**Obsah**

**Úvod**

**Cíle projektu**

**1. Analýza**

**1.1 Komponentová architektura s využitím MEF**

Rozebereme typickou strukturu aplikace při použití MEF. Publikovány pouze interface ->nezávislost na implementaci. Nahráváme komponenty implementující určité interface = části aplikace řešící nějakou úlohu -> implementací může být více, což umožňuje rozšiřitelnost.

Vzhledem k tomu, že komponenty pocházejí z různých zdrojů je vhodné tyto zdroje zobrazit ve schématu, důležité také spojnice importy ->exporty,.. dává přirozené znázornění schématu kompozice

**1.2 Editace schématu kompozice**

Dle popsaného schématu zvážíme možnost jaké editace by uživatel potřeboval. (přesunování komponent, měnění vlastností katalogů…). Úvaha o způsobu provádění editací (speciální konfigurační soubor, přidané knihovny,..)->dospějeme k nejvhodnějšímu řešení přímou úpravou zdrojových kódů. Z toho vyplývá nutnost umět zdrojové kódy analyzovat a upravovat.

**1.3 Analýza aplikace**

Úvahy o tom, jakým způsobem lze v aplikaci zjistit kompoziční schéma (proč nevhodné zkoušet analyzovat spuštěnou aplikaci), dospějeme k potřebě interpretovat metody pro zjištění stavu objektů ovlivňujících kompozici. Definice compositionpoints + v jakém stavu se composition point spouští.

**1.4 Interpretace metod**

Interpretovat je potřeba metody z různých zdrojů (zdrojový kód různých jazyků, MSIL) -> výhoda předzpracování parsováním oproti přímé interpretaci (cachování, jednotný interpreter). Pro samotné interpretování potřeba objektového modelu -> úvahy proč nelze využít typový systém .NETu.

**1.5 Typový systém**

Vzhledem k potřebě objektového modelu -> typový systém, nad kterým poběží. Nastínit problém generických tříd. Virtuální třídy. Možnost pružně reagovat na změny vyvolané uživatelem ( různá stádia rozpracovanosti typů ). Možnost pracovat s několika assemblies současně.

**1.5 Objektový model**

Výhody vlastního objektového modelu - máme nad objekty plnou kontrolu -> měření instrumentace (řeší halting problém), dokážeme z nich získat stav.

Musí umožnit statické třídy, volání metod. Nastíněna myšlenka oddělené deklarace od implementace pro virtuální/abstraktní metody.

Pro reprezentaci primitivních typů bude pracovat s .NET objekty, které musí zpřístupnit přímo. Definovat Instance jako objekt Objektového modelu

**1.6 Vytváření editací**

Nutnost umožnit editace v metodách, nezávisle na jejich zdroji ->Návrh rozhraní, které k tomu budeme potřebovat.

Některé editace souvisejí s konkrétní Instancí, některé ne.

Úvahy o tom, že editace souvisejí s typem editovaného objektu => Editace budou definovány v definicích typů, které budou překrývat jejich skutečné implementace. Díky rozšiřitelnosti o definice typů pak můžeme libovolně upravit nabízené editace.

**1.7 Vykreslování instancí**

Vhodnost zobrazení instance na základě jejího typu. Nastínit problém s rozpoznáním „logicky stejné“ instance v různých spuštěních jednoho CompositionPoint. Rozšiřitelnost o definice zobrazení umožňuje specializovat schéma kompozice pro konkrétní projekt

**2. Rozšiřitelnost Microsoft Visual Studia 2010**

Popsány třídy, které budou využívány pro spojení s Microsoft Visual Studiem + „nástrahy“ na které je třeba dát pozor

**2.1. Projekt VsPackage**

Popis základního projektu vytvořeného pro psaní pluginů do Visual studia

Přehled služeb, které dává Visual Studio k dispozici připojeným pluginům

**2.2. EnvDTE**

Základní objekt pro interakci s Visual studiem

**2.3. CodeElement**

Základní objekt pro přístup k sémantickým stromům postaveným nad zdrojovými kódy a projekty, které dává Visual Studio k dispozici

Možnosti jak k nim lze přistupovat z pluginů

**2.4. Události vyvolané uživatelem**

O událostech, které uživatel při své činnosti vyvolává v rámci EnvDTE

**3. Implementace editoru**

**3.1 Struktura**

-vztahy mezi parsery, interpretery, objektovým modelem a jak spolu komunikují

-jakým způsobem se nahrávají rozšíření

**3.2 Typový systém**

Jak je naiplementován typový systém, jak jsou řešeny assemblies, a reprezentace typů

**3.2.1 Reprezentace assemblies**

Jakým způsobem jsou reprezentovány assemblies získané z rozepsaného projektu, z referencí a z .dll knihoven.

Assembly Runtime, pro předefinováni chování typů

**3.2.1.1 UsrAssembly**

**3.2.1.2 MSILAssembly**

**3.2.1.3 Assembly Runtime**

**3.2.1.4 AssemblyLoader**

**3.2.2 Reprezentace typů**

O třech stádiích rozpracovanosti InternalType

**3.2.2.1 TypeTicket**

K čemu slouží TypeTicket, jeho výhody pro typy nad zdrojovými kódy a nevhodností pro použití s MSIL knihovnami

**3.2.2.2 ITypeDefinition**

K čemu slouží typové definice, a implementace ITypeDefinition použité v editoru

**3.2.2.3 InternalType**

Implementace „virtuálního typu“ typového systému. Jakým způsobem se vytváří a co poskytuje

**3.2.2.4 Dědičnost**

Jak je řešena dědičnost mezi třídami ->přes InternalType

**3.2.2.5 Generické typy**

Jak se pracuje s generickými typy

**3.2.2.6 Metody**

Způsob reprezentace metod, IMethodInfo, IMethod (oddělená deklarace od implementace, kvůli virtuálním metodám)

**3.2.3 TypesManager**

Třída řešící vytváření a invalidování typů, co vše pro assemblies zajišťuje, vytváření InternalTypes, generické typy

**3.3 Objektový model**

Jakým způsobem jsou vyřešeny virtuální objekty

**3.3.1 Instance**

Popis třídy, reprezentující virtuální objekt

**3.3.2 Volání metod na instancích**

**3.3.3 Vytváření instancí**

Instance se vytvářejí pouze v rámci spuštění nějakého CompositionPoint – pro ten je limitován počet vytvořených instancí. Nastíněny potřeby různých způsobů pro vytváření instancí + kde se používají.

**3.3.3.1 Přímé vytvoření**

**3.3.3.2 Vytvoření voláním konstruktoru**

**3.3.3.3 Vytvoření injektováním ze zadaných dat**

**3.3.4 Sdílené instance**

Instance reprezentující statické třídy a jak je zajištěno jejich fungování

**3.3.5 Koncept dirty instancí**

Reprezentovat korektně veškeré typy v .NETu by bylo příliš výpočetně náročné, proto editor pracuje korektně pouze se „zajímavými“ typy. Aby však mohl pracovat s metodami, kde se vyskytují i jiné typy, pokusí se je reprezentovat pomocí Dirty instance – ta propaguje dirty příznak do všech instancí, které s ní nějak pracují.

**3.4 Reakce na události vyvolané uživatelem**

Práce se zdrojovými kódy je založená na CodeElement – je nutné reagovat na změny, které na nich uživatel provádí, jak se zjišťuje že nějaká změna nastala a jak je editorem zpracována

**3.4.1 Změna aktivního Solution**

Jak se řeší a detekuje zavření/otevření solution

**3.4.2 Změny v projektech aktivního Solution**

Při přidávání/odebírání/přejmenování projektů -> Není častá operace, může dojít k znovunačtení celého typového systému

**3.4.2 Přidání nového CodeElement**

**3.4.3 Odebrání CodeElement**

**3.4.4 Přejmenování CodeElement**

**3.4.5 Změna těla CodeElement**

**3.5 Systém závislostí**

-Díky výše zmíněným událostem je možné definovat různé cíle závislosti objektů, které ovlivňí jejich platnost ( platnost typových definic, assembly,..)

-Které objekty závislosti využívají a k čemu jsou dobré.

**3.5.1 Cíle závislostí**

Na čem mohou objekty záviset, a jak se detekují změny na těchto cílech

**3.5.1.1 Závislost na implementaci metody**

Změna těla metody ve zdrojovém kodu

**3.5.1.2 Závislost na signatuře**

Signatura typu a metod – může být vytvořen nový typ, zrušen typ,..

**3.5.1.3 Závislost na souborech**

Důležité pro obnovu DirectoryCatalogů a MSIL načtených assemblies

**3.5.1 Závislosti instrukcí v IInvokeInfo**

Na čem závisejí instrukce různých invokeInfo, výhoda parsování v několika krocích –při změně sémantiky se nemusí znovu parsovat syntaxe,..

**3.5.2 Závislosti typových definic**

Na čem mohou záviset typové definice a jak se řeší jejich invalidace

**3.5.3 Závislosti assemblies**

Na čem závisejí assemblies – hlídání změn na souborech a jak se provádí

**3.5.4 Závislosti schématu kompozice**

Jak se určuje, kdy je nutné obnovit schéma kompozice

**3.6 Editace**

Jakým způsobem jsou řešeny editace, rozdíl mezi editacemi, které souvisejí s konkrétní instancí a které ne

**3.6.1 ILanguageDefinition**

Popis, rozhraní, používané pro zápis editací do zdrojové metody (nezávisle na jazyku)

**3.6.2 Koncept EditTransformation**

Popis objektů, které zprostředkovávají editace a jejich použití v editoru, skládání transformací, transformace se netýká pouze syntaxe, ale také sémantiky

**3.6.3 Accept editace**

Jak fungují accept editace (umožňují přijmout instanci při Drop akci ve schématu)

**3.6.4 Remove editace**

Odstranění instance ze zdrojového kódu je provedeno přes Remove editaci

**3.6.5 Write editace**

Přepsání hodnot v argumentu funkce nebo přidání argumentu např. pro cestu v DirectoryCatalogu

**3.6.5 Editace poskytované instancí**

Editace týkající se hierarchie objektů (např. seznam obsahuje prvky) – lze definovat accepting a removing editace

**3.7 Komponentový model**

Jakým způsobem jsou spravovány komponenty, co způsobuje jejich přidávání/odebírání

**3.7.1 Reprezentace komponent**

**3.7.2 Vyhledávání komponent**

**3.7.3 Naplnění importů**

**3.8 Uživatelské rozhraní**

Nástin, jak probíhá zobrazování, schématu -> spuštění composition point, a zpracování získaných výsledků, obsluha události pro překreslení composition point

**3.8.1 Seznam dostupných CompositionPoints**

Jak je zajištěna aktuálnost seznamu compositionpoints

**3.8.2 Vykreslování schématu kompozice**

Průběh vykreslování – Workspace zobrazí explored instance bez rodiče, rekurzivně zobrazuje další instance

**3.8.2.1 Vykreslení instance**

Jak se určí definice pro nakreslení instance, co vše má k dispozici a jak se zobrazí

**3.8.2.1 Určování pozice zobrazených instancí**

Využití Instance.ID pro zobrazení instance na stejném místě napříč composition pointy

**3.8.2.2 Importy, exporty a jejich spojení**

Jak se vykreslují spojnice mezi importy, exporty, oznámení o nenaplněných importech.

**3.8.2 Zamykání editací**

Pokud je uživatelem změněn zdrojový kód, dojde k uzamčení možnosti editovat schéma kompozice za pomocic editoru, dokud není změna patřičně registrována (brání poškození zdrojových kódů)

**4. Uživatelská příručka**

**4.1 Popis rozhraní editoru**

Ovládací prvky editoru

**4.2 Použití editoru**

Tutoriál na vytvoření kompoziční architektury na konkrétním příkladě, doprovázen screeny z použití editoru

**4.3 Rozšiřitelnost editoru**

Celkový pohled na rozšiřitelnost editoru, jak je koncipována a co všechno umožňuje.

**4.3.1 Uživatelské parsery**

Ukázka rozšíření o vlastní parser

**4.3.2 Uživatelské interpretery**

Ukázka rozšíření o vlastní interpreter

**4.3.3 Uživatelské definice typů**

Ukázka rozšíření o vlastní definici typu s podporou různých editací. Navazuje na další kapitolu, kde bude k tomuto typu definováno zobrazení

**4.3.4 Uživatelské zobrazení objektů**

Příklad na vytvoření definice zobrazení vlastního objektu

**4.3.5 Standardní rozšíření**

Zmíněna standardní rozšíření a omezení, která z jejich přítomnosti vyplývají

**5. Závěr**

**6. Reference**