Algoritmus uspořádání schématu kompozice

V předchozí verzi editoru byly *instance* ve výchozí pozici uspořádávány zhruba do čtvercových ploch, aby výsledné schéma kompozice nebylo příliš vysoké ani široké. Tento algoritmus použijeme také v současné verzi editoru. Nicméně předchozí verzi vylepšíme o zamezení překrývání zobrazených *instancí* a také o komplexnější algoritmus pro určování tras spojnic. Díky tomu bude výsledné schéma kompozice přehlednější, než u předchozí verze. Pro případ, že by uživatel preferoval původní způsob zobrazení, dovolí editor tyto algoritmy vypnout v nastavení zobrazení.

#### 4.5.5.1 Překrývání instancí

V předchozí verzi editoru bylo možné umístit *instance* ve schématu kompozice tak, že se vzájemně překrývaly. To mohlo vést až k tomu, že došlo k úplnému zakrytí některé z *instancí* a tím pádem uživatel nemohl vidět kompletní schéma kompozice. Z tohoto důvodu budeme v současné verzi editoru zajišťovat, aby mezi zobrazenými *instancemi* byl dostatečný odstup.

Třídou, která řeší vzájemnou pozici *instancí* je `ItemCollisionRepairer`. Ta dostane na svém vstupu požadované pozice všech zobrazených *instancí* a jejich vzájemným posunováním docílí toho, aby se žádné dvě *instance* nepřekrývaly. Při tomto posunování se algoritmus snaží posunovat pouze ty *instance*, jejichž pozice jsou „nejstarší“. Tím se vyhneme tomu, aby algoritmus odsunul pryč *instanci*, jejíž pozici zrovna určil uživatel.

Algoritmus také při posunech zohledňuje jejich velikost. Aby nedocházelo k velkým změnám ve schématu kompozice, vybírá pro posunutí *instance* takový směr, ve kterém bude posun co možná nejkratší.

#### 4.5.5.2 Výpočet trasy spojnic

Abychom omezili výskyt spojnic protínajících zobrazené *instance* ve schématu kompozice, musíme implementovat algoritmus, který bude vyhledávat vhodnější spojnice, než jsou přímky mezi importy a exporty. Tento algoritmus je implementován ve třídě `JoinGraph`, která představuje graf spojnic a vrcholů zobrazených *instancí*.

Jak jsme zjistili v analýze, v rámci kapitoly 2.9.2, můžeme mezi každou dvojicí *instancí* na stejné úrovni nalézt jejich neprotínající spojnici. Toto tvrzení vychází z předpokladu, že se zobrazované *instance* nepřekrývají. Nyní si popíšeme algoritmus, který dokáže takové neprotínající spojnice nalézt. Poznamenejme, že je vyvíjen převážně s ohledem na jednoduchost implementace, neboť se nejedná o stěžejní část práce.

Algoritmus funguje ve dvou fázích. V první fázi objevujeme různé neprotínající cesty mezi každou dvojicí *instancí*, která má být propojena. Tím vznikne graf, ve kterém ve druhé fázi již snadno najdeme pro každou dvojici nejkratší cestu.

##### Objevování cest v grafu

Na začátku první fáze máme úkol, nalézt v grafu, obsahujícím všechny zobrazené *instance*, spojnici mezi dvojicí zadaných *instancí*. Každou *instanci* v grafu reprezentujeme jako množinu bodů. Tyto body jsou pospojované tak, jak je možné *instanci* spojnicí „obcházet“. Toto je možné právě díky zaručené nenulové vzdálenosti mezi dvojicemi *instancí*.

Vyhledávání začne testem, zda je možné mezi *instancemi* vést přímku, pokud se ukáže, že v cestě leží jiná *instance*, je přidána do grafu cesta k této *instanci*. Vyhledávání pak pokračuje z jejích bodů. Pro zjišťování zda je mezi *instancemi* překážka slouží třída `SceneNavigator`.

Protože testování na přítomnost překážek je výpočetně náročná operace, omezíme počet potřebných testů. U každého bodu si budeme udržovat informaci o jeho *výhledu*. *Výhled* bodu zde definujeme jako část roviny, ve které neleží jemu příslušná *instance*. Spojnice pak povolíme jen mezi body, které se mají vzájemně ve *výhledu*. Tím omezíme test na přítomnost překážek například mezi odvrácenými body na dvojici *instancí*.

### Vyhledání nejkratší spojnice

S využitím grafu, který jsme vytvořili v první fázi je vyhledání spojnice již snadné. Jedná se o tradiční grafovou úlohu, hledání nejkratší cesty mezi dvěma vrcholy. Jelikož se jedná o graf s hranami ohodnocenými nezápornými délkami, využijeme implementačně jednoduchý Dijkstrův algoritmus [10].

Algoritmus končí nalezením nejkratší cesty. Tato cesta je následně využita pro zobrazení spojnice ve schématu kompozice.

## 4.6 Testovací framework

***Namespace**: MEFEditor.UnitTesting*

V kapitole 2.10 jsme zjistili, které části editoru je nutné testovat. Z tohoto důvodu jsme naimplementovali framework, který nám toto testování umožní. Veškeré unit testy, které zahrnují i testování *doporučených rozšíření*, jsou implementovány v projektu *UnitTesting* v příloze [D]. Implementované testy je možné automaticky spouštět přes standardní testovací nástroje *Visual Studia*.

V této kapitole si popíšeme, jak funguje testování nejdůležitějších částí editoru. V projektu *UnitTesting* je však možné nalézt další testy, které testují implementaci editoru a *doporučených rozšíření*.

### 4.6.1 Interpretace analyzačních instrukcí

Pro testování interpretace potřebujeme spouštět zadané instrukce a testovat jejich projevy v běhovém prostředí virtuálního stroje. Pro tyto účely slouží metoda `ExecutionUtils.Run`. Předáme ji metodu, která vygeneruje požadované instrukce. Tyto instrukce jsou následně spuštěny a jejich výsledek můžeme kontrolovat například pomocí metody `AssertVariable`. Jejich použití je zřejmé z následujícího testu:

```
[TestMethod]
public void SimpleAssign()
{
  //spustíme interpretaci testovacích instrukcí
  ExecutionUtils.Run((e) =>
  {
    //vygenerujeme instrukci pro přiřazení literálu
    e.AssignLiteral("var1", "TestValue");
  })
  //otestujeme zda proměnná obsahuje správnou hodnotu
  .AssertVariable("var1").HasValue("TestValue");
}
```

*4-9 Ukázka jednoduchého testu, kde testujeme vygenerované instrukce*

Všechny implementované testy analyzačních instrukcí je možné nalézt ve třídě `Machine_Testing`.

### 4.6.2 Překladače zdrojových instrukcí

Testování překladačů se podobá testování *analyzačních instrukcí*. Nejprve však musíme spustit překlad ze zdrojových instrukcí. Na *analyzačních instrukcích* získaných překladem pak spustíme interpretaci a výsledek budeme kontrolovat stejným způsobem jako u testů *analyzačních instrukcí*.

V našem testovacím frameworku je implementováno testování překladačů C# a CIL z *doporučených rozšíření*. Ke spuštění opět použijeme metodu `Run`. Příklad použití si opět ukážeme na následujícím obrázku:

```
[TestMethod]
public void Compile_SimpleAssign()
{
  //spustíme překlad zdrojového kódu
  //C#, po překladu začne interpretace
  AssemblyUtils.Run(@"
    var test=""TestValue"";
  ")
  //ověříme správnost hodnoty v proměnné test
  .AssertVariable("test").HasValue("TestValue");
}
```

*4-10 Ukázka psaní testů pro překladač jazyka C#*

Testy pro překladač jazyka C# jsou ve třídě `Compiler_CSharp_Testing` a testy pro překlad CIL nalezneme ve třídě `Compiler_CIL_Testing`.

### 4.6.3 Typové definice

Testování *typových definic* je od předchozích případů odlišné. *Typové definice* musíme nejprve nahrát do assembly *Runtime*. Následně potřebujeme vygenerovat instrukce, které budou *typovou definici* používat. Použití *typové definice* bude nejsnazší v testovacích zdrojových kódech. Testování tedy bude probíhat voláním metod *typových definic* v testovacím kódu.

Nahrání *typové definice* do *Runtime* provedeme pomocí metod `AddToRuntime` pro datovou *typovou definici* nebo `AddDirectToRuntime` pro přímou *typovou definici*. Jak se používají, můžeme vidět na následujícím zdrojovém kódu:

```
[TestMethod]
public void RuntimeType_DirectClassType()
{
  //nejprve definujeme kód, který použije
  //testované funkce typové definice pro StringBuilder
  AssemblyUtils.Run(@"
    var test=new System.Text.StringBuilder();
    test.Append(""Test"");
    test.Append(""Value"");

    var result=test.ToString();
  ")
  //pak přidáme přímou typovou definic do Runtime
  .AddDirectToRuntime<StringBuilder>()
  //nakonec otestujeme, zda volání metod vrací správný výsledek
  .AssertVariable("result").HasValue("TestValue");
}
```

*4-11 Ukázkové využití metod pro přidávání typových definic do Runtime a jejich testování*

Testy *typových definic* nalezneme ve třídě `TypeDefinitions_Testing`.

### 4.6.4 Editace ve zdrojových kódech

Testování editací vychází z testování použití *typových definic*, neboť editace definují právě ony. Opět tedy musíme nahrávat *typové definice* do *Runtime*. Poté spustíme překlad a interpretaci testovacího zdrojového kódu.

Abychom mohli spustit testovanou editaci, musíme mít možnost simulovat uživatelské akce. K tomuto účelu slouží metoda `RunEditAction`. Po spuštění editace provedeme porovnání zdrojového kódu s očekávaným výstupem. Editaci však musíme mít možnost vytvořit. To provedeme v rámci metody přidané do *Runtime* pomocí volání `AddMethod`.

Použití těchto metod lépe pochopíme z následující ukázky:

```
[TestMethod]
public void Edit_SimpleReject()
{
  //test začneme vytvořením kódu
  //na kterém chceme spustit editaci
  AssemblyUtils.Run(@"
    var arg=""input"";
    DirectMethod(arg);
  ")
  //do Runtime přidáme metodu, která
  //vytvoří editaci pro vyjmutí argumentu
  .AddMethod("Test.DirectMethod", (c) =>
  {
    //argument, kterému chceme vytvořit editaci
    var arg = c.CurrentArguments[1];
    //přidáme mu editaci na vyjmutí
    c.Edits.RemoveArgument(arg, 1, ".reject");
    //argument nastavíme jako nepoviný
    //aby nedošlo k odstranění celého volání
    c.Edits.SetOptional(1);
  }, Method.Void_StringParam)

  //spustíme editaci na instanci v proměnné arg
  .RunEditAction("arg", ".reject")
  //nakonec porovnáme výsledek editace se vzorem
  .AssertSourceEquivalence(@"
    var arg=""input"";
    DirectMethod();
  ");
}
```

*4-12 Ukázka testovatelnosti editací. V ukázce editaci vytvoříme, spustíme a následně otestujeme její projev ve zdrojovém kódu.*

Testy editací nalezneme ve třídě `Edits_Testing`.

### 4.6.5 Vykreslování schématu kompozice

Testování vykreslování je zaměřené na implementaci třídy `SceneNavigator`. Jejím hlavním úkolem je zjišťování průsečíků v zobrazené scéně na zadané úsečce.

Testování vykreslování začneme použitím `DrawingTest.Create` a následným přidáním zobrazovaných položek do scény metodou `Item`. Pomocí té určíme jméno zobrazované položky a její pozici na souřadnicích x, y. Vytvořenou scénu pak můžeme testovat na průsečíky se zadanou úsečkou s využitím metody

`AssertIntersection`. Tato metoda otestuje, zda první protnutí úsečky nastane s položkou zadaného jména. Ukázkový test můžeme vidět na následujícím obrázku:

```
[TestMethod]
public void Scene_ItemIntersection()
{
  //nejprve vytvoříme testovanou scénu
  DrawingTest.Create
    //přidáme do ní položky 100x100px
    //na zadané pozice
    .Item("A", 0, 0)
    .Item("B", 500, 0)
    .Item("C", 1000, 0)

    //otestujeme, zda zadaná úsečka protne
    //položku B
    .AssertIntersection(
      //bod uvnitř A
      new Point(50, 50),
      //bod uvnitř C
      new Point(1050, 50),
      "B"
    );
}
```

*4-13 Příklad testování výpočtu průsečíku v zadané scéně.*

Testy vykreslování nalezneme ve třídě `SceneNavigator_Testing`.

# 5 Uživatelská příručka

V následujících kapitolách si názorně ukážeme použití našeho editoru při vývoji ukázkových projektů. Nejdříve ale musíme editor spolu s *doporučenými rozšířeními* nainstalovat do *Visual Studia*. Jednotlivé kroky instalace jsou popsány v kapitole 5.1.

Dále se v kapitole 5.2 seznámíme s uživatelským rozhraním editoru a možnostmi jeho nastavení. Nakonec si ukážeme, jak se editor používá na ukázkových projektech při vývoji kompozičního algoritmu v kapitole 5.3.1 a při vývoji v konfiguraci REA v kapitole 5.3.2.

## 5.1 Instalace a spuštění

Po spuštění souboru *Enhanced_MEF_Component_Architecture_Editor.vsix* z přílohy [E] se zobrazí dialogové okno, které nás provede přidáním editoru do prostředí *Microsoft Visual Studia 2010* nebo *Microsoft Visual Studia 2012*, z nichž alespoň jedno musí být na cílovém počítači nainstalované a to ve verzi *Professional* nebo vyšší.

Po provedení instalace spustíme *Visual Studio* a položkou v menu: *View > Other Windows >Enhanced MEF Component Architecture Editor* spustíme editor. Při jeho prvním spuštění je vytvořena *složka rozšíření* editoru.

Cesta složky rozšíření pro *Microsoft Visual Studio 2010*:

```
Dokumenty/Visual Studio 2010/Enhanced MEF Component Architecture Editor/
```

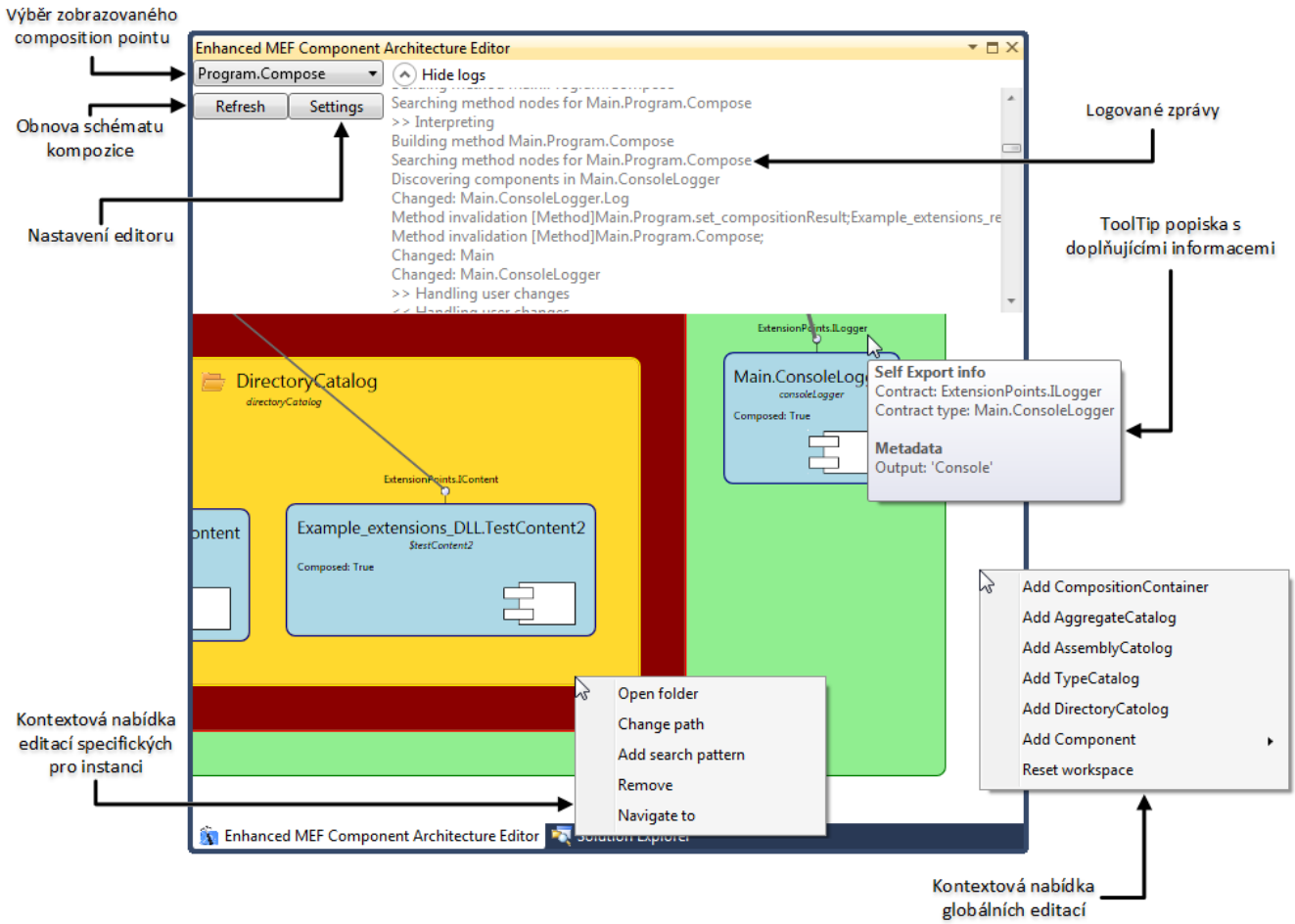
Cesta složky rozšíření pro *Microsoft Visual Studio 2012*:

```
Dokumenty/Visual Studio 2012/Enhanced MEF Component Architecture Editor/
```

Do *složky rozšíření* jsou následně nahrány knihovny *doporučených rozšíření*, které umožní využít editor v projektech napsaných jazykem C# a v projektech vyžadujících analýzu zkompilovaných knihoven v jazyce CIL. Pokud nechceme, aby byla *doporučená rozšíření* v editoru dostupná, stačí tyto knihovny smazat.

Pro přidání uživatelských rozšíření zkopírujeme knihovny, ve kterých jsou implementována, do výše uvedené složky. Rozšíření budou nahrána do editoru při příštím spuštění *Visual Studia*. Seznam nahraných rozšíření, případně chyby objevené při jejich nahrávání jsou zobrazovány v logovaných zprávách.

## 5.2 Uživatelské rozhraní




*5-1 Popis pracovní plochy uživatelského rozhraní*

Editor zobrazuje schéma kompozice na základě právě vybraného *composition pointu*. Po vybrání se zobrazí patřičné schéma kompozice nebo bude vypsána chyba, ke které došlo v průběhu analýzy, 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.

### Spouštění editací

V zobrazeném schématu kompozice jsou editace prováděny pomocí kontextových nabídek zobrazených *instancí*. Dalším způsobem jak lze editace provádět je drag&drop akce, kdy požadovanou *instanci* zkusíme přesunout do prostoru nějakého katalogu, kontejneru nebo na volnou plochu schématu kompozice. Pokud je editace možná, dojde po drop akci k zapsání změn do zdrojového kódu.

Editace, které nejsou vázané na konkrétní *instanci*, se zobrazují v kontextové nabídce volné plochy schématu kompozice.

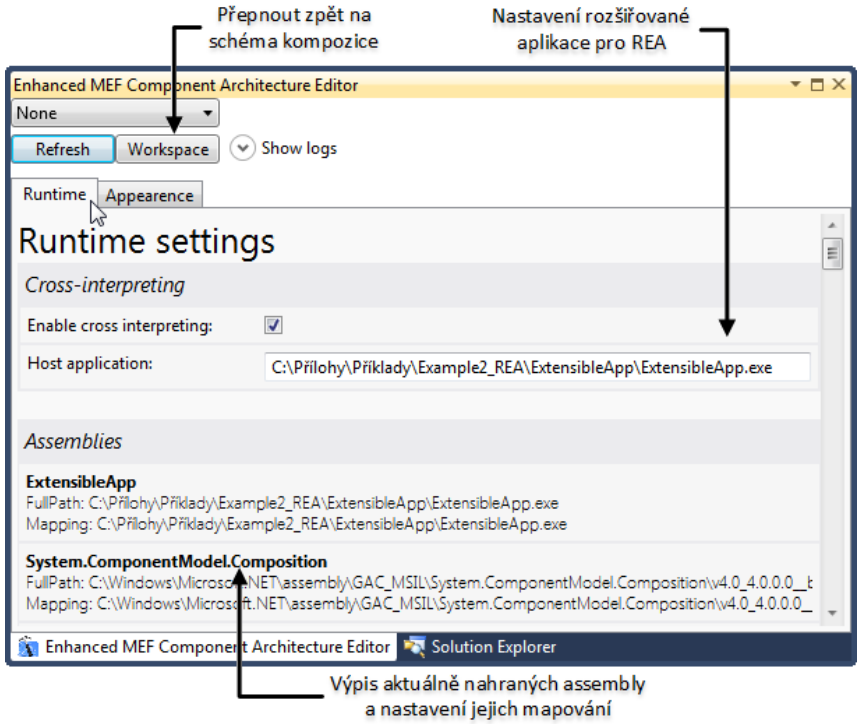
### Změna zobrazení schématu kompozice

Schéma kompozice může uživatel přizpůsobovat pro lepší přehlednost. Je možné měnit přiblížení schématu kompozice pomocí rolování kolečka myši. Uživatel také může tažením zobrazených položek měnit jejich rozmístění. Schéma kompozice je také možné libovolně posunovat, tažením za volnou plochu schématu kompozice.

### Logování zpráv

V průběhu analýzy editoru vzniká množství zpráv, které mohou objasnit případné nečekané chování editoru. Může se jednat například o chybějící implementace metod, 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.

### Nastavení interpretačního prostředí

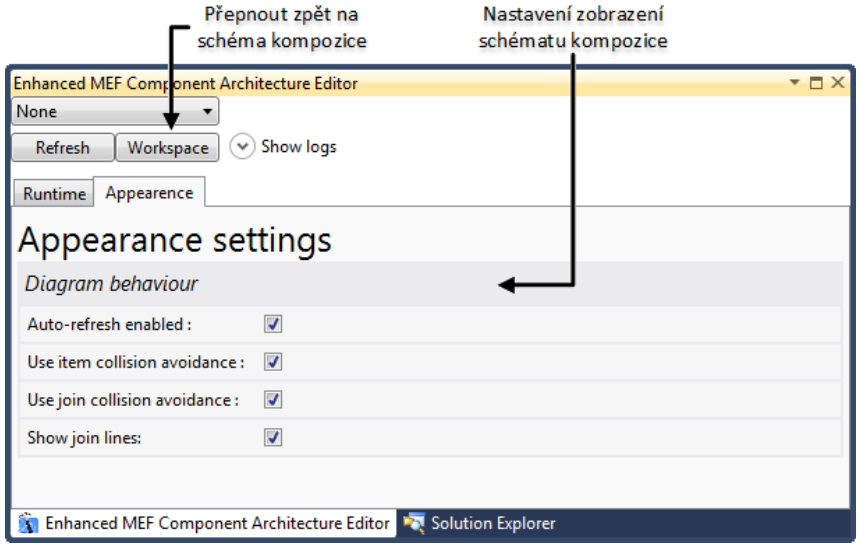



*5-2 Uživatelské rozhraní pro nastavení interpretačního prostředí.*

V záložce *Runtime* je možné nastavit parametry důležité pro analýzu vývojové konfigurace typu REA popsané v kapitole 1.3. Mapování assembly však můžeme využít i ve zdrojových kódech, kde umožní katalogům načítání assembly ve zdrojových kódech, jako by se jednalo o zkompilovanou assembly.

Využití nastavení interpretačního prostředí je předvedeno v kapitole 5.3.2.

**Nastavení vzhledu schématu kompozice**




*5-3 Uživatelské rozhraní pro nastavení vzhledu schématu kompozice.*

Vykreslování schématu kompozice využívá pro uspořádání zobrazovaných položek a spojnic mezi nimi algoritmy, které usnadňují čitelnost výsledného schématu. V rozsáhlých schématech kompozice však může být výpočet těchto algoritmů časově náročný, proto má uživatel možnost je vypnout.

V případech, kdy je na schématu příliš mnoho spojnic a nejsou pro uživatele důležité, může jejich zobrazení vypnout. Také je možné vypnout automatické překreslování schématu kompozice při změnách zdrojových kódu, nebo sledovaných assembly.

## 5.3 Použití editoru

Editor nainstalovaný do *Visual Studia*, podle kapitoly 5.1, můžeme použít při vývoji komponentových aplikací. Jeho použití si předvedeme na dvou ukázkových projektech.

V prvním projektu, ukázaném v kapitole 5.3.1, provedeme uživatele vývojem celého kompozičního algoritmu. Ukážeme si, že téměř všechny kroky sestavování kompozice dokáže uživatel provést z vizuální reprezentace schématu kompozice zobrazované editorem. Také si předvedeme, že nás editor dokáže upozornit na chyby, které by mohly v průběhu kompozice vzniknout.

Abychom si ukázali i nové možnosti analýzy editoru, předvedeme si jeho použití v konfiguraci REA. V příkladu v kapitole 5.3.2 nám poslouží jako nástroj pro ulehčení vývoje ukázkového rozšíření pro rozšiřovanou aplikaci.

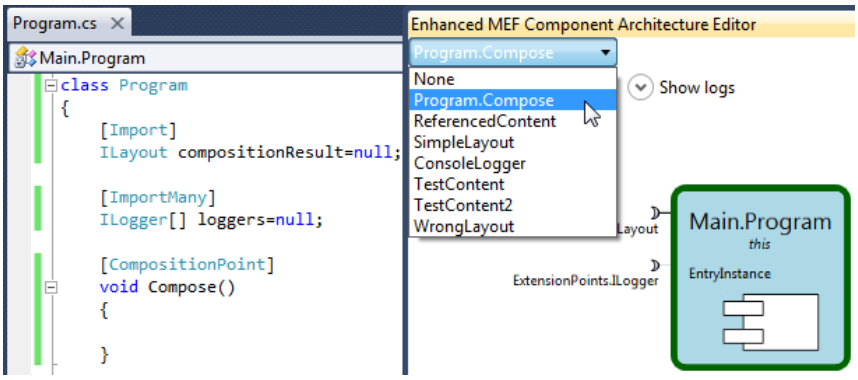
### 5.3.1 Použití editoru při vývoji kompozičního algoritmu

Předpokládejme, že máme spuštěný editor a otevřené solution *Example1_CompositionAlgorithm.sln* z přílohy [G]. 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*.

### Vytvoření composition pointu

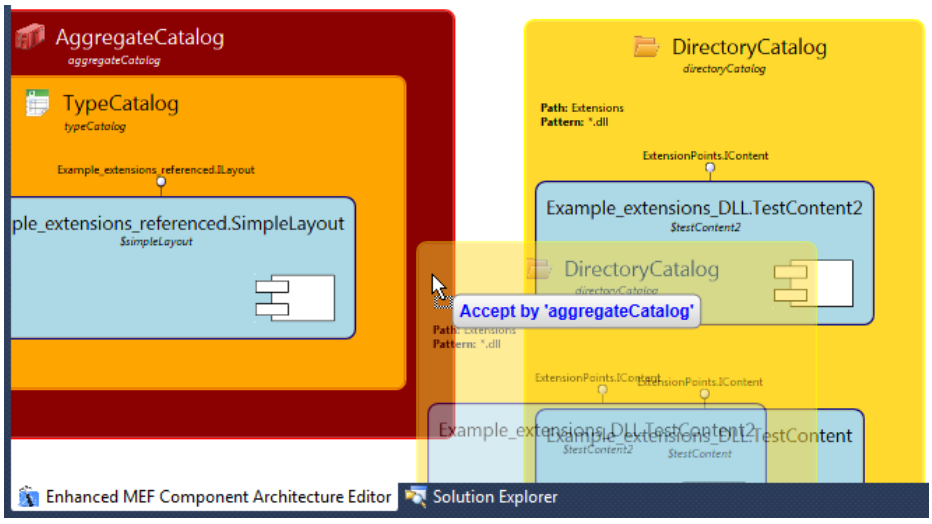
Kompozici chceme provést v metodě `Program.Compose`, označme ji tedy atributem `CompositionPoint`. V seznamu pro výběr *composition pointů* se nám objeví právě přidaná metoda. Jejím zvolením se nám zobrazí informace o komponentě `Program`.




*5-4 Takto vypadá prostředí Visual Studia po označení metody atributem CompositionPoint a jejím vybrání v editoru.*

### Příprava katalogů

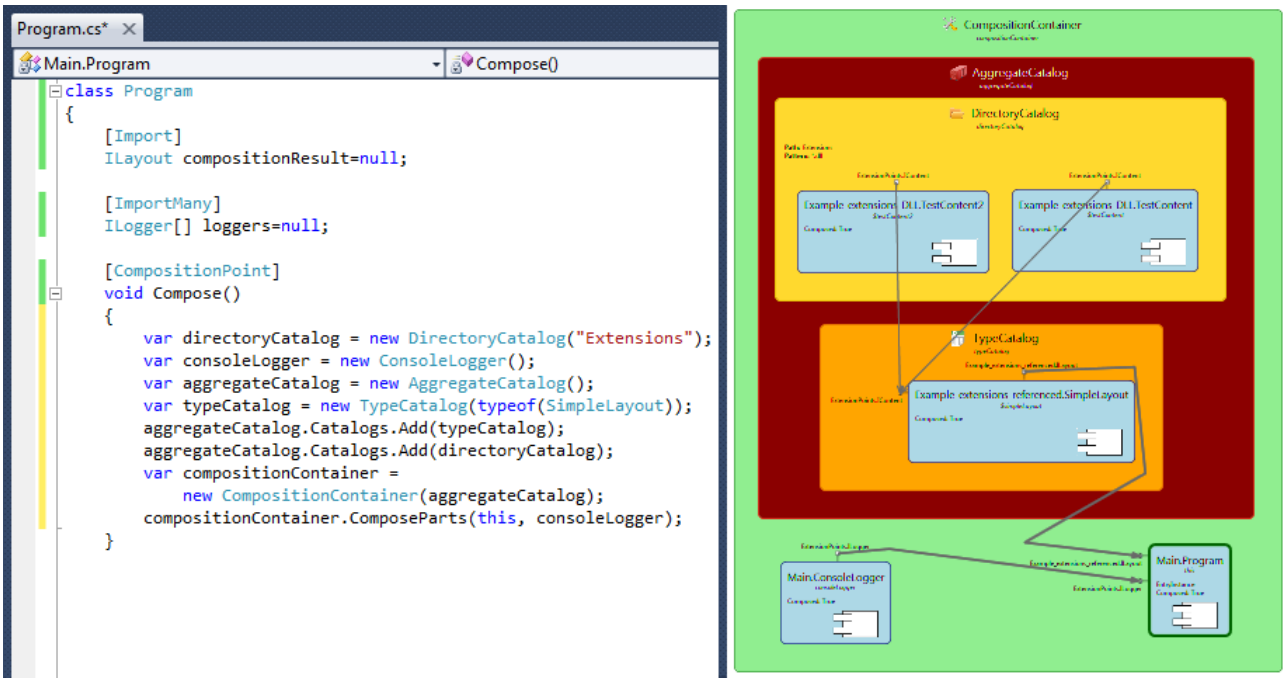
V kontextové nabídce globálních editací přidáme `DirectoryCatalog` pro složku *Debug/Extensions*. Stejným způsobem pak vytvoříme `TypeCatalog` a pomocí jeho specifické editace *Add component type* přidáme `SimpleLayout` komponentu. Pro kompozici však potřebujeme všechny katalogy přesunout do jediného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




*5-5 Postup přesouvání katalogů s komponentami do AggregateCatalog*

### Kompozice schématu

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

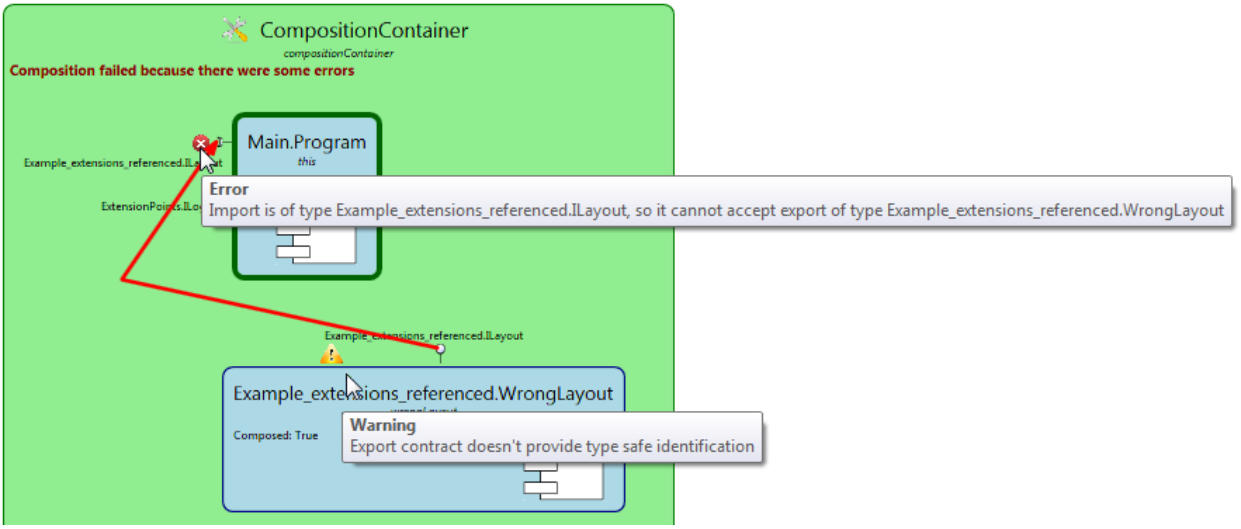



*5-6 Konečná podoba schématu kompozice vytvořená v rámci ukázkového příkladu. Zdrojový kód v metodě Compose byl vygenerován editorem na základě prováděných editací.*

Spuštěním aplikace získáme http server, poskytující stránku vygenerovanou komponentami dostupnými při kompozici. Definovali jsme tedy kompozici ukázkové aplikace.

### Další použití editoru

Pomocí *doporučených rozšíření* můžeme navíc detekovat některé chyby, které mohou při kompozici vzniknout. Příklad zobrazení chyby můžeme vidět na následujícím obrázku:




*5-7 Ukázka jakým způsobem jsou hlášeny chyby odhalené v kompozici. Zde například třída WrongLayout nesplňuje rozhraní ILayout slibované kontraktem.*

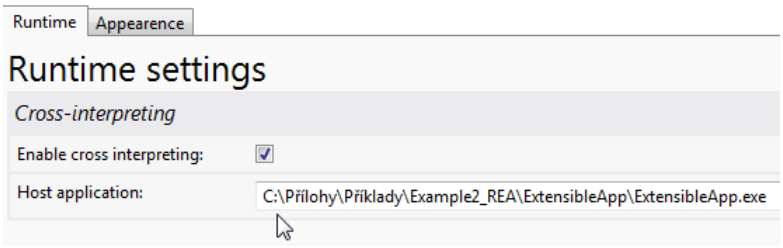
Použití editoru nemusí však vycházet pouze z nabízených editací. Editor reaguje i na změny zdrojového kódu, provedené přímo uživatelem. Stejně tak je schéma kompozice překresleno například při změně ve složce sledované katalogem `DirectoryCatalog`. Díky tomu může sloužit pro kontrolu aktuálního schématu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

### 5.3.2 Použití editoru v konfiguraci REA

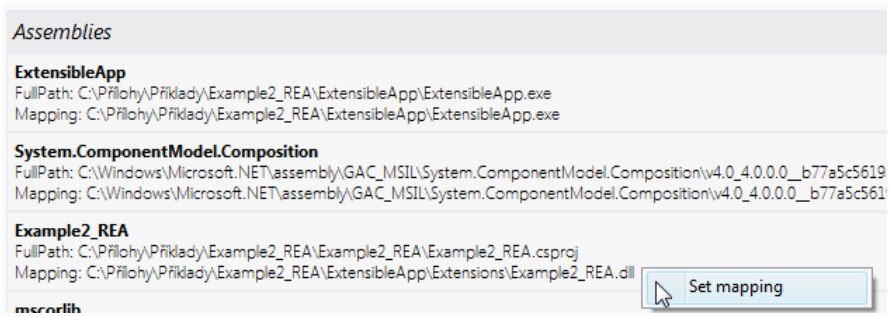
Předpokládejme, že máme spuštěný editor a ve *Visual Studiu* otevřené solution *Example2_REA.sln* z přílohy [G]. Struktura solution má reprezentovat situaci, kdy píšeme rozšíření pro zkompilovanou aplikaci *ExtensibleApp.exe*. Rozšíření budou muset implementovat rozhraní definovaná v knihovně *ExtensibleApp.Interfaces.dll*. Samotnou rozšiřovanou aplikaci však referencovat nebudeme.

Popsaná vývojová konfigurace odpovídá konfiguraci REA z kapitoly 1.3. Díky tomu si můžeme ukázat, jakým způsobem je možné v této konfiguraci editor použít. Nejprve je nutné nastavit cestu ke zkompilované aplikaci, kterou rozšiřujeme. Přepneme se tedy do nastavení editoru a v záložce *Runtime* nastavíme pole *Host application* poklepáním myši a výběrem souboru aplikace v zobrazeném dialogu. Soubor je možné nalézt v příloze [G]. Průběh nastavování je zachycen na obrázku:



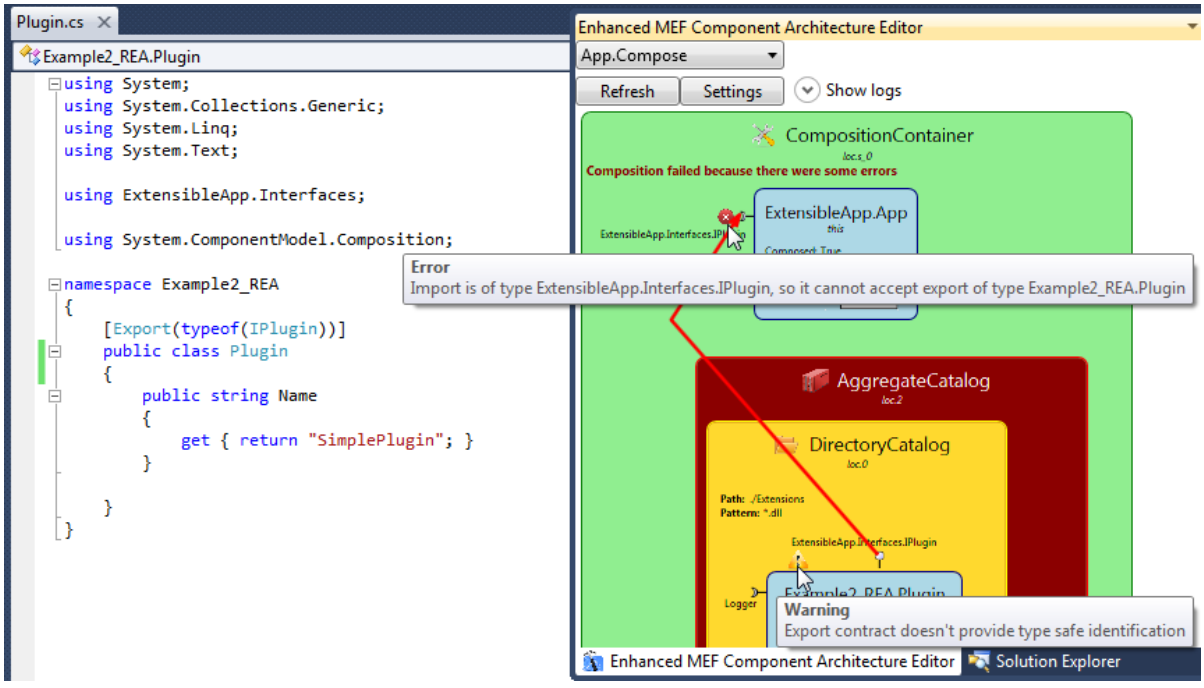

*5-8 Nastavování cesty k rozšiřitelné aplikaci.*

Rozšiřovaná aplikace očekává přítomnost rozšíření ve formě zkompilovaných knihoven v adresáři *Extensions*. V nastavení *Runtime* v sekci *Assemblies*, proto ještě nastavíme mapování assembly *Example2_REA.csproj* na *Example2_REA.dll*. Tak zajistíme, že při pokusu rozšiřované aplikace o načtení rozšíření bude naše knihovna nalezena. Nastavení provedeme s využitím kontextového menu zobrazeného po kliknutí pravým tlačítkem myši na požadovanou assembly, jak můžeme vidět na následujícím obrázku:



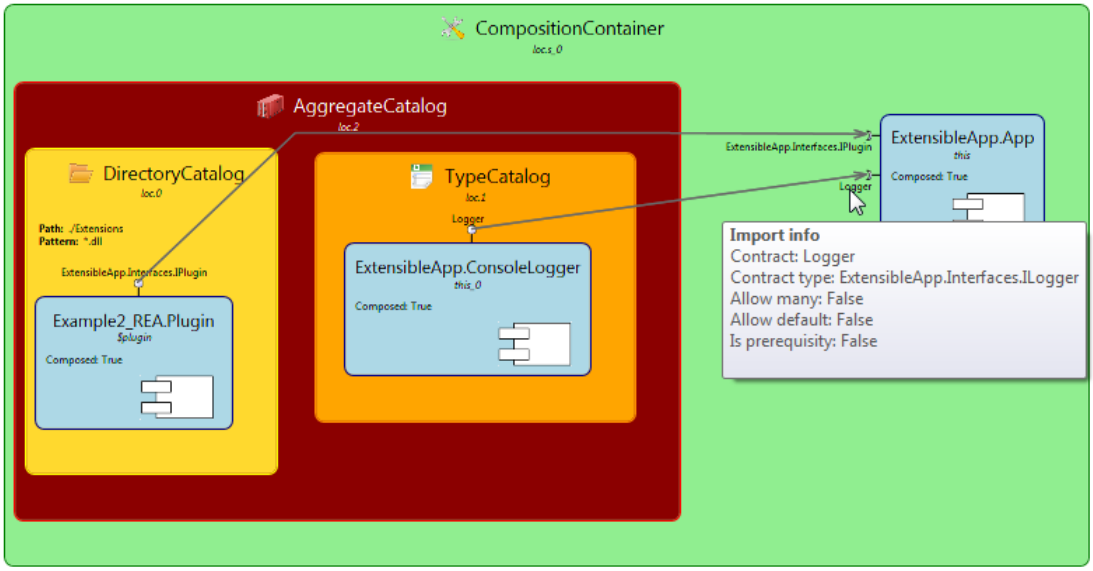

*5-9 Nastavení mapování do složky rozšíření rozšiřitelné aplikace.*

Nastavením cesty k rozšiřované assembly se nám objevil v seznamu *composition pointů composition point* `App.Compose`. Jeho spuštěním získáme kompoziční schéma rozšiřované aplikace. Připomeňme, že toto schéma pochází z analýzy zkompilované aplikace *ExtensibleApp.exe*. Schéma můžeme vidět na následujícím obrázku:




*5-10 Chybové hlášení kvůli chybějícímu rozhraní třídy Plugin.*

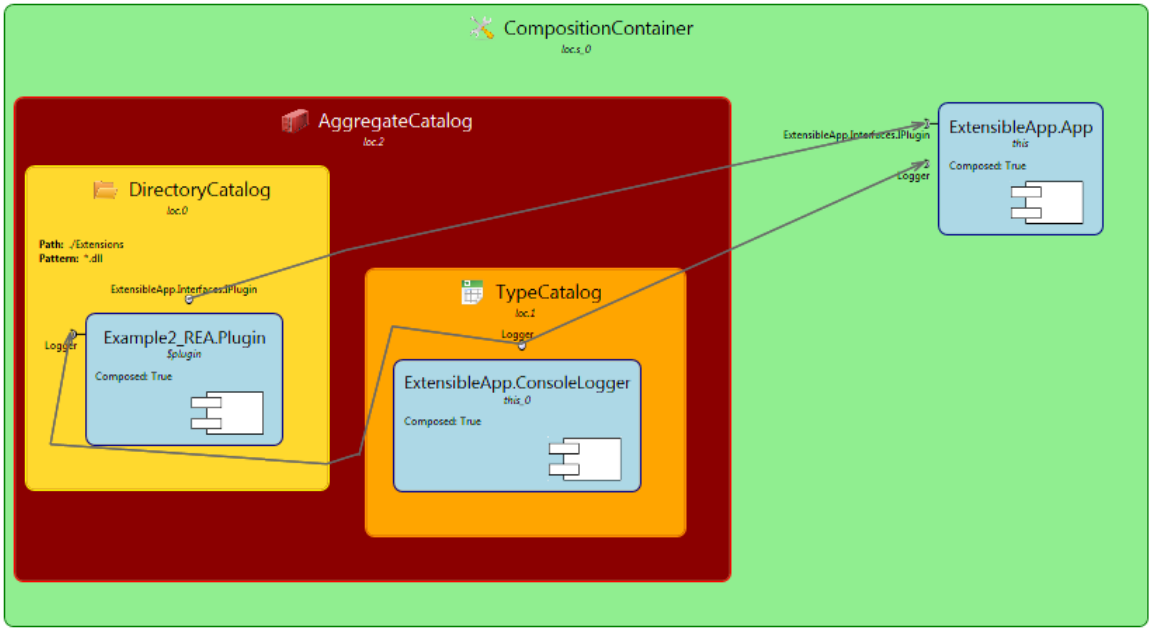
Všimněme si, že při analýze kompozice aplikace byla odhalena chyba. Ta je způsobena tím, že rozšíření v podobě třídy `Plugin` není odvozené od rozhraní `IPlugin`. Přidáním tohoto rozhraní do seznamu předků chybu vyřešíme a kompozice proběhne v pořádku, jak můžeme vidět na následujícím obrázku:




*5-11 Ukázka úspěšné kompozice po opravě chyby.*

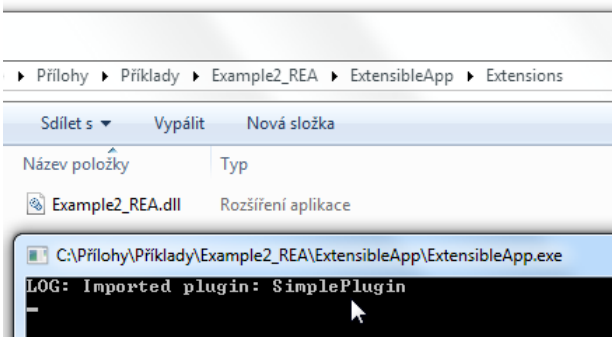
Z kompozičního schématu můžeme dále vidět, že rozšiřovaná aplikace poskytuje komponentu pro logování zpráv. Abychom ji mohli v rozšiřující knihovně využít, musíme nejprve zjistit kontrakt a typ, se kterým je exportovaná. Toho docílíme podržením ukazatele myši nad konektorem, kde je komponenta využita, dokud se nám nezobrazí kontextová informace.

Vidíme, že pro import je nutné definovat kontrakt `Logger`, který splňuje rozhraní `ILogger`. Upravíme proto patřičně naše rozšíření a v zobrazeném schématu kompozice uvidíme, že komponenta pro logování zpráv je naimportována dle očekávání. Úpravu komponenty s výsledným schématem kompozice můžeme vidět na následujícím obrázku:




*5-12 Kompozice s využitím komponenty pro logování zpráv, poskytnuté rozšiřovanou aplikací.*

Nyní můžeme knihovnu rozšíření zkompilovat a umístit ji do složky *Extensions* rozšiřované aplikace. Pokud aplikaci spustíme, uvidíme, že bylo naše rozšíření úspěšně načteno, což poznáme podle ukázkového výpisu v okně konzole:




*5-13 Ukázka výpisu rozšiřitelné aplikace po spuštění s načteným pluginem.*

Tím jsme dokončili ukázku základního použití editoru v konfiguracích REA. Jak vidíme, editor dokáže nejen upozorňovat na možné chyby kompozice, ale pomáhá i lépe pochopit architekturu rozšiřované aplikace.

# 6 Rozšiřitelnost editoru

V této kapitole seznámíme uživatele, který chce rozšířit možnosti editoru, se způsobem, jakým se rozšíření implementují. Možnosti rozšiřitelnosti však nejsou omezené pouze na služby, které budeme využívat v ukázkových příkladech. Pro hlubší pochopení a náhled na rozšiřitelnost doporučujeme prostudování vývojářské dokumentace dostupné v příloze [F].

V první části této kapitoly připravíme prostředí rozšiřující knihovny. Následně implementujeme podporu pro ukázkovou assembly, abychom si ukázaly možnosti rozšiřitelnosti editoru o další jazyky. V kapitole 6.3 se seznámíme s tvorbou vlastních typových definic, které umožňují definovat vlastní chování typů a editace, které nabízí. Poslední rozšíření implementujeme v kapitole 6.4. Toto rozšíření představí možnosti zobrazení vlastního typu ve schématu kompozice.

Implementovaná rozšíření použijeme v kapitole 6.6. Zde si ukážeme možnosti aplikace určené pro ladění rozšíření mimo Visual Studio.

Na závěr si v kapitole 6.7 popíšeme *doporučená rozšíření*, která byla vyvinuta v rámci této práce.

## 6.1 Projekt pro rozšiřující knihovnu.

Pro seznámení se s rozšiřitelností našeho editoru naimplementujeme několik ukázkových rozšíření. Ve všech případech budeme rozšíření vyvíjet v rámci C# projektu vytvořeném ve Visual Studiu. Pomocí průvodce, který Visual Studio nabízí, tedy vytvoříme projekt *UserExtensions*.

Napojení na služby poskytované naším editorem zajistí reference na následující knihovny: *MEFEditor.TypeSystem.dll, MEFEditor.Analyzing.dll, MEFEditor.Drawing.dll* z přílohy [E]. Tím jsou přípravy pro implementaci rozšíření dokončené.

Projekt s kompletními zdrojovými kódy, vytvořenými v rámci příkladů je možné nalézt v solution *UserExtensions.sln* z přílohy [G].

## 6.2 Poskytování assembly

Typový systém potřebuje pro svou práci načítat *typové definice* z různých reprezentací assembly. Příkladem takových reprezentací může být například zkompilovaná assembly v instrukcích CIL, nebo zdrojové kódy projektu jazyka C#. Aby bylo možné editor rozšířit o podporu nového typu assembly, je nutné vytvořit rozšíření v podobě třídy implementující `AssemblyProvider`, které budeme nazývat *poskytovatel assembly*. Toto rozšíření je pak nutné registrovat podle návodu, který bude uveden v kapitole 6.5, a knihovnu se zkompilovaným rozšířením umístit do *složky rozšíření* editoru.

Implementace poskytovatele assembly v sobě zahrnuje implementaci algoritmů pro vyhledávání komponent, metod typů definovaných v assembly a jejich překlad do analyzačních instrukcí.

Z těchto důvodů, implementace poskytovatele assembly může zahrnovat značné množství kódu. Aby byl přesto editor snadno rozšiřitelný o podporu dalších .NET jazyků obsahují *doporučená rozšíření* knihovnu *RecommendedExtensions.Core.dll,* kde máme předpřipravenou třídu `VsProjectAssemblyProvider`, u které pouze

implementujeme překlad ze zdrojového kódu přidávaného jazyka do analyzačních instrukcí.

### Implementace jednoduchého poskytovatele assembly

Abychom si na jednoduchém příkladu ukázali možnosti přidání nového typu assembly, implementujeme jednoduchý poskytovatel assembly. Ten bude vytvářet assembly pro soubory s koncovkou *.test*. Vytvořená assembly bude definovat jediný typ s jedinou metodou.

Implementaci začneme přípravou třídy `SimpleAssemblyProvider` odvozenou od `AssemblyProvider`. Také přidáme metody, které jsou pro poskytovatele assembly povinné a datové položky, ve kterých budeme uchovávat informace o assembly a metodě, kterou definuje. Protože je implementace této třídy rozsáhlejší, uvedeme si její signaturu postupně. Začneme deklarací potřebných datových položek:

```
class SimpleAssemblyProvider : AssemblyProvider
{
    //jméno reprezentované assembly
    private readonly string _name;

    //cesta k souboru který "definuje" assembly
    private readonly string _fullPath;

    //typ který je definován reprezentovanou assembly
    private readonly TypeDescriptor _declaringType;

    //zde jsou uchovány všechny metody, které assembly definuje
    private readonly HashedMethodContainer _methods;
```

*6-1 Datové položky, které budeme v implementaci potřebovat.*

Dále bude následovat několik metod, které jsou důležité pro potřeby ukázkového příkladu. Zde naimplementujeme požadovanou funkčnost. Jejich signatura je vidět zde:

```
    //konstruktor, který dostává parametrem cestu k definujícímu souboru
    public SimpleAssemblyProvider(string testFileFullPath)

    //typovému systému umožníme zjistit cestu definujícího souboru
    protected override string getAssemblyFullPath()

    //každá assembly musí mít nějaké jméno
    protected override string getAssemblyName()

    //vyhledávání se provádí postupným procházením assembly
    public override SearchIterator CreateRootIterator()

    //metoda pro získávání generátoru instrukcí zadané metody
    public override GeneratorBase GetMethodGenerator(MethodID method)

    //metoda pro zjišťování hierarchie dědičnosti zadaného typu
    public override InheritanceChain GetInheritanceChain(PathInfo typePath)
```

*6-2 Signatury metod důležitých pro implementaci ukázkového rozšíření.*

Nakonec si ukážeme signatury metod, které nejsou důležité pro ukázkový příklad, nicméně je nutné poskytnout alespoň jejich triviální implementaci, neboť jsou tyto metody vynucené abstraktní třídou `AssemblyProvider`:

```
    //metoda která začne vyhledávání komponent v assembly
    protected override void loadComponents()

    //metoda pro získání identifikátoru metody
    //implementující zadanou generickou metodu
    public override MethodID GetGenericImplementation(MethodID methodID,
       PathInfo methodSearchPath, PathInfo implementingTypePath,
       out PathInfo alternativeImplementer)

    //metoda pro získání generátoru instrukcí zadané generické metody
    public override GeneratorBase GetGenericMethodGenerator(MethodID method,
       PathInfo searchPath)

    //metoda pro získání identifikátoru metody implementující zadanou metodu
    public override MethodID GetImplementation(MethodID method,
       TypeDescriptor dynamicInfo, out TypeDescriptor alternativeImplementer)

}
```

*6-3 Signatury metod, které nejsou z hlediska ukázkového rozšíření důležité.*

Pro potřeby ukázky nebude reprezentovaná assembly poskytovat žádné generické metody, implementovat žádná rozhraní ani definovat žádné komponenty. Odpovídající metody proto budou mít pouze triviální implementaci. Jedná se o metody:

- `GetGenericMethodGenerator` – je volána pro získání generátoru instrukcí generické metody
- `GetGenericMethodImplementation` – je volána pro zjištění metody, která implementuje zadanou metodu z generického rozhraní,
- `GetImplementation` – je volána pro zjištění metody, která implementuje zadanou metodu z rozhraní,
- `loadComponents` – je volána jako požadavek o načtení komponent definovaných v reprezentované assembly.

Další metody, které pro ukázku nejsou důležité, také implementujeme triviálním způsobem. Jedná se o metody `getAssemblyName` a `getAssemblyFullPath`. Implementaci všech těchto metod je možné nalézt v solution *UserExtensions.sln* z přílohy [G].

Implementaci podstatných částí našeho poskytovatele začneme v konstruktoru. Nejprve uložíme informace o assembly. Také připravíme kontejner, do kterého uložíme definovanou metodu.

```
//uchováme cestu k "definujícímu" souboru
_fullPath = testFileFullPath;

//jméno assembly odvodíme od názvu souboru
_name = Path.GetFileName(_fullPath);

//připravíme kontejner kam vložíme definovanou metodu
_methods = new HashedMethodContainer();
```

*6-4 Inicializační část konstruktoru.*

Nyní začneme s definicí metody, kterou chceme v assembly definovat. Nejprve potřebujeme určit informace o signatuře pomocí třídy `TypeMethodDescriptor`. Způsob definice signatury je evidentní z následujícího obrázku:

```
//vytvoření metody začneme přípravou typu, kde je definovaná
_declaringType = TypeDescriptor.Create("MEFEditor.ProviderTest");
//určíme jméno metody
var methodName = "GetDefiningAssemblyName";
//návratový typ metody
var returnType = TypeDescriptor.Create<string>();

//z definovaných údajů můžeme vytvořit popis
//metody, která nebude mít žádné parametry a bude statická
var methodInfo = new TypeMethodInfo(
    _declaringType, methodName, returnType,
    ParameterTypeInfo.NoParams,
    isStatic: true,
    methodTypeArguments: TypeDescriptor.NoDescriptors);
```

*6-5 Vytvoření definice signatury metody*

Poté co máme vytvořen popis metody, můžeme konečně vytvořit její reprezentaci. Součástí této reprezentace je i `GeneratorBase`, který zajišťuje generování instrukcí metody. Takto definovanou metodu nakonec vložíme do kontejneru definovaných metod, jak můžeme vidět zde:

```
//k dokončení definice metody stačí vytvořit
//generátor jejích analyzačních instrukcí
var methodGenerator = new DirectedGenerator(emitDirector);

//definovanou metodu vytvoříme
var method = new MethodItem(methodGenerator, methodInfo);
//aby byla metoda dohledatelná, musíme ji ještě zaregistrovat
_methods.AddItem(method);
```

*6-6 Vytvoření a definice metody s využitím kontejneru _methods.*

Všimněme si metody `emitDirector`, která je předána generátoru instrukcí. Tento delegát je zavolán v případě, že jsou potřeba instrukce definované metody. Následně pak tyto instrukce vytvoří pomocí `EmitterBase`. Implementace je patrná z následujícího kódu:

```
private void emitDirector(EmitterBase emitter)
{
    //emitujeme instrukci pro uložení jména
    //assembly do proměnné
    emitter.AssignLiteral("result", _name);

    //instrukce pro vrácení uložené hodnoty
    emitter.Return("result");
}
```

*6-7 Vytvoření instrukcí metody, kterou v assembly definujeme.*

Tím máme dokončenou inicializaci našeho poskytovatele assembly. Aby však bylo definovanou metodu možné nalézt, musíme implementovat metodu `CreateRootIterator`, která slouží pro prohledávání metod definovaných v assembly. Takto například překladače vyhledávají signatury metod podle zdrojových kódů. Samotná signatura by ale pro interpretaci nestačila a je tedy nutné poskytovat i generátor instrukcí. K tomuto účelu slouží `GetMethodGenerator`. S využitím připravených tříd v typovém systému je jejich implementace snadná:

```
public override SearchIterator CreateRootIterator()
{
    //vytvoříme iterátor, který dokáže procházet
    //definované metody
    return new HashedIterator(_methods);
}

public override GeneratorBase GetMethodGenerator(MethodID method)
{
    //zkusíme nalézt metodu dle zadaného ID
    return _methods.AccordingId(method);
}
```

*6-8 Metody umožňující vyhledávání typů a implementací metod v assembly.*

K dokončení poskytovatele assembly zbývá přidat metodu, ze které *typový systém* získává informace o dědičné hierarchii definovaného typu. K tomuto účelu slouží `GetInheritanceChain`. Pokud z parametru zjistíme, že se typový systém dotazuje na typ, který definujeme, musíme vrátit informace o jeho dědičné hierarchii. V kódu to pak vypadá následovně:

```
public override InheritanceChain GetInheritanceChain(PathInfo typePath)
{
    //informace o dědičnosti poskytujeme pouze
    //pro náš definovaný typ
    if (typePath.Signature == _declaringType.TypeName)
    {
        //definovaný typ je potomkem typu object
        InheritanceChain baseType;
        baseType = TypeServices.GetChain(TypeDescriptor.ObjectInfo);
        return TypeServices.CreateChain(_declaringType, new[] { baseType });
    }

    //dotaz se týkal typu, který nedefinujeme
    return null;
}
```

*6-9 Poskytování informací o dědičnosti definovaného typu s využitím TypeServices*

Všimněme si využití třídy `TypeServices`, která zpřístupňuje služby typového systému. V tomto případě získáme informace o dědičnosti typu `object` z assembly, která ho definuje. Na jeho základě pak vytvoříme `InheritanceChain` pro náš definovaný typ.

Tím máme implementaci rozšíření kompletní. Aby byl poskytovatel assembly do editoru načten, musíme poskytovatele nejprve zaregistrovat. Jakým způsobem se registrace provádí se dozvíme v kapitole 6.5.

## 6.3 Typové definice

V této kapitole si představíme nové možnosti rozšiřitelnosti editoru o typové definice. Díky typovým definicím můžeme definovat chování *instancí* v průběhu interpretace. Proto mohou typové definice vytvářet editace, nabízené *instancemi*.

Všechny rozšiřující typové definice ze složky rozšíření jsou při spuštění editoru nahrávány do assembly *Runtime*, typového systému. Díky tomu, že *Runtime* je vždy první assembly, ve které se hledají implementace metod, budou typové definice přepisovat chování typů ve zdrojových kódech nebo referencovaných assembly. Každá rozšiřující typová definice musí být odvozena od typu `RuntimeTypeDefinition`. Pokud chceme typovou definici použít v editoru, musíme ji exportovat podle návodu, který bude uveden v kapitole 6.5.

### Druhy typových definic

Současná verze editoru podporuje dva různé typy typových definic popsaných v kapitole 4.3.5. První typ je přímá typová definice `DirectTypeDefinition`, který se využívá v případech, kdy chceme reprezentovaný objekt uchovávat v nativní .NET podobě. Kvůli této vlastnosti jsou vhodné pro reprezentace objektů, jejichž chování nechceme příliš měnit.

Druhým typem jsou datové typové definice `DataTypeDefinition`, které využíváme pro typy, s jejichž objekty nechceme pracovat v nativní podobě. Příkladem mohou být typy, které pracují se souborovým systémem. Pokud bychom s nimi v průběhu interpretace pracovali v nativní podobě, docházelo by tak k nechtěným změnám v souborovém systému.

Další odlišností typových definic je ten, že přímé typové definice dokáže typový systém automaticky konvertovat do nativní podoby. Díky tomu je na rozdíl od předchozí verze editoru implementace rozšíření snazší.

### Metody typových definic

Stěžejní částí typových definic je implementace metod, které budou simulovat chování reprezentovaného objektu v průběhu interpretace. Jejich definice se v současné verzi editoru provádí odlišně od verze předchozí. Nový způsob jejich definice je v současné verzi přehlednější a vyžaduje menší množství kódu, který musí uživatel implementovat.

Metody v typové definici, které mají být pro definovaný typ dostupné, musí být označené jako `public` a jejich jméno musí podléhat konvencím z následující tabulky:

| **Jmenná konvence** | **Význam** |
|---|---|
| `_method_ctor` | Definuje konstruktor, který je volán při vytvoření *instance* reprezentovaného typu. |
| `_static_method_cctor` | Definuje statický konstruktor, který je volán při prvním použití nějaké statické metody reprezentovaného typu. |
| **Prefix:** `_method_` | Podle jména uvedeného za prefixem definuje metodu reprezentovaného typu. |
| **Prefix:** `_static_method` | Podle jména uvedeného za prefixem definuje statickou metodu reprezentovaného typu. |
| **Prefix:** `_get_` | Podle jména uvedeného za prefixem definuje getter vlastnosti reprezentovaného typu. |
| **Prefix:** `_static_get` | Podle jména uvedeného za prefixem definuje statický getter vlastnosti reprezentovaného typu. |
| **Prefix:** `_set_` | Podle jména uvedeného za prefixem definuje setter vlastnosti reprezentovaného typu. |
| **Prefix:** `_static_set` | Podle jména uvedeného za prefixem definuje statický setter vlastnosti reprezentovaného typu. |

*Tabulka se jmennými konvencemi pro definice metod v typových definicích.*

U definic těchto metod je také důležité, jaké definují parametry a návratové typy, tedy jakou definují signaturu. Reprezentovaná metoda totiž bude mít stejnou signaturu, až na automatické konverze prováděné z přímé instance na nativní objekt a zpět.

U typů, pro něž nemáme přímou typovou definici, musíme uvést jako typ parametru nebo návratové hodnoty `Instance`. Typ tohoto parametru nebo návratové hodnoty pak můžeme explicitně definovat pomocí atributů `ParameterTypes` nebo `ReturnType`.

V následujícím obrázku si ukážeme, jakým způsobem se používají atributy pro určení typů. První definice nepotřebuje určit typy, proto žádné atributy nevyužívá. U druhé definice využijeme určení návratové hodnoty i typu jejích parametrů:

```
//pokud máme přímou typovou definici
//pro int můžeme typ určit přímo
public void _method_Test1(int p1)

//určíme návratový typ metody
//v případě že se jedná o object však
//explicitní typování není nutné
[ReturnType(typeof(object))]
//parametry zapisujeme v pořadí, jak se objevují v metodě
[ParameterTypes(typeof(int),typeof(string))]
public Instance _method_Test2(Instance p1, Instance p2)
```

*6-10 Ukázka definic metod a použití explicitního typování pomocí atributů.*

### Kontext metod typových definic

Stejně jako v předchozí verzi editoru jsou metody typových definic vyvolávány v průběhu interpretace. Pokud tedy dojde k vyvolání metody na *instanci*, dohledá se podle jejího typu odpovídající metoda patřičné typové definice, která bude volání na *instanci* simulovat.

Je zřejmé, že v průběhu interpretace se může objevit více *instancí* stejného typu. Kvůli tomu mohou i simulovaná volání v typových definicích patřit pokaždé k různé *instanci*. Z tohoto důvodu typový systém před každým voláním mění kontext v odpovídající typové definici. Do kontextu patří údaje o `this` objektu a editace spojená s aktuálním voláním.

Ze stejného důvodu také není možné ukládat datové položky přímo v typové definici, ale je nutné využít třídu `Field`, kterou si blíže představíme v ukázkovém příkladu pro datovou typovou definici.

## 6.3.1 Příklad implementace DirectTypeDefinition

První typová definice, kterou naimplementujeme, bude přímá typová definice. Tato definice bude upravovat chování metody `ToString` na objektech typu `string`. Metodu změníme tak, aby k běžnému výsledku volání této metody přidala prefix. Jak bude chování změněné metody vypadat, demonstruje následující obrázek:

```
var result = "String".ToString();
//Díky pozměněné metodě ToString bude
//proměnná result obsahovat hodnotu:
//Changed: String
```

*6-11 Kód demonstrující možnosti typových definic.*

Poznamenejme, že tohoto chování není možné běžnými prostředky v jazyce C# dosáhnout. V průběhu interpretace máme však větší kontrolu nad prováděným kódem. Použijeme proto tento příklad jako ukázku schopností typových definic.

Implementaci zahájíme vytvořením třídy `StringDefinition`, odvozenou od třídy `DirectTypeDefinition<T>`. Generický parametr `T` zde představuje typ objektu, který typová definice reprezentuje. Objekt tohoto typu bude také nativně uložen v *instanci*. V našem případě za parametr `T` dosadíme `string`.

Také vytvoříme metodu, ve které naimplementujeme změněné chování metody `ToString`. Jméno metody musí odpovídat konvencím definovaným v tabulce kapitoly 6.3.

Tím dostáváme následující kód:

```
class StringDefinition : DirectTypeDefinition<string>
{
    //definice metody, ve které implementujeme
    //změněné chování String.ToString
    public string _method_ToString()
}
```

*6-12 Přímá typová definice, připravená k implementaci.*

Přestože v definici implementujeme jedinou metodu, může výsledná typová definice automaticky přidat i další metody typu `String`. Pokud jsou v *Runtime* všechny typy pro parametry a návratové hodnoty reprezentovány přímými definicemi, dokáže *typový systém* takové metody upravit, pro použití při interpretaci.

Jediným podstatným krokem při tvorbě naší přímé typové definice bude implementace změněného chování v metodě `_method_ToString`. Pro tento krok využijeme `ThisValue`, která díky kontextu, který je pro každé volání přepnut, obsahuje uchovávaný nativní objekt. S jeho využitím je implementace snadná:

```
public string _method_ToString()
{
    //v ThisValue máme hodnotu reprezentovanou
    //právě volanou instancí, vytvoříme
    //proto upravenou hodnotu
    var result = "Changed: " + ThisValue;
    //a tuto hodnotu vrátíme
    return result;
}
```

*6-13 Implementace změněného chování metody ToString.*

Tím je implementace přímé typové definice dokončena. Pokud umístíme zkompilovanou knihovnu do složky rozšíření, bude při spuštění *Visual Studia* tato definice načtena.

### 6.3.2 Příklad implementace DataTypeDefinition

Ukázková implementace datové typové definice bude představovat jednoduchou třídu určenou pro pomocné výpisy zobrazované ve schématu kompozice. Pro tuto implementaci předpokládáme stejné požadavky na projekt vytvářené rozšiřující knihovny, jako při vývoji přímé typové definice v předchozí kapitole.

V projektu vytvářené rozšiřující knihovny nejprve vytvoříme třídu `DiagnosticDefinition`, odvozenou od třídy `DataTypeDefinition`. Také vytvoříme metody, které budou reprezentovat chování objektu. Metody musí odpovídat konvencím definovaným v tabulce kapitoly 6.3. Dále do typové definice přidáme definice datových položek, které budou ukládány přímo v *instanci*. Definice provedeme s využitím třídy `Field`.

Tím dostáváme následující zdrojový kód:

```
public class DiagnosticDefinition : DataTypeDefinition
{
    //datová položka pro časovač
    protected Field<Stopwatch> Watch;

    //datová položka pro uložené instance
    protected Field<List<Instance>> Instances;

    //datová položka pro zprávu
    protected Field<string> Message;

    //v konstruktoru provedeme inicializaci
    public DiagnosticDefinition()
´
    //zde implementujeme poskytování informací pro vykreslení
    protected override void draw(InstanceDrawer drawer)

    //====následuje definice metod reprezentovaného typu====

    //definice konstruktoru
    public void _method_ctor()

    //definice setteru pro vlastnost Message
    public void _set_Message(string message)

    //definice getteru pro vlastnost Message
    public string _get_Message()

    //definice metody Start
    public void _method_Start()

    //definice metody Stop
    public void _method_Stop()

    //definice metody Accept, ve které využíváme
    //explicitní určení návratového typu atributem
    //typ parametru neuvedeme, bude proto defaultně
    //brán jako object[]
    [ReturnType(typeof(int))]
    public Instance _method_Accept(params Instance[] accepted)

}
```

*6-14 Typová definice s metodami připravenými pro implementaci.*

Implementaci začneme inicializací informací o typu v konstruktoru *typové definice*. K těmto informacím patří jméno definovaného typu a jeho předek v typové dědičnosti. Kdybychom měli k dispozici odpovídající nativní typ, mohli bychom opět využít metodu `Simulate`, jak jsme si ukázali v příkladu v předchozí kapitole. Nyní však musíme tyto informace uvést explicitně.

Implementaci konstruktoru zakončíme definicí globální editace na vytvoření *instance* ve zdrojovém kódu. Upozorněme, že datové položky `Field` jsou inicializované automaticky typovým systémem a není nutné je v konstruktoru přiřazovat. Dostáváme tak následující zdrojový kód:

```
//nastavíme jméno reprezentovaného typu
FullName = "MEFEditor.Diagnostic";

//a také nastavíme typ od kterého dědíme
//poznamenejme, že object je defaultní předek
//zde je však uveden pro ukázku použití API
ForcedSubTypes = new[] { typeof(object) };

//nakonec přidáme editaci na vytvoření instance
AddCreationEdit("Add Diagnostic");
```

*6-15 Implementace konstruktoru naší typové definice.*

Nyní se již můžeme pustit do implementace metod naší *typové definice*. Nejprve začneme implementací `_method_ctor`, odpovídající konstruktoru reprezentovaného objektu.

V této metodě inicializujeme datové položky:

```
public void _method_ctor()
{
    //inicializujeme datové položky instance
    Message.Value = "DefaultMessage";
    Watch.Value = new Stopwatch();
    Instances.Value = new List<Instance>();
}
```

*6-16 Inicializace položek v konstruktoru reprezentovaného typu.*

Všimněme si způsobu použití datových položek. Na rozdíl od předchozí verze již není nutné provádět při čtení datové položky z *instance* explicitní přetypování. Upozorněme také, že datové položky jsou v *instanci* uchovávány v nativní podobě.

Třída `Diagnostic` bude nabízet měření intervalů pomocí metod `Start` a `Stop`. Jejich implementace bude pouze obsluhovat spuštění a zastavení časovače, jak je evidentní z následujícího zdrojového kódu:

```
public void _method_Start()
{
    //spustíme měření času
    Watch.Value.Start();
}

public void _method_Stop()
{
    //zastavíme měření času
    Watch.Value.Stop();
}
```

*6-17 Obsluha časovače v metodách Start a Stop.*

Abychom si ukázali, jak se definují vlastnosti, bude naše *typová definice* obsahovat vlastnost `Message`. Aby ji bylo možné číst i nastavovat, musíme implementovat getter i setter. S využitím třídy `Field` je taková implementace snadná, jak můžeme vidět na následujícím kódu:

```
public void _set_Message(string message)
{
    //nastavíme hodnotu pro zprávu
    Message.Value = message;
}

public string _get_Message()
{
    //vrátíme hodnotu pro zprávu
    return Message.Value;
}
```

*6-18 Implementace getteru a setteru vlastnosti Message.*

Poslední metodou, kterou bude *typová definice* nabízet, je metoda `Accept`. Tato metoda bude přidávat *instance* do seznamu uchovávaných *instancí*. Vracet bude jejich celkový počet.

Metoda `Accept` bude také podporovat editace na přidávání a odebírání komponent pomocí akcí drag&drop. Implementaci začneme zpracováním argumentů. Určíme, že za poslední argument je možné přidat další instanci. Každý argument pak nastavíme jako logického potomka *instance*, na které je volán `Accept`. Díky tomu je bude možné pomocí editací odstranit. Ve zdrojovém kódu to vypadá následovně:

```
//nastavíme akceptování instancí za poslední argument
AcceptAsLastArgument(componentAccepter);

//všechny argumenty přidáme do seznamu
for (var i = 0; i < accepted.Length; ++i)
{
    //zpracovávaný argument
    var acceptedInstance = accepted[i];
    //označíme jako logické dítě získané z parametrického volání metody
    ReportParamChildAdd(i, acceptedInstance, "Accepted child", true);

    //následně argument uložíme do seznamu
    Instances.Value.Add(acceptedInstance);
}
```

*6-19 Zpracování argumentů v metodě Accept.*

Všimněme si metody `componentAccepter`, která bude využita pro akceptování argumentů z uživatelem prováděných editací. V této metodě je implementována logika, která rozhodne, zda chceme *instanci* akceptovat, podle toho, zda se jedná o komponentu. Pokud se o komponentu nejedná, můžeme uživateli sdělit, proč *instanci* akceptovat nechceme, metodou `Abort`. Jinak vrátíme proměnnou, ve které se *instance* nachází. Její získání může způsobit různé editace ve zdrojovém kódu, které však řeší metoda `Edits.GetVariableFor`. Způsob použití vidíme na následujícím kódu:

```
private object componentAccepter(ExecutionView view)
{
    //instanci, kterou uživatel přesunul myší, získáme následovně
    var toAccept = UserInteraction.DraggedInstance;

    //zjistíme, zda se jedná o komponentu
    var componentInfo = Services.GetComponentInfo(toAccept.Info);
    //akceptovat chceme pouze komponenty
    if (componentInfo == null)
    {
        //pokud se o komponentu nejedná, sdělíme uživateli
        //proč nelze instanci akceptovat a editaci zrušíme
        view.Abort("Can only accept components");
        return null;
    }

    //pokud akceptujeme komponentu,
    //získáme proměnnou, ve které je dostupná
    //a vrátíme ji jako hodnotu k akceptování
    return Edits.GetVariableFor(toAccept, view);;
}
```

*6-20 Logika akceptování instancí podle toho, zda se jedná o komponenty.*

Implementaci metody `Accept` dokončíme vytvořením návratové hodnoty, která říká, kolik *instancí* již bylo akceptováno. Abychom si ukázali vytváření *instancí*, musíme využít služby `Context.Machine`.

Použití je evidentní ze závěru implementace metody `Accept`:

```
//získáme celkový počet uložených instancí
int nativeCount = Instances.Value.Count;
//vytvoříme přímou instanci pro zadaný počet
Instance wrappedCount = Context.Machine.CreateDirectInstance(nativeCount);

//vytvořenou instanci vrátíme jako výsledek
return wrappedCount;
```

*6-21 Vytvoření návratové hodnoty pro metodu Accept.*

Nyní máme implementovány všechny metody, které bude reprezentovaný typ nabízet. Po naší *typové definici* však budeme požadovat možnost vykreslení ve schématu kompozice. Musíme proto ještě implementovat metodu `draw`. Jejím úkolem je zpřístupnění informací, které může následně definice zobrazení využít pro vykreslení ve schématu kompozice.

Zpřístupňovat budeme akceptované *instance*, tak abychom si ukázali vytvoření slotu, který dokáže akceptované *instance* zobrazit uvnitř jiné *instance*.

Slot, podle něhož bude možné provést vykreslení, vytvoříme následovně:

```
//přidáme definici slotu, ve kterém se budou
//zobrazovat uložené instance
var instanceSlot = drawer.AddSlot();

//vytvořený slot naplníme instancemi
foreach (var instance in Instances.Value)
{
    //pro každou instanci musíme získat reprezentaci
    //jejího zobrazení
    var instanceDrawing = drawer.GetInstanceDrawing(instance);
    //do slotu patří pouze reference na vytvořené zobrazení
    instanceSlot.Add(instanceDrawing.Reference);
}
```

*6-22 Vytvoření slotu s přidáním instancí, které v něm chceme zobrazovat*

Další informace, které chceme zobrazovat, se týkají nastavené zprávy a naměřeného času. K jejich zpřístupnění využijeme metody `drawer.PublishField` a `drawer.SetProperty`. Veškeré informace se předávají v textové podobě, neboť chceme, aby v definici zobrazení byla pouze zobrazovací logika, proto veškeré výpočty musíme provést v *typové definici*.

Implementaci zakončíme nastavením příznaku, že chceme instance reprezentující objekty našeho typu vykreslovat. Příznak se nastavuje pomocí `drawer.ForceShow`, jak můžeme vidět v následujícím obrázku:

```
//předáme informace které vypíšeme uživateli
drawer.PublishField("message", Message);
drawer.SetProperty("time", Watch.Value.ElapsedMilliseconds + "ms");

//vynutíme zobrazení této instance
//ve schématu kompozice
drawer.ForceShow();
```

*6-23 Dokončení implementace metody draw, publikací informací pro definici zobrazení.*

Tím jsme dokončili implementaci celé typové definice. Tu je nyní možné používat v průběhu analýzy *composition pointu* po přidání do složky rozšíření editoru v podobě zkompilované knihovny. Nicméně *instance* získané z této *typové definice* nebudou ještě zobrazovány ve schématu kompozice. Aby je bylo možné zobrazit, potřebujeme ještě implementovat definici zobrazení. Tuto implementaci si ukážeme v následující kapitole.

## 6.4 Definice zobrazení

V kapitole 6.3.2 jsme implementovali ukázkovou rozšiřující typovou definici. Abychom mohli zobrazovat ve schématu kompozice *instance* typu, který je *typovou definicí* reprezentován, musíme ještě definovat způsob, jakým se budou *instance* zobrazovat.

Definice zobrazení musí být odvozena od třídy `ContentDrawing`. Jelikož využívá k zobrazování služeb WPF, musíme přidat do projektu rozšiřující knihovny reference na následující systémové knihovny: *PresentationCore, WindowsBase, System.Xaml, PresentationFramework.*

Po vytvářené definici zobrazení budeme po požadovat, aby zobrazila veškeré vlastnosti, které publikovala *typová definice* v metodě `draw`. Také budeme chtít rekurzivně zobrazit veškeré instance přidané metodou Accept.

Implementace definice zobrazení začneme přípravou typu `DiagnosticDrawing`, kde musíme explicitně uvést konstruktor, který má parametr pro předání `DiagramItem`, uvnitř které bude reprezentovaná *instance* zobrazena.

Dostáváme tak následující kód:

```
class DiagnosticDrawing : ContentDrawing
{
    //konstruktor definice zobrazení musí
    public DiagnosticDrawing(DiagramItem item) :
        base(item)
    {
        //zde bude implementováno vykreslení
    }
}
```

*6-24 Třída definice zobrazení připravená k implementaci.*

Veškeré rutiny potřebné pro vykreslení provedeme v konstruktoru. Nejprve vytvoříme panel, který bude definovat rozložení vykreslovaných elementů. Do tohoto panelu následně přidáme nadpis, který bude označovat zobrazovanou položku. Popsané kroky naimplementujeme následujícím způsobem:

```
//vytvoříme panel, ve kterém zobrazíme informace i položce
var layout = new StackPanel();
layout.Background = Brushes.Green;
//vzhled bude určen panelem layout
Child = layout;

//vytvoříme nadpis označující typ instance
var headline = new TextBlock();
headline.Text = Definition.DrawedType;
headline.FontSize *= 2;
layout.Children.Add(headline);
```

*6-25 Příprava panelu, ve kterém budeme zobrazovat jednotlivé elementy.*

Dále potřebujeme zobrazit hodnoty všech vlastností dostupných pro zobrazovanou položku. Jejich výčet získáme s využitím `Definition.Properties`, jak můžeme vidět v následujícím kódu:

```
//zobrazíme veškerá dostupné vlastnosti zobrazované instance
foreach (var property in Definition.Properties)
{
    //vytvoříme textovou reprezentaci hodnoty vlastnosti
    var block = new TextBlock();
    block.Text = property.Name + ": " + property.Value;

    //a zobrazíme ji ve schématu kompozice
    layout.Children.Add(block);
}
```

*6-26 Zobrazení hodnot dostupných vlastností.*

Implementaci dokončíme zobrazením akceptovaných *instancí*. Informace o nich jsou dostupné v prvním slotu, neboť do něj typová definice, tak jak jsme ji naimplementovali v předchozí kapitole, akceptované *instance* přidala.

S využitím informací o slotu již dokážeme naplnit `SlotCanvas`, který bude *instance* zobrazovat. Naplnění provedeme s využitím metody `Item.FillSlot`. Ve zdrojovém kódu to pak vypadá následovně:

```
//přidáme canvas, do kterého budeme zobrazovat
//akceptované instance
var slotCanvas = new SlotCanvas();
layout.Children.Add(slotCanvas);

//k naplnění canvasu potřebujeme definici
//patřičného slotu
var slotDefinition = Definition.Slots.FirstOrDefault();
Item.FillSlot(slotCanvas, slotDefinition);
```

*6-27 Vykreslení akceptovaných instancí z definice slotu.*

Tím jsme dokončili implementaci naší definice zobrazení. Pokud ji zaregistrujeme s využitím `ExtensionExporteru`, způsobem, jaký bude popsán v následující kapitole 6.5, bude možné ve schématu kompozice zobrazit *instance* reprezentující objekty typu `MEFEditor.Diagnostic`.

## 6.5 Registrace rozšíření

Každé rozšíření, které chceme v editoru použít, musí být nejprve registrováno. Tento přístup je odlišný od předchozí verze, kdy se rozšíření musela označovat patřičným atributem pro export a byla nahrána automaticky.

V současné verzi však chceme lépe oddělit implementaci rozšíření od nutnosti jejich použití v editoru. Můžeme tedy knihovny, kde jsou rozšíření implementovaná ponechat například pro použití uživatelem píšícím svá vlastní rozšíření. Z těchto knihoven pak může využít některé funkce, aniž by byl nucen využít i rozšíření, která jsou v těchto knihovnách již implementovaná. Registraci rozšíření je totiž možné vyčlenit do samostatné knihovny. Tento přístup byl i použit při implementaci *doporučených rozšíření*, která budou podrobněji popsána v kapitole 6.7.

Abychom si ukázali, jak se `ExtensionExport` používá, provedeme registraci všech uživatelských rozšíření, která jsme v předcházejících kapitolách implementovali. Zdrojový kód, který tuto registraci provede, je opět možné nalézt v solution *UserExtensions.sln* z přílohy [G].

Registrace se provádí exportem třídy odvozené od `ExtensionExport`. V ní definujeme metodu `Register`, kde můžeme zaregistrovat jednotlivá rozšíření. Třída připravená pro implementaci vypadá následovně:

```
//exportujeme třídu jako rozšíření editoru
[Export(typeof(ExtensionExport))]
public class UserExtensionsExport : ExtensionExport
{
  //registraci všech rozšíření provedeme v této metodě
  protected override void Register()
}
```

*6-28 Třída exportu uživatelských definic připravená pro implementaci.*

V průběhu registrace rozšíření máme možnost vypisovat ladící a chybové zprávy, které se uživateli zobrazí v logu editoru metodami `Message` a `Error`. Další metody, které třída `ExtensionExport` nabízí, umožňují registraci rozšíření. Jejich

použití můžeme vidět na následujícím zdrojovém kódu metody `Register`, kde exportujeme všechna ukázková rozšíření definovaná v předchozích kapitolách:

```
//zpráva, která se zobrazí v logu editoru
Message("Exporting UserExtensions");

//exportujeme poskytovatele assembly
ExportAssemblyFactory<string>(assemblyFactory);

//exportujeme pozměněnou definici pro string
ExportDefinition(new StringDefinition());

//exportujeme definici pro Diagnostic i s definici vykreslení
ExportDefinitionWithDrawing<DiagnosticDefinition>(
  (item) => new DiagnosticDrawing(item));
```

*6-29 Implementace metody Register, která exportuje ukázková rozšíření.*

Všimněme si metody `assemblyFactory`, pomocí které chceme exportovat poskytovatele assembly. V této metodě otestujeme, zda má cesta k assembly patřičný formát a podle ní případně vytvoříme poskytovatele assembly. Její implementaci můžeme vidět na následujícím obrázku:

```
private AssemblyProvider assemblyFactory(string path)
{
   //zkontrolujeme, zda cesta má příponu .test
   var isTestAssembly = Path.GetExtension(path) == ".test";
   if (!isTestAssembly)
     //cesta má jiný tvar, než požadujeme
     return null;

     //vytvoříme poskytovatele assembly
     return new SimpleAssemblyProvider(path);
}
```

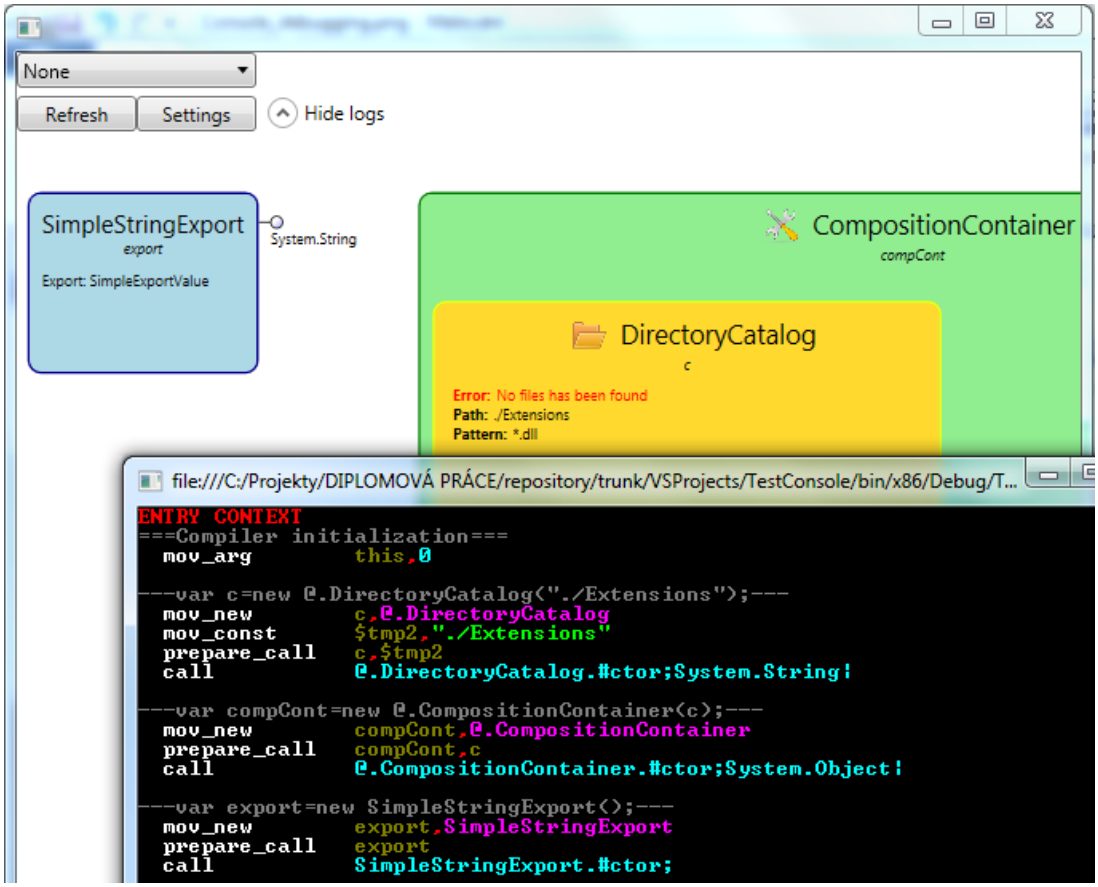
*6-30 Implementace metody pro export poskytovatele assembly.*

Tímto máme dokončenou implementaci exportu ukázkových uživatelských rozšíření. Pokud knihovnu s tímto exportem umístíme do složky *rozšíření editoru*, budou definovaná rozšíření při spuštění editoru načtena.

## 6.6 Ladění mimo Visual Studio

Abychom si ukázali, že je možné spouštět prostředí editoru i jiným způsobem, než jako plugin ve *Visual Studiu*, ukážeme si využití ladící aplikace dostupné v solution *TestConsole.sln* z přílohy [C]. Cílem této aplikace je poskytnutí stejných nástrojů, které máme v testovacím frameworku popsaném v kapitole 4.6. Navíc ale umožňuje zobrazení analyzačních instrukcí přeložených metod v přehledné konzoli a také nám dovoluje spouštět uživatelské rozhraní editoru.

Výstup konzole i uživatelského rozhraní můžeme vidět na následujícím obrázku:




*6-31 Ukázka uživatelského rozhraní editoru (vzadu) spuštěného mimo Visual Studio a konzole (vpředu) s ladícími informacemi o analyzovaných metodách.*

Jelikož je konzolová aplikace určená pro pomoc při vývoji rozšíření editoru, je způsob jejího použití založen na přímých úpravách zdrojových kódů. Díky tomu nepřijdeme o možnost ladění za pomoci běžného debuggeru *Visual Studia*, jako kdybychom načítali vyvíjená rozšíření v podobě zkompilovaných knihoven.

V této kapitole si předvedeme, jak projekt s konzolovou aplikací použít na ladění rozšiřující knihovny implementované v předcházejících kapitolách. Veškerý kód, který zde naimplementujeme je již dostupný v solution *TestConsole.sln* z přílohy [C].

Pro náš příklad budeme testovat, zda se *instance* reprezentující objekt typu `Diagnostic` zobrazí ve schématu kompozice a zda bude nabízet patřičné editace. Implementujeme tedy metodu, která patřičný test provede. K dispozici máme stejné nástroje, které jsou dostupné v testovacím frameworku popsaném v kapitole 4.6. Místo spouštění testu ale budeme v testovací metodě vytvářet objekt `TestingAssembly`.

Abychom mohli v projektu konzolové aplikace použít třídy implementované v rozšiřující knihovně, musíme na ni přidat referenci. Poté do statické třídy `TestCases` přidáme testovací metodu `TestExtensions`. Zde vytvoříme jednoduchou komponentu spolu s instancí testované `DiagnosticDefinition`.

Testovací metoda bude vypadat následovně:

```
static internal TestingAssembly TestExtensions()
{
  //definujeme zdrojový kód, ve kterém použijeme
  //testovanou typovou definici
  return AssemblyUtils.Run(@"
    var export=new SimpleStringExport();
    var diagnostic=new MEFEditor.Diagnostic();
    diagnostic.Start();
    diagnostic.Accept();
    diagnostic.Stop();
  ")
    //do Runtime přidáme testovanou definici
    //i s odpovídající definicí pro vykreslení
    .AddToRuntime<DiagnosticDefinition, DiagnosticDrawing>()
    //pro potřeby testu také potřebujeme nějakou komponentu
    .AddToRuntime<SimpleStringExport>()
  ;
}
```

*6-32 Test typové definice DiagnosticDefinition pro použití mimo Visual Studio.*

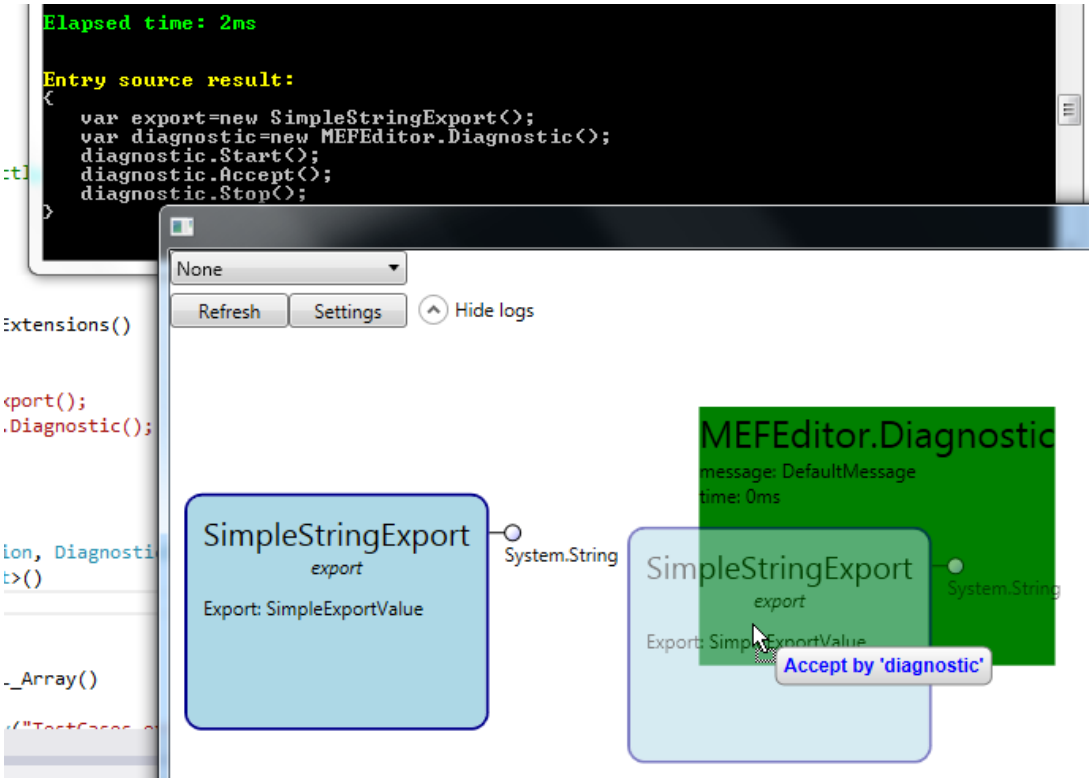
Nyní nám zbývá v metodě `Program.Main` nastavit spuštění analýzy na `TestingAssembly`. Provedení je patrné na následujícím obrázku:

```
public static void Main()
{
  //nejprve vytvoříme testovací assembly,
  //a nastavíme podmínky testu rozšíření
  var testAssembly = TestCases.TestExtensions();
  //spustíme test a zobrazíme výsledek
  DisplayTestResult(testAssembly);
}
```

*6-33 Metoda Main, která spustí test rozšíření.*

Po spuštění projektu se zobrazí okno s konzolí, která vypisuje informace o proměnných, instrukcích metod i zdrojových kódech. Pokud je v průběhu analýzy nalezena instance, která má být vykreslena ve schématu kompozice, otevře se navíc okno s uživatelským rozhraním editoru. V něm je možné provádět běžné editace. Jejich dopad na zdrojový kód pak můžeme sledovat v konzoly.

Výsledek můžeme vidět na následujícím obrázku:




*6-34 Test editací nabízených DiagnosticDefinition v konzolové aplikaci.*

## 6.7 Doporučená rozšíření

V rámci této práce byla pro editor implementována *doporučená rozšíření*, která umožňují nasazení editoru v projektech psaných jazykem C#. Pro podporu vývojových konfigurací, ve kterých je kompoziční algoritmus definován ve zkompilovaných assembly, obsahují doporučená rozšíření podporu pro jejich analýzu.

*Doporučená rozšíření* je možné vyzkoušet v solution *ExtensionsTests.sln* z přílohy [G]. Jsou zde příklady zdrojových kódů pro záludné situace, se kterými se dokáží vypořádat *doporučená rozšíření*.

V následujících kapitolách si popíšeme vlastnosti jednotlivých rozšíření. Nejprve se podíváme na knihovny, které byly spolu s doporučenými rozšířeními implementovány. Tyto knihovny obsahují množství předpřipravených tříd, které usnadní další rozšiřování. Jejich stručný popis je v kapitole 6.7.1. V dalších kapitolách pak budou popsána jednotlivá rozšíření.

### 6.7.1 Knihovny doporučených rozšíření

Na rozdíl od *doporučených rozšíření* v předchozí verzi editoru jsou současná *doporučená rozšíření* rozdělena do několika knihoven. Díky tomu je možné využívat služby implementované v rámci *doporučených rozšíření*, aniž by bylo nutné nahrávat rozšíření, která obsahují. V následujícím seznamu si popíšeme význam jednotlivých knihoven:

- *RecommendedExtensions.Core.dll* – Zde jsou implementovány všechny služby *doporučených rozšíření*, které je možné využít i pro uživatelská rozšíření. Služby zahrnují překladače pro C#, CIL, poskytovatel assembly z projektů *Visual Studia*, simulaci kompozičního algoritmu a nástroje pro vykreslování. Dále také veškeré *typové definice* a definice vykreslení nabízené *doporučenými rozšířeními*.
- *RecommendedExtensions.TypeDefinitions.dll* – Obsahuje export *typových definic*. Po odstranění této knihovny ze *složky rozšíření* je možné použít vlastní *typové definice* pro analýzu MEF.
- *RecommendedExtensions.DrawingDefinitions.dll* – Zde jsou exportovány všechny definice zobrazení. Po odstranění této knihovny ze *složky rozšíření* je možné použít vlastní definice zobrazení MEF objektů.
- *RecommendedExtensions.AssemblyProviders.dll* – Obsahuje export poskytovatelů assembly. Po odstranění této knihovny ze *složky rozšíření* je možné využít vlastní poskytovatele assembly pro C# a CIL.

Zdrojové kódy těchto knihoven je možné nalézt v solution *RecommendedExtensions.sln* dostupném v příloze [B].

### 6.7.2 Rozšíření pro poskytování assembly

*Doporučená rozšíření* obsahují poskytovatele assembly pro získávání assembly ze C# projektů otevřených ve Visual Studiu a také pro assembly ze zkompilovaných knihoven. V této kapitole si popíšeme jejich vlastnosti.

#### Poskytovatel C# assembly

Poskytování assembly z otevřeného projektu *Visual Studia* je možné rozdělit do dvou částí. První částí je vyhledávání typů, komponent a metod v hierarchické struktuře objektů `CodeElement` popsané v kapitole 3.3. Problematické jsou hlavně nedeterministické změny, kterým tato struktura podléhá na základě uživatelských zásahů. Poskytovatel assembly tyto změny sleduje a na jejich základě dokáže editoru poskytovat informace pro překreslení schématu kompozice.

Druhou částí poskytovatele assembly je překladač, který dokáže přeložit obvyklé konstrukce jazyka C# do *analyzačních instrukcí*.

Jmenovitě jsou to:

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

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

#### Poskytovatel zkompilovaných assembly

K vyhledávání komponent, typů a jejich metod ze zkompilovaných knihoven využíváme služby knihovny `Mono.Cecil` [8]. Nástroje této knihovny dokážou načíst

assembly, aniž by musela být nahrána do aplikační domény. To je výhodné hlavně kvůli paměťové efektivitě, kdy nahranou assembly by již nebylo možné uvolnit z paměti.

Klíčovou součástí poskytovatele zkompilovaných assembly je překlad instrukcí CIL do analyzačních instrukcí. Vzhledem k množství instrukcí, které CIL definuje, je jejich kompletní pokrytí nad rámec této práce. Podporované jsou nejběžnější z nich, které umožňují analýzu kompozičních algoritmů. Jmenovitě jsou to následující instrukce:

- Nahrávání argumentů na zásobník – *ldarg.0, ldarg.1, ldarg.2, ldarg.3, ldarg.s, ldarg*
- Nahrávání lokálních proměnných na zásobník – *ldloc.0, ldloc.1, ldloc.2, ldloc.3, ldloc.s, ldloc*
- Ukládání do lokálních proměnných ze zásobníku – *stloc.0, stloc.1, stloc.2, stloc.3, stloc.s, stloc*
- Nahrávání číselných konstant na zásobník – *ldc.i4.m1, ldc.i4.0, ldc.i4.1, ldc.i4.2, ldc.i4.3, ldc.i4.4, ldc.i4.5, ldc.i4.6, ldc.i4.7, ldc.i4.8, ldc.i4.s, ldc.i4, ldc.i8*
- Instrukce volání – *call, calli, callvirt, ret*
- Instrukce řízení běhu programu – *br.s, brtrue.s, blt.s, nop*
- Matematické operace – *add, add.ovf, add.ovf.un, clt, clt.un*
- Operace s objekty – *newobj, box, ldfld, stfld, ldsfld, stsfld*
- Operace s poli – *newarr, ldelem, ldelem.i1, ldelem.u1, ldelem.i2, ldelem.u2, ldelem.i4, ldelem.u4, ldelem.i8, ldelem.i, ldelem.r4, ldelem.r8, stelem.i, stelem.i1, stelem.i2, stelem.i4, stelem.i8, stelem.r4, stelem.r8, stelem*
- Operace se zásobníkem – *ldstr, ldtoken, dup, pop*

Změny zkompilovaných assembly nejsou tak časté, jak ve zdrojových kódech, proto nemusí *poskytovatel assembly* kontrolovat zda nedochází ke změnám metod použitých v průběhu interpretace. Přesto se však může stát, že dojde k přepsání zkompilované assembly jinou verzí. Tyto případy dokáže poskytovatel assembly rozpoznávat a upozorní editor, který dokáže novou verzi nahrát.

### 6.7.3 Rozšiřující typové definice

Editace nabízené editorem závisí na dostupných rozšiřujících typových definicích. Z tohoto důvodu obsahují doporučená rozšíření množství typových definic. Nejdůležitější z nich jsou typové definice, zaměřené na práci s MEF. Proto si uvedeme jejich seznam:

| **Typová definice** | **Implementované vlastnosti** |
|---|---|
| `AggregateCatalog` | • **Accept/Exclude** editace na `ComposablePartCatalog` |
| `TypeCatalog` | • **Add component type** – přidání typu ze seznamu typů dostupných komponent<br>• **Exclude from TypeCatalog** – odstranění typu přítomného v katalogu |
| `DirectoryCatalog` | • **Set path** – nastavení výchozí cesty pro vyhledávání knihoven<br>• **Set pattern** – nastavení vyhledávacího vzoru |

| | |
|---|---|
| | • **Open folder** – otevře aktuálně nastavenou cestu pro vyhledávání knihoven<br>• **Sledování změn ve složce** – projevuje se překreslením schématu při změně složky pro vyhledání komponent |
| `CompositionContainer` | • **Accept/Exclude** editace na komponenty<br>• **Accept/Exclude** editace na `ComposablePartCatalog`<br>• 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<br>• Zobrazuje informace o načtené assembly |

*Tabulka implementovaných typových definic důležitých z hlediska MEF*

Všechny tyto *typové definice* navíc přidávají globální editaci na vytvoření objektu patřičného typu do kontextového menu globálních 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 *RecommendedExtensions.Core.csproj* ze solution *RecommendedExtensions.sln* z přílohy [B].

### 6.7.4 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.

Pro vykreslování komponent je implementována defaultní definice zobrazení, využívající rozhraní editoru pro vytváření zobrazení importů a exportů. Spojení mezi těmito komponentami proto může ovlivňovat libovolná rozšiřující *typová definice*. Příkladem může být *typová definice* pro `CompositionContainer`, které pomocí spojnic importů a exportů znázorňuje vztahy kompozice.

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.

# 7 Závěr

V úvodních kapitolách jsme uvedli cíle týkající se rozšíření předchozí verze MEF editoru pocházejícího od stejného autora, jako je tato práce. V následující kapitole si popíšeme, jak se nám je podařilo splnit.

### Funkční požadavky

*1) Analýza kompozice definovaná ve zkompilovaných assembly*

V rámci *doporučených rozšíření* byl naimplementován poskytovatel zkompilovaných assembly. Díky tomu je možné provádět jejich analýzu.

*2) Schopnost analyzovat aplikaci v konfiguraci REA*

Mapování assembly ve spojení s analýzou zkompilovaných assembly umožňuje analýzu konfigurace REA.

*3) Provádění editací kompozičního algoritmu ve zdrojových kódech*

Analyzační instrukce spolu s transformacemi, které nad nimi analyzační knihovna nabízí, poskytují základ pro tvorbu editací. Využití tohoto základu typovými definicemi a poskytovateli assembly umožňuje pohodlnou editaci zdrojových kódů. Editor je navíc snadné upravit na poskytování editací kompozičního algoritmu ve zkompilovaných assembly.

*4) Vylepšení způsobu vykreslování schématu kompozice*

V předchozí verzi editoru mohlo docházet k zakrývání zobrazených položek schématu kompozice a k protínání položek spojnicemi. Současná verze editoru obsahuje algoritmy, které zamezují překrývání položek. Také byl odstraněn problém s protínáním položek spojnicemi, v případech, kdy to bylo možné.

*5) Integrace editoru do Microsoft Visual Studio 2012*

Editor je možné přidat jako rozšíření Microsoft Visual Studio 2012 ve formě pluginu. Navíc je editor zpětně kompatibilní s verzí Microsoft Visual Studio 2010.

*6) Poskytování schématu kompozice ze zdrojových kódů aplikace otevřené ve Visual Studiu*

Díky napojení editoru ve formě pluginu na služby Visual Studia, může editor analyzovat zdrojové kódy otevřené aplikace. Tato analýza poskytuje dostatečné informace pro zobrazení schématu kompozice a nabízení různých editací.

*7) Podpora hlavních konceptů MEF*

V rámci doporučených rozšíření byly implementovány typové definice a definice zobrazení pro typy, které se týkají hlavních konceptů MEF. Díky tomu je dokáže editor zobrazovat v kompozičním schématu, nabízet nad nimi editace a zobrazovat chybová hlášení. Editor je navíc koncipován s ohledem na rozšiřitelnost o vedlejší koncepty MEF.

*8) Podpora jazyka C#*

V rámci doporučených rozšíření byl implementován poskytovatel assembly, který dokáže přeložit obvyklé konstrukce jazyka C# do analyzačních instrukcí. Díky tomu

může editor nad zdrojovými kódy psanými v jazyce C# provádět analýzu a patřičné editace.

9) *Uživatelská rozšiřitelnost editoru*

Editor je navržen s ohledem na snadnou uživatelskou rozšiřitelnost. Ta byla demonstrována v ukázkových rozšířeních, kde jsme implementací provedli uživatele krok za krokem. Editor je rozšiřitelný o podporu dalších jazyků a typů assembly pomocí poskytovatelů assembly. Možnosti analýzy jsou rozšiřitelné přidáváním rozšiřujících typových definic. Rozšiřitelné je i vykreslování ve schématu kompozice, díky možnosti přidávat uživatelské definice zobrazení.

## Kvalitativní požadavky

Ze zkušeností s vývojem předchozí verze editoru jsme do cílů zahrnuli i následující kvalitativní požadavky. Shrneme si, jak se nám jich v rámci této práce podařilo dosáhnout:

A) *Části editoru budou testovatelné pomocí frameworku unit testů*

Architektura současné verze editoru podporuje testování všech důležitých částí. V průběhu implementace bylo napsáno množství testů, které ověřují správné fungování editoru i doporučených rozšíření.

B) *Pro vývoj rozšíření bude dostupná konzole s přehlednými ladícími výpisy*

Pro usnadnění vývoje rozšíření, ale i celého editoru byl vyvinut projekt v jazyce C#. Umožňuje využívat veškeré nástroje testovacího frameworku a poskytovat ladící výpisy s přehledným zvýrazněním syntaxe v konzoli.

C) *Editor bude možné spustit i mimo Microsoft Visual Studio*

Knihovny implementované pro editor v rámci této práce jsou většinou nezávislé na Visual Studiu. Díky tomu je možné editor spouštět i mimo něj. Příkladem, kdy je editor spouštěn mimo Visual Studio může být aplikace s ladicí konzolí, která také dokáže zobrazit uživatelské rozhraní editoru.

## Univerzálnost principu analýzy

Vzhledem k přílišnému rozsahu frameworku MEF a podporovaných jazyků, které by nebylo možné v rámci této práce plně pokrýt, byl editor implementován s velkým důrazem na rozšiřitelnost. Pomocí uživatelských rozšířením je tak možné editor rozšířit o další koncepty MEF, ale i o podporu dalších jazyků.

Díky značné rozšiřitelnosti je také možné editor snadno specializovat pro použití v konkrétním projektu. Můžeme například definovat specifické zobrazení pro různé komponenty, tak aby bylo schéma kompozice názornější. Případně můžeme nechat zobrazit třídy ve schématu kompozice, které s MEF vůbec nesouvisí. Takto zobrazené třídy by opět mohly nabízet různé druhy editací.

Z těchto důvodů není využití editoru omezené pouze na ulehčení práce s MEF. Díky univerzálnímu způsobu analýzy projektů a nabízení editací je možné i současnou verzi editoru rozšířit o možnosti zobrazení a editace libovolných objektů a vztahů mezi nimi.

Další specializace editoru na využití pro jiný framework nebo knihovnu, než je MEF, by spočívala v drobných úpravách. Příkladem takových úprav může být změna vyhledávání startovní metody analýzy (kde pro MEF vyhledáváme *composition*

*pointy*) nebo změna globálních editací zaměřených na podporu komponent (například editace na vytvoření nové komponenty).

### Shrnutí

V rámci této práce jsme provedli téměř kompletní reimplementaci předchozí verze editoru. Editor je v současné verzi použitelný jako plugin do *Microsoft Visual Studia 2012* i *Microsoft Visual Studia 2010*. Stejně jako v předchozí verzi umožňuje editor analyzovat a vykreslit schéma kompozice získané analýzou zdrojových kódů z rozpracovaných projektů. Navíc byla přidána podpora pro schopnost analýzy instrukcí ve zkompilovaných knihovnách. Další rozšíření se týká konfigurace REA, která je současnou verzí pokryta a usnadňuje tak vývoj pluginů ve Visual Studiu do již existujících aplikací.

V editoru zůstala zachována schopnost překreslit schéma kompozice při zaznamenání změn ve zdrojovém kódu i při změnách souborů knihoven a složek, které jsou při kompozici čteny.

I současná verze editoru je široce rozšiřitelná o *typové definice*, které definují schopnosti analýzy. Stejně tak je snadné pomocí uživatelských rozšíření kompletně změnit vzhled zobrazeného schématu kompozice. V předchozí verzi editoru bylo možné přidat podporu nových jazyků dodáním patřičných parserů a interpretů. V současné verzi se přidání podpory nového jazyka provádí pomocí poskytovatelů assembly, což uživateli dává mnohem širší možnosti použití ve srovnání s parsery a interprety.

V knihovně *RecommendedExtensions.AssemblyProviders.dll* je implementován poskytovatel pro assembly načtené z otevřených projektů ve Visual Studiu. V této knihovně také najdeme implementaci poskytovatele pro zkompilované assembly.

Jazykové moduly implementované v rámci této práce se nacházejí v knihovně *RecommendedExtensions.Core.dll*. Spolu s knihovnou *RecommendedExtensions.TypeDefinitions.dll*, která obsahuje implementace typových definic pro důležité MEF objekty umožňují *doporučená rozšíření* analyzovat projekty C# i zkompilované instrukce CIL. Pro definici zobrazení schématu kompozice pak slouží poslední knihovna *doporučených rozšíření RecommendedExtensions.DrawingDefinitions.dll*.

### Další vývoj

Přirozeným pokračováním vývoje editoru je implementace rozšíření, která pokryjí vedlejší koncepty MEF uvedené v kapitole 1.2. Pro potřeby rozsáhlejších projektů a zpřesnění analýzy by bylo také výhodné rozšířit podporu jazyka C# i instrukcí CIL a přidat podporu pro další .NET jazyky.

Univerzálnost analýzy editoru ho však neomezuje pouze na použití pro MEF. Bylo by proto možné editor specializovat na jiný framework či knihovnu tak, aby pomáhal při vývoji aplikací, které je využívají.

# 8 Seznam použitých zdrojů


[1] Editor komponentových architektur pro MEF, bakalářská práce, řešitel: Miroslav Vodolán, školitel: Mgr. Pavel Ježek, Ph.D., 2012

[2] Pojednání o principech uzavřenosti objektů, MSDN, webová adresa: *http://blogs.msdn.com/b/alfredth/archive/2011/07/26/encapsulation-an-introduction.aspx*

[3] Stránka vývojového prostředí Microsoft Visual Studio, webová adresa: *http://www.visualstudio.com/cs-cz/visual-studio-homepage-vs.aspx*

[4] Nástroj Code Contracts, Microsoft Research, webová adresa: *http://research.microsoft.com/en-us/projects/contracts/*

[5] Stránka webového prohlížeče Google Chrome, webová adresa: *http://www.google.cz/intl/cs/chrome/browser/*

[6] Představení operačního systému HelenOS, webová adresa: *http://www.helenos.org/*

[7] Úvod do použití formátu DGML pro tvorbu grafů, MSDN, webová adresa: *http://blogs.msdn.com/b/camerons/archive/2008/12/16/introduction-to-directed-graph-markup-language-dgml.aspx*

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

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

[10] Popis Dijkstrova algoritmu, Alogoritmy.net, webová adresa: *http://www.algoritmy.net/article/5108/Dijkstruv-algoritmus*

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

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

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

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

[15] Stránka projektu MEF Visualizer Tool, Codeplex, webová adresa: *http://mefvisualizer.codeplex.com/*

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

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

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

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

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

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

[22] Microsoft Visual Studio SDK, Microsoft, webová adresa: *http://www.microsoft.com/en-us/download/details.aspx?id=30668*

[23] .NET Reflection, MSDN, webová adresa: *http://msdn.microsoft.com/cs-cz/library/gg145033(v=vs.110).aspx*
[24] Windows Presentation Foundation, MSDN, webová adresa: *http://msdn.microsoft.com/en-us/library/ms754130(v=vs.110).aspx*
[25] Sandcastle Help File Builder, CodePlex, webová adresa: *http://shfb.codeplex.com/*

# 9 Přílohy

Veškeré uvedené přílohy je možné nalézt na CD přiloženém k této diplomové práci a také v repositáři https://github.com/m9ra/MEFEditor_v2.0.

**A. Implementace Enhanced MEF Component Architecture Editor**
Je umístěna ve složce `Implementace/Editor`

Obsahuje solution pro *Visual Studio 2012*, ve kterém byl implementován a zkompilován náš editor ve formě pluginu pro *Visual Studio*. Součástí solution jsou okomentované zdrojové kódy.

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

Obsahuje solution pro *Visual Studio 2012*, ve kterém byly implementovány a zkompilovány knihovny *doporučených rozšíření*. Součástí solution jsou okomentované zdrojové kódy s resources soubory.

**C. Implementace konzolové aplikace pro ladění rozšíření**
Příloha je umístěna ve složce `Implementace/LadiciKonzole`

Obsahuje solution pro *Visual Studio 2012*, ve kterém byla implementována konzole pro ladění uživatelských rozšíření. Použití této aplikace spočívá v uživatelských úpravách jejích zdrojových kódů, proto není součástí příloh její zkompilovaná verze.

**D. Implementace testů editoru a doporučených rozšíření**
Příloha je umístěna ve složce `Implementace/Testy`

Obsahuje solution pro *Visual Studio 2012*, ve kterém jsou implementovány automatické testy knihoven editoru a *doporučených rozšíření*.

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

Obsahuje soubor *Enhanced_MEF_Component_Architecture_Editor.vsix*, určený pro instalaci editoru formou pluginu do *Visual Studia 2010* a *2012*. Dále obsahuje knihovny *doporučených rozšíření* popsané v kapitole 6.7.1. Přítomnost knihoven ve složce rozšíření editoru určuje, zda budou do editoru načtena *doporučená rozšíření*, jak je popsáno v kapitole 5.1.

**F. Dokumentace vygenerovaná ze zdrojových kódů editoru**

Příloha je umístěna ve složce `Dokumentace`

Obsahuje automaticky generovanou dokumentaci získanou ze zdrojových kódů pomocí nástroje *Sandcastle Help File Builder* [25].

Dokumentace se skládá z následujících souborů:

- *Enahnced_MEF_Component_Architecture_Editor.chm*
  Dokumentace typů použitých v implementaci editoru.
- *Recommended_Extensions_Documentation.chm*
  Dokumentace typů použitých v implementaci *doporučených rozšíření*.
- *Test_Console_Documentation.chm*
  Dokumentace typů využívaných pro ladění uživatelských rozšíření.

**G. Ukázkové projekty na použití editoru**

Příloha je umístěna ve složce `Příklady`

Obsahuje projekty pro *Visual Studio 2012*, které byly použity v příkladech kapitol 5.3.1 a 5.3.2. Dále obsahuje ukázková rozšíření implementovaná v příkladech kapitoly 6.

**H. Elektronická verze této práce**

Příloha je umístěna ve složce `Dokumentace`

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