VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Podpora vestavného procesně funkcionálního jazyka v nástroji Visual Studio

Support of Embedded Process Functional Language in the Tool Visual Studio

2011 Daniel Lichovník

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 15.dubna 2011 ....

Chtěl bych poděkovat Ing. Marku Běhálkovi, Ph.D. za cenné připomínky, návrhy a doporučení při konzultacích.

Abstrakt

Tato diplomová práce se zabývá problematikou rozšíření Visual Studia o podporu nového programovacího jazyka. Její první část obecně přibližuje možnosti rozšíření tohoto IDE, a specificky se zaměřuje na problematiku implementace podpory nově zaváděných programovacích jazyků. Jsou zde analyzovány a naznačeny možnosti pro zavedení projektových šablon, připojení kompilátoru, podporu jazyka v editoru, a podporu ladění.

Ve druhé části pak jsou tyto poznatky přeneseny do praxe, kdy některé z těchto prvků jsou probrány podrobněji při demonstrování zavedení podpory funkcionálního jazyka e-PFL do prostředí Visual Studia 2010.

Klíčová slova: IDE, Visual Studio, rozšiřitelnost, MEF, podpora programovacího jazyka, editor VS, šablona projektu, značkování kódu, barvení syntaxe, Intellisense

Abstract

This thesis concentrates on the problems of Visual Studio extensibility in the area of support for new programming language. In the first part the IDE extension possibilities are considered with a focus on the implementation problems of support for newly introduced programming languages. Possible solutions for design and implementation of project templates, compilator binding, support in editor, and support for debugging are analyzed and described here.

In the second part these information are used in practical demonstration when some of the extensibility features are examined in depth during implementation of support for e-PFL functional language in Visual Studio 2010.

Keywords: IDE, Visual Studio, extensibility, MEF, programming language support, VS editor, project template, tagging, syntax highlighting, Intellisense

Obsah

1 Úvod **4**

2 Microsoft Visual Studio 2010 a jeho rozšiřitelnost **6**
- 2.1 Architektura Visual Studia . . . 6
- 2.2 Rozšiřitelnost (Extensibility) . . . 8
  - 2.2.1 MEF – Managed Extensibility Framework . . . 8
  - 2.2.2 Stručný náhled na rozšiřitelnost jiných IDE . . . 12

3 Podpora nového jazyka ve Visual Studiu 2010 **14**
- 3.1 Projekt . . . 14
  - 3.1.1 Šablony projektů . . . 15
  - 3.1.2 Struktura tříd pro generování projektů . . . 18
  - 3.1.3 Vlastnosti projektu . . . 20
- 3.2 Kompilátor . . . 22
  - 3.2.1 Připojení vlastního kompilátoru . . . 23
  - 3.2.2 Logika procesu překladu . . . 24
- 3.3 Rozšíření editoru . . . 25
  - 3.3.1 Struktura editoru VS . . . 26
  - 3.3.2 Implementace rozšíření editoru . . . 29
- 3.4 Debugger . . . 39

4 Implementace podpory jazyka e-PFL ve Visual Studiu **43**
- 4.1 Funkcionální jazyky . . . 43
- 4.2 Jazyk e-PFL . . . 44
- 4.3 Integrace jazyka e-PFL do prostředí Visual Studia . . . 44
  - 4.3.1 Struktura rozšíření . . . 44
  - 4.3.2 Správa projektů . . . 45
  - 4.3.3 Připojení kompilátoru . . . 47
  - 4.3.4 Rozšíření editoru o podporu e-PFL . . . 49
  - 4.3.5 Deployment . . . 57

5 Závěr **59**

6 Literatura **60**

Přílohy **61**

A Příloha 1: Gramatika jazyka e-PFL **62**

B Příloha 2: Obsah přiloženého CD a postup instalace **64**

Seznam obrázků

1 Architektura VSPackages a Services . . . 7
2 MEF model . . . 9
3 Struktura složek se šablonami . . . 16
4 Dialogové okno New Project . . . 17
5 Class diagram architektury pro generování projektů . . . 19
6 Class diagram třídy implementující property page . . . 21
7 Dialogové okno Vlastnosti projektu . . . 22
8 Class diagram třídy tasku + soubor *.targets* . . . 24
9 Logický náhled na editor VS . . . 27
10 Náhled na základní prvky text modelu . . . 28
11 Část hierarchie Content Types ve Visual Studiu . . . 30
12 Agregování klasifikátorů . . . 35
13 Náhled na čtyři funkce Intellisense . . . 36
14 MVC model Intellisense . . . 37
15 Schéma debuggeru VS . . . 40
16 Stavový diagram debugger session . . . 42
17 Ukázka práce analyzéru – označování textu symboly . . . 51
18 Class diagram struktury obecného Token taggeru . . . 52
19 Nastavení obsahu VSIX balíčku pro e-PFL . . . 58

Seznam výpisů zdrojového kódu

1 Deklarace exportů tříd 10
2 Deklarace exportu třídy jako rozhraní 10
3 Deklarace exportu properties 11
4 Deklarace exportu metod 11
5 Ukázka definice vlastních parametrů v šabloně *.vstemplate* 18
6 Definice a export content typu 30
7 Definice a export napojení na souborovou příponu 31
8 Definice a export klasifikačního typu 32
9 Definice a export formátování 33
10 Ukázka připojení agregátoru 35
11 Ukázka použití agregátoru 36
12 Defaultní obsah souboru Main.pfl 45
13 Ukázka šablony *.vstemplate* projektu Main Project Application 45
14 Ukázka definice cesty k souboru *.targets* ze souboru *.pflproj* 47
15 Ukázka algoritmu pro spuštění překladu 48
16 Ukázka přesunu a přejmenování přeloženého souboru do očekávané složky 48
17 Ukázka směrování výpisu kompilátoru do dialogu Output 49
18 Metoda *GetTags()* Token taggeru 52
19 Ukázka vytvoření agregátoru ve třídě *EPFL_ClassifierProvider* 53
20 Část metody *GetTags()* klasifikačního taggeru 53
21 Ukázka definice klasifikačního typu a definice jeho formátování 54
22 Metoda *GetTags()* Error taggeru 55
23 Ukázka víceřádkové funkce jazyka e-PFL před a po sbalení 56

1 Úvod

V raných fázích vývoje softwaru, kdy vývojáři vkládali své programy do počítačů pomocí děrných štítků, pásek apod., neexistovaly komplexní vývojové nástroje, které by tuto činnost usnadňovaly. Počítače zpracovaly a přeložily takové vstupy do svého vnitřního jazyka, pomocí něhož pak program spustily. S nástupem modernějších programovacích jazyků a nových přístupů k vývoji softwaru se objevily i vývojové nástroje pro zefektivnění, usnadnění a zrychlení práce programátorů. Ještě později se tyto nástroje začaly spojovat do jediného komplexního nástroje nazývaného integrované vývojové prostředí – Integrated Development Environment (IDE).

Společným rysem takovýchto nástrojů je především integrace editoru zdrojového kódu, překladače zdrojového kódu do strojového jazyka (compiling), nástroje pro spuštění a procházení kódu za běhu za účelem nalezení potencionálních chyb (debugging), a nástroje pro sestavení vyvíjeného programu do samostatně spustitelné aplikace a její nasazení (build, deployment). Všechny součásti vývojových prostředí zde mají jednotné uživatelské prostředí umožňující uživateli poměrně rychle se s prostředím sžít, snadnou orientaci a plynulou práci během celého procesu vývoje.

Historicky prvním takovým integrovaným prostředím byl program Maestro I, vyvinutý německou společností Softlab Munich v roce 1975. Později byl následován dalšími s různě pozměněnými přístupy a přidanými nástroji a službami podle použité technologie a požadavků trhu. Tento vývoj vedl až k současným moderním vývojovým prostředím, jak je známe dnes.

Velká spousta integrovaných vývojových prostředí je orientována na podporu vývoje v jednom programovacím jazyku se všemi jeho specifiky, další jsou orientovány na vizualizaci vývoje, tvorbu modelů s podporou běžných standardů, jiné například na podporu týmové spolupráce (JBuilder, Turbo C++, Delphi, VisualWorks, . . .).

Nejsofistikovanější moderní vývojová prostředí (Microsoft Visual Studio, Eclipse, NetBeans) nabízejí komplexní přístup, kdy nabízejí většinu z výše uvedeného, zpravidla standardně podporují vývoj ve více programovacích jazycích i platformách, a své možnosti dokáží i rozšiřovat. Takováto vývojová prostředí lze modulárně na míru upravovat pomocí různých maker, add-inů, balíčků apod. Rozšíření může jít cestou automatizace často prováděných úloh, přidávání nových funkcionalit, nebo rozšířením o podporu dalšího programovacího jazyka.

Protože vývoj rozšíření pro IDE bývá často komplexnější netriviální programátorskou úlohou, bývá ze strany výrobců podporován pomocí dodatečných vývojových nástrojů a frameworků, obvykle nazývaných Software Development Kit (SDK).

Tato práce se zabývá možnostmi rozšíření vývojového prostředí Microsoft Visual Studio 2010 o nový programovací jazyk. Nastiňuje implementaci některých možností podpory tohoto jazyka a demonstruje ji ve své druhé části na implementaci vybraných možností podpory pro funkcionální jazyk e-PFL.

Poznámka: V běžné praxi se většina programátorů s úlohou rozšíření IDE o podporu nového jazyka nesetkává. Navíc, v době vzniku této práce je Microsoft Visual Studio ve verzi 2010 poměrně krátce na trhu. Zřejmě z obou těchto důvodů neexistuje mnoho pub-

likací zabývajících se tímto tématem. Většina citací a odkazů v této práci proto pochází z internetových zdrojů, ať už z Microsoft MSDN Library, z článků na serverech zabývajících se touto nebo podobnou tematikou, nebo z diskuzních fór.

2 Microsoft Visual Studio 2010 a jeho rozšiřitelnost

Microsoft Visual Studio je integrované vývojové prostředí vyvíjené od roku 1997 firmou Microsoft. Jde o komplexní systém umožňující vývoj různých typů aplikací (konsolové aplikace, Windows aplikace, webové aplikace, atd.). Primárně je zaměřen na vývoj aplikací nad proprietárním softwarovým frameworkem Microsoft .NET Framework, jeho architektura je ovšem navržena tak, aby jej bylo možno rozšířit o další funkce, včetně jiných programovacích jazyků.

Visual Studio poskytuje podle své verze (edice) řadu funkcí, standardně obsahuje pokročilý editor kódu, designéry pro vizualizaci vyvíjeného softwaru, podporu pro správu projektů a debuggování. Vyšší verze nabízejí pokročilejší funkce, například podporu pro týmovou spolupráci a verzovací systém.

Visual Studio je dodáváno s podporou několika programovacích jazyků – C#, Visual Basic, C++ nebo F#. Tyto však nejsou součástí Visual Studia jako takového, ale jsou do něj vloženy ve formě balíčků. Takové jazykové balíčky a balíčky s jinými funkcemi pak vzájemně komunikují a poskytují uživateli žádanou funkcionalitu (viz. 2.1 – Architektura VS). Právě cestou vývoje nového jazykového balíčku je možné poměrně snadno Visual Studio rozšířit o nový programovací jazyk.

2.1 Architektura Visual Studia

Visual Studio IDE představuje framework, který hostuje různé softwarové moduly (balíčky funkcí) a umožňuje vzájemnou komunikaci mezi nimi pomocí služeb, které nabízejí.

Tyto softwarové moduly neboli balíčky – zvané zde VSPackages – produkují určitou funkcionalitu. Ta je ve formě služeb (services) šířena napříč celým IDE. Vazba mezi VSPackage a službou je zde dvojsměrná: VSPackage nabízí služby a zároveň konzumuje služby nabízené jinými balíčky (Obrázek 1) [20].

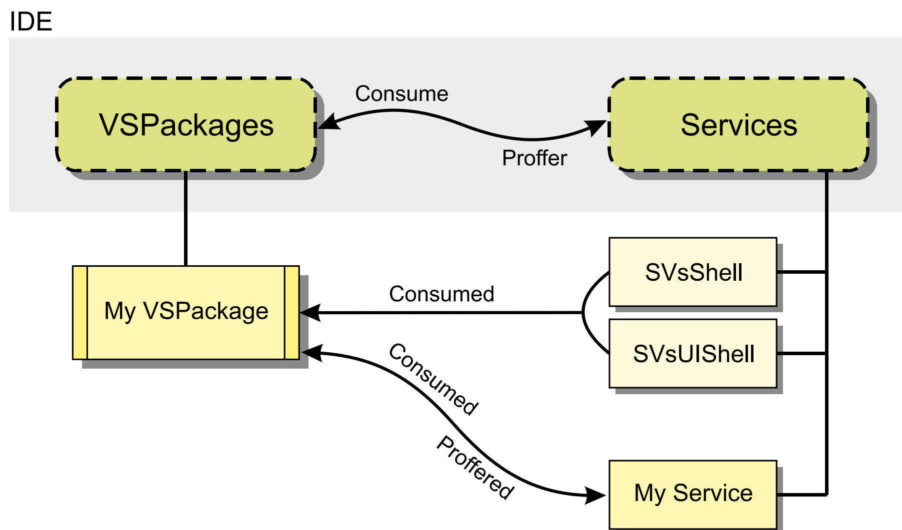
IDE standardně obsahuje 3 VSPackage moduly, jejichž služby jsou široce využívány ostatními:

- SVsShell – funkce zajišťující základní funkcionalitu, přístup k jednotlivým balíčkům a jejich registraci v IDE
- SVsUIShell – funkce poskytující funkcionalitu uživatelského rozhraní a UI služeb Windows (panely, záložky, nástrojové lišty, okna nástrojů, . . .)
- SVsSolution – základní funkce spojené s řešením (solution) – vytváření nových projektů, jejich číslování, monitorování změn v projektech

Moduly VSPackages a samotné IDE jsou vzájemně silně závislé díky svému těsnému propojení. I přes tuto silnou integraci jsou však zodpovědné za různé činnosti.

Visual Studio IDE zodpovídá za tyto úkoly:

- Poskytování nezbytných a elementárních služeb externím VSPackages



Obrázek 1: Architektura VSPackages a Services

- Poskytování programovatelného rozhraní umožněním přístupu ke standardním prvkům UI
- Vytváření instancí modulů VSPackages, jejichž služby jsou požadovány uživatelem nebo jinými balíčky
- Poskytování služeb umožňujících komunikaci a koordinaci mezi moduly VSPackages
- Správu řešení (solutions) a jimi vyžadovaných souborů
- Management oken vývojového prostředí
- Panely nástrojů, menu, kontextová menu, spouštění příkazů

VSPackages zodpovídají za tyto úkoly:

- Inicializace a ukončování podprogramů
- Zápis informací do registrů, které jsou využívány pro načítání vhodných VSPackages v případě potřeby
- Nabízení služeb nutných pro komunikaci s jinými VSPackages
- Implementace nových typů projektů, editorů a designerů
- Poskytování rozšíření pro standardní prvky UI

2.2 Rozšiřitelnost (Extensibility)

Vývojáři používající Visual Studio mohou jeho funkce rozšířit a zefektivnit svou práci. Microsoft poskytl několik způsobů, jak lze Visual Studio rozšířit o nové funkce nebo některé, již existující, zautomatizovat. Tato rozšíření lze pak distribuovat pro použití v dalších instancích Visual Studia.

Existují 3 možnosti rozšíření:

1. **Macro** – automatizuje opakující se úkoly a akce, které vývojáři mohou nahrát, uložit, přehrávat a dále distribuovat. Makra mohou být dále použita k implementaci nových příkazů nebo vytváření oken nástrojů. Jsou psány ve Visual Basicu a nejsou zkompilovány.
2. **Add-In** – zajišťuje přístup k objektovému modelu Visual Studia a lze jej propojit s nástroji IDE. Může být použito k implementaci nové funkcionality a nových oken nástrojů. Je připojen k IDE pomocí COM a může být vytvořen v jakémkoli COM kompatibilním jazyce.
3. **VSPackage** – nejvyšší úroveň rozšiřitelnosti. Pro jeho vytvoření je nutný SDK (Software Development Kit). Používá se k vytváření komplexních rozšíření jako např. designéry, zcela nové funkce nebo podpora dalších programovacích jazyků.

Tato práce se dále věnuje pouze jednomu typu rozšíření – a to rozšíření o podporu funkcionálního jazyka pomocí VSPackage modulů. Tento typ rozšíření se jeví jako jediný vhodný pro tak komplexní úlohu, jako je podpora nového jazyka. Navíc, Visual Studio ve verzi 2010 nabízí výhodnou možnost správy takových rozšíření a jejich snadnou škálovatelnost skrze využití frameworku MEF (viz. 2.2.1).

Visual Studio 2010 a s ním dodávaný .NET Framework 4 posunují vývoj rozšíření o krok dál. Celé IDE bylo kompletně přepsáno do WPF (Windows Presentation Foundation), čímž značně vzrostly možnosti rozšíření UI. Dále byl do .NET Frameworku začleněn MEF (Managed Extensibility Framework) – knihovna, usnadňující vývoj rozšíření.

2.2.1 MEF – Managed Extensibility Framework

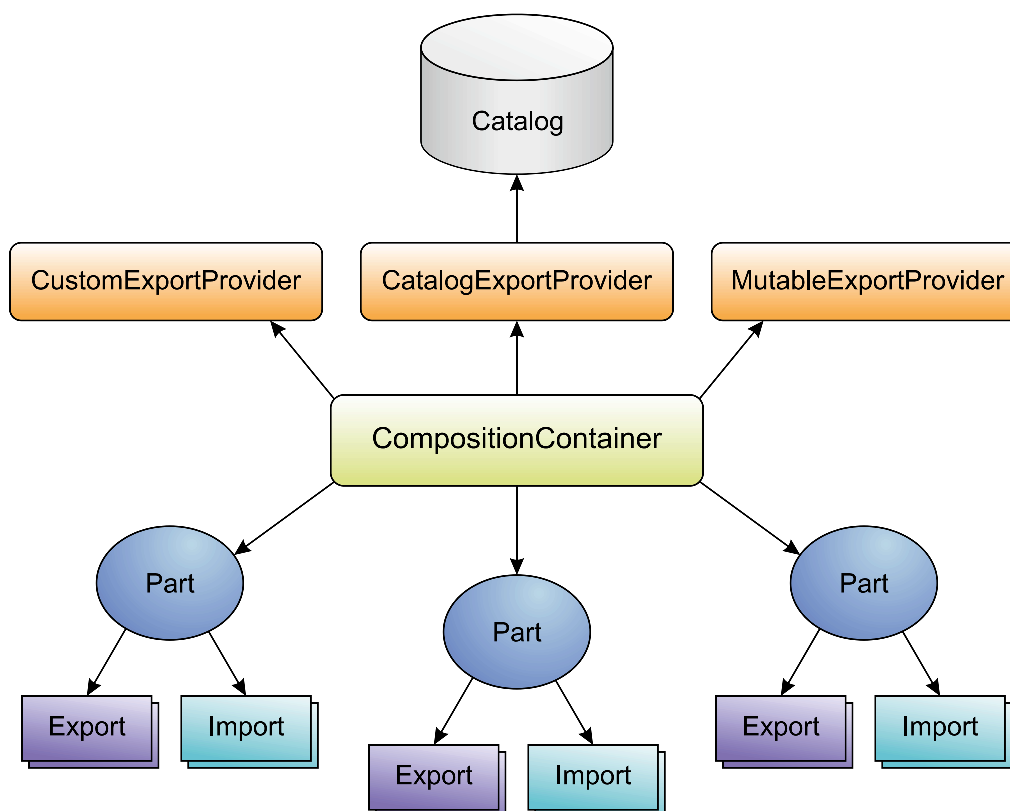
MEF (Managed Extensibility Framework) umožňuje vývojářům aplikací pro rozšíření snadno nalézt, propojit, skládat a vzájemně používat rozšíření bez složitých konfigurací prostředí. Je přímou součástí .NET Frameworku 4 a je tedy dostupný všude tam, kde je .NET Framework 4 nainstalován.

Před příchodem knihovny MEF musely aplikace, které podporovaly použití plug-inů, obsahovat vlastní infrastrukturu podporující plug-in model. Takové plug-iny jsou pak často specifické pro danou aplikaci a nemohou být použity napříč jinými aplikacemi.

MEF představuje standardizovaný přístup k hostování aplikací, které zveřejňují své služby a zároveň využívají služeb okolních externích aplikací (rozšíření). Rozšíření vytvořená pomocí MEF mohou ze své podstaty být znovupoužity v různých aplikacích, ale přesto být implementovány způsobem, který je specifický právě pro jednu aplikaci [15].

Jednotlivá rozšíření mohou být závislá jedno na druhém. O jejich vzájemné propojení a jeho správné pořadí se postará MEF, vývojáři tak odpadají starosti s tímto obvykle spojené.

MEF nabízí aplikacím několik přístupů k nalezení a načtení rozšíření, které vyžadují. Umožňuje také značkování rozšíření pomocí dodatečných metadat, čímž usnadňuje jejich nalezení.



Obrázek 2: MEF model

Jádrem MEF je **katalog** a **kontejner**. Katalog hraje roli v procesu nalezení hledaného rozšíření a kontejner pak koordinuje jeho vytvoření i vytvoření potřebných závislostí (Obrázek 2).

Hlavními jednotkami, které tvoří MEF model jsou složitelné díly – **Composable Parts**. Tyto díly zveřejňují (exportují) služby, které jiné díly potřebují a využívají, a konzumují (importují) služby jiných dílů [1]. V MEF modelu jsou tyto díly označeny atributy:
[System.ComponentModel.Composition.ExportAttribute] a
[System.ComponentModel.Composition.ImportAttribute],
čímž deklarují, co exportují nebo importují.

Composable Part by měl obsahovat alespoň jeden export. Composable Parts jsou pak buď explicitně vloženy do kontejneru, nebo v něm vytvořeny s použitím katalogu. Standardně jsou Composable Parts identifikovány katalogem pokud obsahují atribut *[Export]*.

Následující kód ukazuje 3 možné rovnocenné deklarace exportu a kontraktů.

```
[Export]
public class Exporter {...}

[Export(typeof(Exporter))]
public class Exporter1 {...}

[Export("MEFSample.Exporter")]
public class Exporter2 {...}
```

Výpis 1: Deklarace exportů tříd

Obvykle je výhodnější, aby Composable Part exportoval rozhraní (případně abstraktní typ), než typ samotný. To vede ke známému efektu, kdy se zde může importér zcela oprostit od specifik implementace importovaného typu a pracovat s ním obecně.

Další ukázka zobrazuje implementaci 2 tříd exportujících sebe coby *IMessageSender* rozhraní. Třída *Notifier* importuje kolekci tříd s *IMessageSender* rozhraním, nad kterými volá metodu *Send()*. Jakýkoli další sender tak může být snadno přidán pouhým implementováním rozhraní a exportem deklarovaným jako rozhraní *IMessageSender*.

```
[Export(typeof(IMessageSender))]
public class EmailSender : IMessageSender {
    ...
}

[Export(typeof(IMessageSender))]
public class TCPSender : IMessageSender {
    ...
}

public class Notifier
{
  [ImportMany]
  public IEnumerable<IMessageSender> Senders {get; set;}

  public void Notify(string message)
  {
      foreach(IMessageSender sender in Senders)
      sender.Send(message);
  }
}
```

Výpis 2: Deklarace exportu třídy jako rozhraní

Kontrakt exportu může obsahovat další metadata, které mohou být použita pro filtrování vyhledávání v katalogu.

Jak již bylo uvedeno, exportovat lze nejen celé třídy (typy), ale také properties . . .

```
public class Configuration
{
  [Export("Timeout")]
  public int Timeout
  {
    get { return int.Parse(ConfigurationManager.AppSettings["Timeout"]); }
  }
}

[Export]
public class UsesTimeout
{
  [Import("Timeout")]
  public int Timeout { get; set; }
}
```

Výpis 3: Deklarace exportu properties

a metody . . .

```
public class MessageSender
  {
    [Export(typeof(Action<string>))]
    public void Send(string message)
    {
      Console.WriteLine(message);
    }
  }

[Export]
public class Processor
{
  [Import(typeof(Action<string>))]
  public Action<string> MessageSender { get; set; }

  public void Send()
  {
    MessageSender("Processed");
  }
}
```

Výpis 4: Deklarace exportu metod

Filozofie, se kterou byl MEF model vytvořen, umožňuje nalézt a připojit potřebné Composable Parts dynamicky za běhu aplikace. Odpadají tak např. „natvrdo“ zakódo-

vané reference, konfigurační soubory, nebo podobné přístupy. MEF tak ukazuje způsob, jak snadno nalézt a ověřit vhodnost připojených komponent (Composable Parts) pouze za pomocí metadat, tedy bez instanciování komponent nebo připojování jiných assembly. Díky tomu není nutné řešit, kdy a jak komponenty připojit.

Navíc, kromě toho, že Composable Parts definují své exporty, definují i importy, tedy prvky, které importují z jiných částí, které potřebnou funkcionalitu řeší. To umožňuje nejen snadnou komunikaci mezi jednotlivými Parts, ale poskytuje i dobré možnosti faktoringu, kdy například služby společné více komponentám mohou být faktorovány do samostatného Composable Part a zde pak snadno v případě nutnosti upravovány.

Právě relativní jednoduchost skládání těchto Composable Parts komponent do souvislého celku a absence závislostí na aplikačních assemblies dělá z MEF ideální model pro rozšiřování Visual Studia 2010. Tyto principy budou uplatňovány při popisu rozšíření v následujících kapitolách.

2.2.2 Stručný náhled na rozšiřitelnost jiných IDE

Závěr této kapitoly je věnován stručnému přehledu řešení rozšiřitelnosti jiných známých integrovaných vývojových prostředí — Eclipse, NetBeans a Visual Studia 2008.

Eclipse

Eclipse je výkonné integrované prostředí pro vývoj různých typů aplikací. V povědomí vývojářů je známo zejména jako hlavní vývojový nástroj pro projekty založené na platformě Java.

Jedná se o otevřenou platformu, navrženou tak, aby byla snadno a neomezeně rozšiřitelná balíčky třetích stran. Eclipse je postaveno nad frameworkem OSGI, který poskytuje dynamickou modulární architekturu pro zavádění balíčků (bundles). Samotnou aplikaci Eclipse tak netvoří jeden velký program, ale pouze relativně malá aplikace – tzv. loader. Ten je obklopen rozsáhlou množinou plug-inů, které jsou realizovány pomocí programů v jazyce Java a rozšiřují funkcionalitu prostředí Eclipse. Plug-in je realizován jako soubor *.jar*. Je to soběstačný balíček zahrnující zdrojový kód a všechny zdroje, které ke své činnosti potřebuje. Každý plug-in může být konzumentem služeb poskytovaných jinými balíčky, nebo sám jiným své služby nabízet. Zde lze pozorovat jistou analogii s architekturou MEF Visual Studia. Tyto plug-iny jsou nahrány do prostředí Eclipse loaderem v okamžiku jeho startu nebo při vyvolání požadavku na funkcionalitu, kterou poskytují. Plug-iny mohou Eclipse rozšiřovat například o nová menu, nástroje, editory, nebo perspektivy.

NetBeans

NetBeans je další z integrovaných vývojových prostředí cílených především na platformu Java. Prostředí je i zde postaveno na modulární architektuře, umožňující neomezenou rozšiřitelnost.

Základem je aplikace (NetBeans Platform) tvořící jakousi kostru pro připojování dalších rozšíření. Poskytuje základní funkcionalitu pro vývoj, jako jsou menu, správa do-

kumentů zdrojových kódů a různá nastavení. Další nové funkce lze doplnit připojením tzv. modulů (nebo též plug-inů), které stávající funkcionalitu rozšíří. Ty jsou uloženy v registru modulů – System Filestystem. Hlavním konfiguračním souborem je zde soubor *layers.xml*, který definuje registrované moduly. Samotné moduly jsou deklarovány v souborech *manifest.mf* a jsou tvořeny množinou Java tříd definujících novou funkcionalitu. Moduly tvoří základní jednotku rozšiřitelnosti, pokud je více modulů vzájemně závislých, tvoří jednotku jejich nasazení tzv. Module Suite. Podobně jako u ostatních prostředí, i zde lze moduly použít na rozšíření různých prvků IDE – menu, nové funkce, rozšíření editoru, apod.

Visual Studio 2008

O předchozí verzi Visual Studia (verzi 2008) lze říci, že uplatňuje podobné principy pro rozšíření, jako verze 2010, ochuzené o některé nové rysy dané evolucí tohoto prostředí. Základní logickou jednotkou rozšíření je zde VsPackage skládající se z jedné hlavní a případně několika dalších doplňkových COM komponent. Interakce mezi IDE a VsPackages je abstrahována pomocí množiny COM rozhraní. Především absence frameworku MEF, jakož i editor nevybudovaný nad WPF, činí z implementace rozšíření složitější úlohu než implementace téhož ve verzi 2010.

Orientace na novou moderní architekturu, u níž lze v budoucnu očekávat další rozvoj, možnost snadného přístupu k editoru a snadná modifikovatelnost jeho chování, a možnost jednoduché modularity rozšíření, byly hlavními důvody proč pro implementaci podpory pro jazyk e-PFL bylo zvoleno Visual Studio ve verzi 2010.

3 Podpora nového jazyka ve Visual Studiu 2010

Nezbytným nástrojem pro rozšíření Visual Studia 2010 o podporu nového jazyka je tzv. Visual Studio SDK (Software Development Kit). Jedná se o volně stažitelný doplněk, složený z nových nástrojů a šablon, umožňující vývojářům rozšiřovat Visual Studio o nová menu, nástroje, okna, různá vylepšení editoru, podporu nových jazyků a další funkce.

Rozšíření Visual Studia 2010 o podporu nového programovacího jazyka představuje jednu ze složitějších programátorských úloh. Visual Studio 2010 k této úloze přistupuje oproti předchozím verzím inovativně zavedením knihovny MEF a dalšími vylepšeními [21]. Tuto poměrně komplexní úlohu lze rozdělit do několika logických celků (viz níže), a i ty lze dále rozdrobit do menších. Tato výhoda znamená u této úlohy možnost začít používat nový jazyk v prostředí Visual Studia dříve, než jsou dokončeny všechny součásti jeho podpory. Podporu jazyka pak lze později kdykoli rozšířit o další funkce a přibližovat se tak podpoře v současnosti standardně dodávaných jazyků jako jsou C#, Visual Basic, C++ nebo F#.

Rozšířit Visual Studia 2010 o podporu nového programovacího jazyka znamená zejména:

- Zavést do prostředí Visual Studia možnost zakládat nové projekty na bázi nového jazyka a standardním způsobem s nimi pracovat
- Umožnit Visual Studiu přístup k překladači jazyka (kompilátoru) a zajistit rozhraní pro spolupráci s ním
- Připravit doplňky pro editor Visual Studia tak, aby umožňoval uživateli s novým jazykem komfortně pracovat
- Zajistit podporu pro procházení a analýzu kódu (debuggování) případně další specifické funkce

Následující kapitoly popisují problematiku implementace některých částí z těchto podúloh.

3.1 Projekt

Vývojáři při práci ve Visual Studiu, ať již používají jakýkoli jazyk, obvykle začínají vytvořením nového řešení (solution), které obsahuje jeden výchozí projekt na bázi zvolené šablony projektu. Vývoj podpory nového jazyka proto logicky začíná zavedením nových typů projektů na něm založených.

Projekty Visual Studia jsou vlastně jakési kontejnery obsahující soubory se zdrojovými kódy a další pomocné soubory. Jejich struktura a závislosti jsou zobrazeny v některém z nástrojových oken, např. Solution Explorer, a vývojář je zde může dále organizovat, spouštět, analyzovat, atd. Projekty jsou následně zkompilovány do jednotek sestavení – tzv. assembly.

Základem pro rozšíření Visual Studia o možnost správy projektů nového jazyka je implementace nového modulu VSPackage. Šablona pro tento typ projektu je dostupná po instalaci Visual Studio SDK 2010. Dalším podstatným krokem je přidání reference na projektový framework *Microsoft.VisualStudio.Project*. Jedná se o rozšíření frameworku MPF (Managed Package Framework), které není součástí instalace .NET Frameworku, ale je volně ke stažení ve formě zdrojového kódu [16].

Úkoly, které je třeba splnit při implementaci podpory pro projekty [11], jsou:

- Vytvoření šablon projektů
- Vytvoření nového typu projektu
- Vytvoření ikon projektů
- Registrace šablon projektů ve Visual Studiu
- Vytvoření project factory, která generuje nové projekty
- Návrh a implementace parametrů projektů
- Podpora pro parametry projektu

3.1.1 Šablony projektů

Šablony, na jejichž základě bude nový projekt nového jazyka vytvářen, jsou prvním krokem při implementaci.

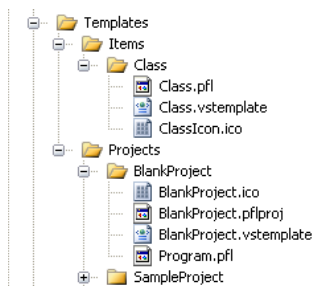
V rámci projektu je třeba vytvořit adresářovou strukturu (Obrázek 3), kde ve složce *Projects* jsou umístěny šablony projektů – každá v samostatném adresáři, který tvoří:

- Soubor šablony projektu (*.vstemplate*)
- Hlavní soubor projektu (*.XXXproj*)
- Ikona projektu (*.ico*)
- Šablony prvků projektu – další soubory, které budou v nově vygenerovaném projektu obsaženy (soubory s kódem, apod.)

Hlavní soubor šablony (*.vstemplate*) musí mít vlastnost **Build Action** nastavenu na **VSTemplate**. Při sestavení budou všechny položky sloučeny do *.zip* souboru, který bude umístěn do úložiště projektových šablon.

Šablony jsou popsány speciálními XML soubory s příponou *.vstemplate*. Popisují, jak jsou šablony zobrazeny v dialogovém okně **New Project**, jaké parametry jsou použity při vytváření projektu a jak jsou naplněny, a dále jaké soubory tvoří samotný projekt.

Soubor *.vstemplate* a všechny potřebné soubory jsou zazipovány do *.zip* souboru, který je uložen na místě, které je Visual Studiu známo, a které je použito jako zdroj šablon projektů při otevírání dialogu **New Project**.



Obrázek 3: Struktura složek se šablonami

Šablonu definuje tag <VSTemplate>, který dále obsahuje tagy <TemplateData> a <TemplateContent>.

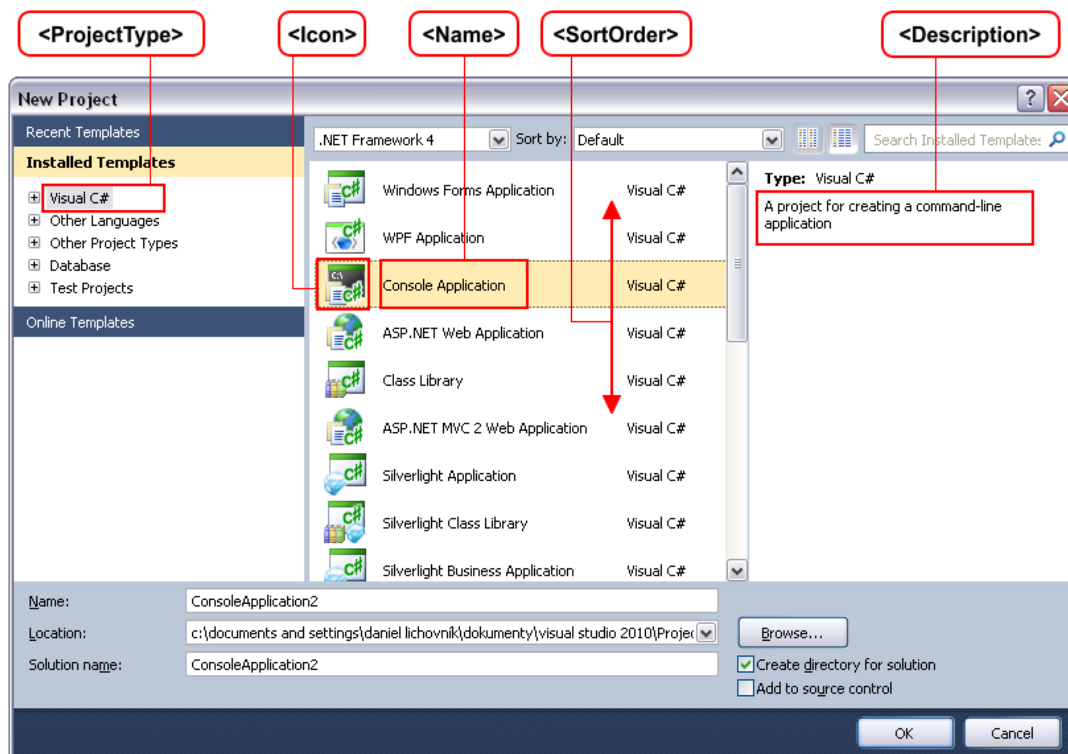
Tag <TemplateData> obsahuje informace o tom, jak bude šablona zobrazena v dialogovém okně **New Project**, a kde bude umístěna ve struktuře ostatních šablon (Obrázek 4). Významné informace:

- <Name> – Název projektu
- <Description> – Informace popisující účel projektu uživateli
- <Icon> – Název souboru .ico s ikonou projektu
- <ProjectType> – Název typu projektu – nejvyšší úroveň v hierarchii – obvykle název jazyka
- <SortOrder> – Specifikuje pořadí, podle kterého bude projekt umístěn mezi ostatními

Tag <TemplateContent> obsahuje informace o projektovém souboru *.XXXproj* a informace o všech dalších souborech, které budou při otevření projektu vygenerovány. Dále specifikuje parametry a jejich hodnoty, které budou při generování projektu použity.

- <Project> – specifikuje název hlavního projektového souboru *.XXXproj*, obsahuje podtagy <ProjectItem> popisující jednotlivé prvky projektu
- <ProjectItem> – název dalšího souboru, jeho chování vůči parametrům (ReplaceParameters) a otevření/neotevření (OpenInEditor) v editoru při vygenerování projektu
- <CustomParameters> – uživatelem definované parametry a jejich hodnoty

Vybere-li si uživatel v dialogu **New Project** šablonu, na které založí nový projekt, Visual Studio rozbalí odpovídající *.zip* soubor a načte z něj *.vstemplate* soubor. Podle něj pak vygeneruje v novém nebo existujícím řešení nový projekt se všemi definovanými součástmi (project items). Jak již bylo řečeno, tyto jsou popsány vlastními soubory zvanými



Obrázek 4: Dialogové okno New Project

šablony prvků (item templates), které jsou zabaleny ve stejném *.zip* souboru. Z těchto šablon prvků jsou pak v novém projektu vygenerovány soubory a patřičně upraveny pomocí parametrů. Úprava spočívá v nahrazení řetězců specifikovaných v šablonách prvků ohraničením znakem $ hodnotami parametrů.

Mezi standardně definované parametry patří například:

- `$itemname$` – název prvku projektu zadaný uživatelem v dialogu **Add New Item**
- `$safeitemname$` – totéž s odstraněnými nepovolenými znaky a mezerami
- `$projectname$` – název projektu zadaný uživatelem v dialogu **New Project**
- `$safeprojectname$` – totéž s odstraněnými nepovolenými znaky a mezerami
- `$rootnamespace$` – výchozí namespace daného projektu

Kompletní seznam rezervovaných parametrů má 14 položek [19].

Další vlastní parametry a jejich hodnoty lze definovat ve *.vstemplate* souboru pomocí tagů `<CustomParameter>`.

```
<TemplateContent>
    ...
    <CustomParameters>
        <CustomParameter Name="$ProjectParameter1$" Value="Value1"/>
        <CustomParameter Name="$ProjectParameter2$" Value="Value2"/>
    </CustomParameters>
</TemplateContent>
```

Výpis 5: Ukázka definice vlastních parametrů v šabloně *.vstemplate*

Jednotlivé prvky projektů (project items), které lze k projektu přidávat pomocí dialogového okna **Add New Item**, jsou definovány velmi podobným způsobem – pomocí šablon uplatňující podobné principy jako šablony projektů. Tyto šablony jsou opět umístěny v samostatných složkách, tentokrát pod složkou *Templates/Items* (Obrázek 3).

Šablonu zde netvoří projektový soubor, ale jen soubor *.vstemplate*, ikona a vlastní soubor se šablonou. Ostatní náležitosti jsou stejné jako u šablon projektů.

Projekty jsou popsány speciálními XML soubory s příponou *.XXXproj*, kde XXX je obvykle symbolický název programovacího jazyka (*.csproj*, *.vbproj*).

Tento soubor obsahuje zejména informace o tom, jak bude výsledný projekt sestaván.

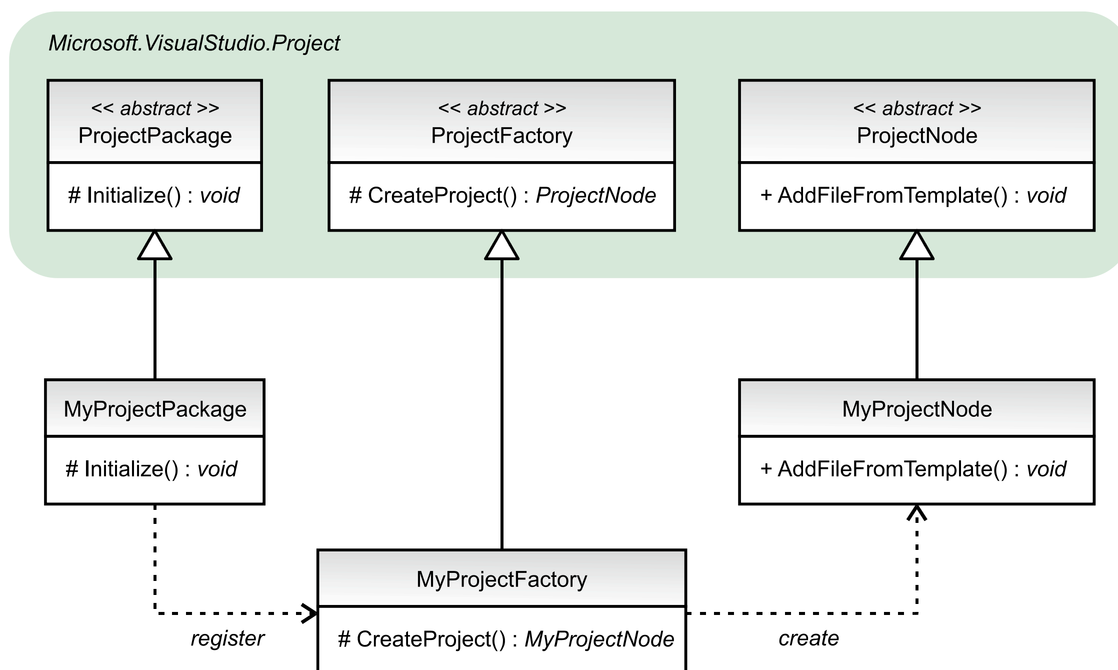
Mimo jiné obsahuje:

- Cestu k souboru *XXX.targets* s údaji pro kompilaci
- GUID projektu
- Typ assembly, do které bude zkompilován (exe, dll, winexe)
- Název výsledné assembly
- Údaje pro debuggování
- Reference na jiná assembly
- Seznam souborů, které budou kompilovány

3.1.2 Struktura tříd pro generování projektů

Po vytvoření šablon projektů je nutné vybudovat systém, který na jejich základě vygeneruje nové projekty. Takový systém poskytuje framework **Microsoft.VisualStudio.Project**, který standardně není součástí .NET Frameworku, ale je volně ke stažení. Rozšiřuje základní framework MPF a je tvořen množinou tříd, které poskytují kompletní kontrolu nad procesem generování nových projektů ve Visual Studiu, jejich správu, perzistenci, přidávání dalších prvků, apod [16].

Minimální implementace systému pro generování a správu projektového systému nad soustavou šablon projektů nového jazyka znamená vytvoření 3 tříd, děděných z jejich abstraktních obrazů v namespace *Microsoft.VisualStudio.Project* (Obrázek 5).



Obrázek 5: Class diagram architektury pro generování projektů

Třída *MyProjectPackage*

Výchozí třídou je třída *MyProjectPackage*. Dědí z abstraktní třídy *ProjectPackage*, která dále dědí ze třídy *Package* patřící do namespace *Microsoft.VisualStudio.Shell*. Třída *Package* implementuje rozhraní *IVsPackage*, které zajišťuje registraci modulu do prostředí Visual Studia.

Úkolem *MyProjectPackage* je registrace modulu VSPackage a registrace tovární třídy, která pak generuje vlastní projekty.

Klíčová je metoda *Initialize()*, která právě slouží k inicializaci a registraci tovární třídy. Dalším nezbytným krokem je označení třídy několika atributy:

- *[PackageRegistration]* – říká registrační utilitě RegPkg, že se jedná o modul VSPackage, který obsahuje další specifika v atributech, které je třeba vzít v úvahu
- *[InstalledProductRegistration]* – informace o modulu, které budou zobrazeny v dialogu „About“ Visual Studia
- *[ProvideProjectFactory]* – významný atribut, deklarující tovární třídu, která generuje projekt. Specifikuje typy projektů, které generuje, a cestu k šablonám projektů
- *[ProvideProjectItems]* – pro daný typ projektu specifikuje prvky projektu (items), které bude možno k projektu připojovat

- *[ProvideObject]* – deklaruje objekty (jejich typy), které tento modul VSPackage poskytuje

Třída *MyProjectFactory*

Třída *MyProjectFactory* musí přetížit metodu *CreateProject* své rodičovské třídy *ProjectFactory*. Jak již název napovídá, jedinými úkoly této metody je vytvořit novou instanci projektu (*MyProjectNode*) a nastavit poskytovatele služby, ze které bude projekt dostupný.

Jediným atributem je atribut *[Guid]* obsahující identifikátor, pod kterým je tovární třída zapsána v registrech.

Třída *MyProjectNode*

Vlastní projekt je objekt popsaný třídou *MyProjectNode*. Jako i ostatní zmiňované třídy, je i tato zděděna z abstraktní třídy *ProjectNode* z frameworku MPF. Jedná se o komplexní třídu implementující řadu rozhraní. Díky MPF a možnosti dědění z něj odpadá vývojáři spousta práce pro zajištění úplné funkčnosti projektu. Vše je připraveno v rodičovské třídě, která zajišťuje zejména:

- Vytvoření složky vygenerovaného projektu a zkopírování souboru *.XXXproj* do této složky. Soubor je přejmenován podle názvu projektu zadaného uživatelem v dialogu **New Project**
- Vyhledání souborů označených jako položky vstupující do sestavení assembly – v souboru *.XXXproj* označených tagem `<compile>`

Vývojáři stačí pouze připravit mechanismus pro získání ikony projektu zobrazované např. v okně Solution Explorer, a zajistit předávání parametrů (jako jsou např. namespace, název souboru (použitelný např. pro název třídy), apod.) přidávaným souborům přetížením metody *AddFileFromTemplate*, která je volána při přidávání souborů do projektu. Kopíruje vybrané soubory ze šablony do cílové složky projektu [18].

Tato implementace představuje minimální strukturu pro generování projektů. Právě využití frameworku MPF z ní dělá relativně snadnou programátorskou úlohu.

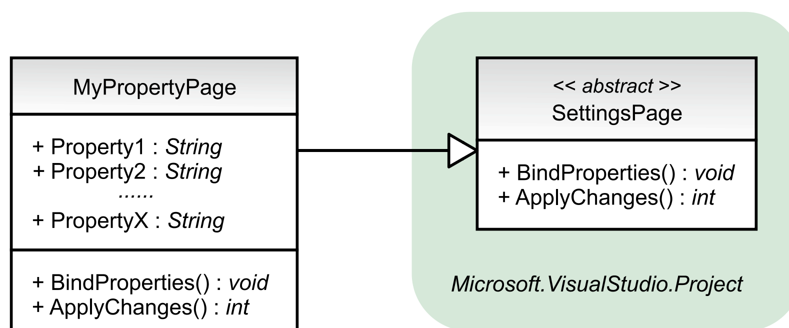
Tuto strukturu lze dále rozšiřovat o další potřebné prvky, vhodná bývá alespoň implementace vlastností (properties) projektu, které pak může uživatel podle potřeby měnit.

3.1.3 Vlastnosti projektu

Projekt může obsahovat vlastnosti definované v šablonách. Některé je vhodné zpřístupnit uživateli, ať již pro pouhé čtení, nebo pro editaci. Toho lze dosáhnout implementací tzv. property page, která vlastnosti zpřístupňuje v samostatném dialogovém okně v tabulkovém formátu. Okno s vlastnostmi projektu je dosažitelné přes kontextovou nabídku projektu v okně Solution Explorer.

Obsluhu zajišťuje třída *SettingsPage* frameworku MPF. Zprostředkovává získávání i persistenci dat uložených ve vlastnostech projektu. Umí pracovat se všemi běžnými datovými typy, které odpovídajícím způsobem zobrazuje v dialogovém okně vlastností.

Zavedení nových vlastností projektu (např. údaje pro kompilátor, apod.) probíhá vytvořením třídy *MyPropertyPage* děděné ze *SettingPage* (Obrázek 6).



Obrázek 6: Class diagram třídy implementující property page

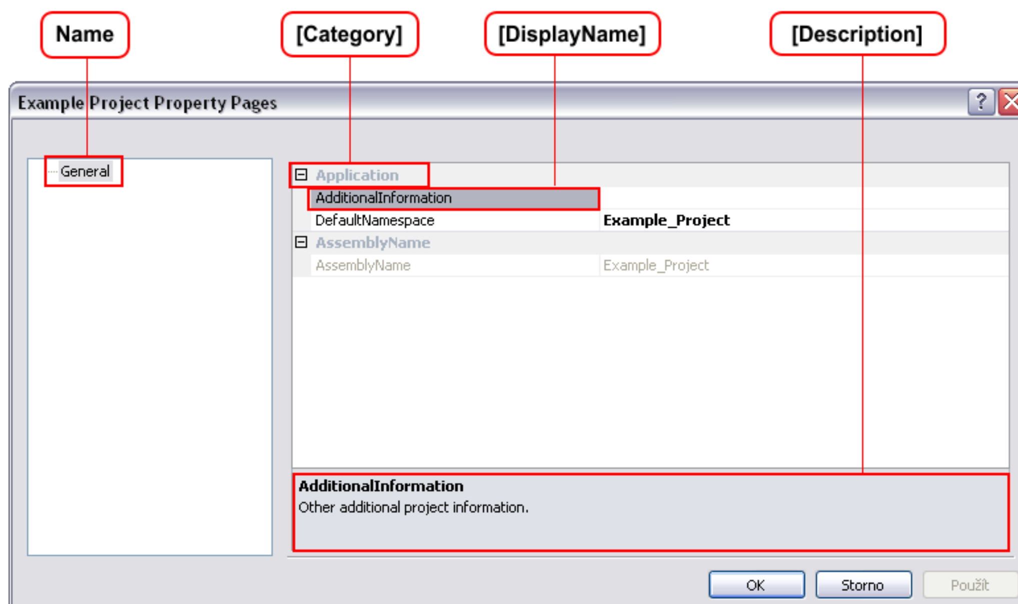
Každá z vlastností je definována veřejnými properties, zamezení možnosti editace vlastnosti uživatelem je dosaženo vynecháním implementace metody *set* (read-only property). Každá taková property musí být označena atributy (Obrázek 7):

- *[Category]* – kategorie, do které vlastnost patří – zobrazena jako sekce v tabulce vlastností
- *[DisplayName]* – jméno vlastnosti, jak bude zobrazeno v tabulce
- *[Description]* – popis vlastnosti – rovněž zobrazeno uživateli

Vlastní uložení / načtení vlastností je dosaženo přetížením metod *BindProperties* / *ApplyChanges*, jež toto provádějí voláním metod *GetProjectProperty* / *SetProjectProperty* na objektu projektu (*ProjectNode*).

Pro správnou funkčnost je třeba doplnit registraci property page třídy v atributu *[ProvideObject]* třídy *MyProjectPackage*, a dále pak ve třídě *MyProjectNode* přetížit metody *GetConfigurationIndependentPropertyPages* a *GetPriorityProjectDesignerPages*, které vracejí property page GUID.

Je možné implementovat více property page tříd, vždy děděných ze *SettingPage*, a vytvořit tak systém oddělených vlastností, např. podle určení. Pro všechny platí vždy výše uvedené náležitosti pro implementaci, atribut *Name* je položka zobrazená v dialogovém okně vlastností (Obrázek 7) určující kategorii vlastností (stránku).



Obrázek 7: Dialogové okno Vlastnosti projektu

3.2 Kompilátor

Kompilátory jsou počítačové programy, jejichž účelem je zprostředkování překladu zdrojového kódu jazyka vyšší úrovně do kódu jazyka nižší úrovně. Jazykem nižší úrovně je zde většinou jazyk cílové platformy, kterému počítač přímo rozumí a umí ho spustit – strojový kód, kód čitelný virtuálním strojem. Cílem překladu je tedy zpravidla spustitelný program.

Překladač (kompilátor) musí provádět dvě základní činnosti: analyzovat zdrojový program a vytvářet k němu odpovídající cílový program. Analýza spočívá v rozkladu zdrojového programu na jeho základní součásti, na základě kterých se během syntézy vybudují moduly cílového programu [4].

Analýza zdrojového programu při překladu probíhá na následujících třech úrovních:

- Lexikální analýza – zdrojový program je coby posloupnost znaků sekvenčně čten zleva doprava a jsou z něj vytvářeny lexikální symboly (tokeny) – např. klíčová slova, operátory, konstanty, identifikátory
- Syntaktická analýza – z lexikálních symbolů jsou vytvořeny hierarchicky zanořené struktury odpovídající gramatice jazyka. Tyto pak tvoří celky s vlastním významem – např. výrazy, příkazy, deklarace.
- Sémantická analýza – ostatní kontroly, které není možno provést v rámci syntaktické analýzy – typová kontrola apod.

Moderní vývojová prostředí obvykle obsahují překladač pro každý jazyk, který podporují. Dokáží tak nejen sestavit vyvíjený program, ale také jej po překladu a sestavení spouštět. Vývojové prostředí pak dokáže zobrazovat zprávy překladače a zobrazit přehled o případných chybách, které mohou být zvýrazněny přímo v editoru kódu. Sofistikovanější prostředí dokáží navíc skenovat zdrojový kód v editoru přímo během jeho vytváření a zvýrazňovat chyby ještě před samotným překladem.

IDE s integrovaným kompilátorem neznamená jen možnost překladu přímo přes vývojové prostředí, ale také spuštění programu následně po překladu. V neposlední řadě nutno zmínit i komfort pro uživatele, který může celý překlad spravovat přes grafické uživatelské rozhraní, včetně možnosti přehledného a úplného nastavování parametrů pro překlad.

3.2.1 Připojení vlastního kompilátoru

Implementaci podpory překladu vlastního jazyka ve Visual Studiu lze rozdělit do dvou etap:

1. Implementaci kompilátoru coby samostatné aplikace schopné vygenerovat cílový kód ze zdrojových kódů nového jazyka dodaného kompilátoru v množině souborů
2. Vlastní propojení kompilátoru s Visual Studiem

Tato práce se dále nezabývá problematikou vytváření aplikací pro kompilaci a předpokládá existenci takové aplikace.

Proces kompilace ve Visual Studiu je popsán tzv. tasky. Tasky jsou definovány v samostatných třídách děděných ze třídy .NET Frameworku *Task* v namespace *Microsoft.Build.Utilities*.

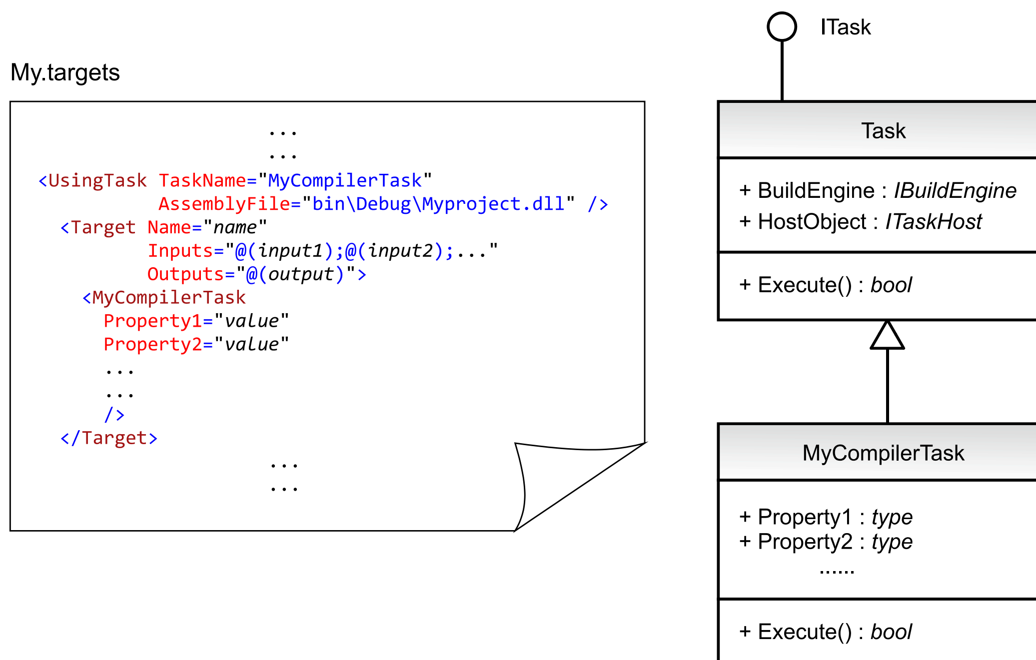
Třída *Task* poskytuje funkcionalitu nezbytnou pro úspěšné vykonání překladu definováním logiky v metodě *Execute()*. Tato metoda je jedinou metodou definovanou rozhraním *ITask*, které třída *Task* implementuje. Build engine Visual Studia vyžaduje a rozeznává třídy s právě tímto rozhraním jako jednotky s kódem, který lze volat při operacích překladu.

Task popsaný vlastní třídou je dále nutné deklarovat ve speciálním konfiguračním XML souboru s příponou *.targets* (Obrázek 8) [13].

Každý task je popsán v tagu `<UsingTask>` jménem třídy a assembly, ve které se nachází.

Tag `<Target>` pak definuje všechny potřebné vstupy a výstupy pro proces překladu a definuje způsob plnění veřejných properties třídy tasku, které dále definují některé nezbytnosti.

Cesta k souboru *.targets* je definována v konfiguračním souboru projektu *.XXXproj* v každé šabloně projektu.



Obrázek 8: Class diagram třídy tasku + soubor *.targets*

3.2.2 Logika procesu překladu

Visual Studio při vyvolání procesu překladu uživatelem (příkaz Build solution), nebo nějakého jiného procesu, jehož součástí je překlad (např. Run), spustí metodu *Execute()* ve třídách (tascích), definovaných v souboru *.targets*.

Základní implementace metody *Execute* předpokládá především:

1. Přípravu parametrů pro překlad
2. Spuštění kompilačního programu s těmito parametry
3. Zajištění umístění výstupů kompilace na místo vyžadované Visual Studiem
4. Zobrazení logů kompilačního programu v prostředí Visual Studia

Ad 1) **Parametry** potřebné **pro překlad** jsou definovány jako veřejné properties třídy děděné ze třídy *Task*. Jsou dále deklarovány v souboru *.targets* pod tagem `<Target>`. Mezi základní properties běžně používané Visual Studiem, které lze snadno definovat a automaticky před překladem plnit, patří:

- SourceFiles – zdrojové soubory, které budou přeloženy do cílové assembly. Patří sem všechny soubory s kódem označené v souboru *.XXXproj* tagem `<compile>`.

- OutputAssembly – assembly včetně cesty z adresáře projektu, kde Visual Studio očekává výstup kompilátoru. Obvykle „obj/Debug/*assemblyName*“.
- ReferencedAssemblies – seznam assemblies referencovaných v projektu.
- ResourceFiles – seznam dalších zdrojových (resource) souborů.
- MainFile – označuje vstupní bod aplikace.
- TargetKind – typ assembly, do které bude kompilováno – exe, dll, winexe.
- DebugSymbols – generování informací pro debuggování – ano/ne.
- ProjectPath – cesta k adresáři projektu.

Tyto parametry lze plnit přímo přiřazením odpovídajících definovaných systémových nebo projektových parametrů. Výčet parametrů tímto výčtem není zdaleka limitován, vývojář může připravit jakékoli jiné, podle potřeby kompilátoru, a plnit je dynamicky.

Ad 2) **Spuštění kompilačního programu** v zásadě předpokládá pouze vyřešení zavolání a spuštění aplikace zajišťující překlad. Je vhodné řešit spuštěním na pozadí, bez jakékoli možnosti přímé interakce s cílovým uživatelem.

Ad 3) Visual Studio očekává **výstup kompilátoru** na pevně daném místě. Obvykle je to „*projectPath*/obj/Debug/*assemblyNameWithExtension*“. S výstupem na tomto místě pak dále pracuje – přesouvá jej do adresáře *Bin*, odkud jej pak případně spouští. Pokud kompilační program nedokáže přímo zajistit generování výstupu do tohoto umístění, a s tímto jménem, je úlohou vývojáře zde toto ošetřit – zajistit přesun/přejmenování výstupu.

Ad 4) **Výstupy z překladu** – logy, reporty, chybová hlášení, apod. – které kompilační program může generovat, je vhodné zobrazovat přímo v prostředí Visual Studia. Tyto výstupy pak přes uživatelské rozhraní Visual Studia představují komunikační kanál uživatel-kompilátor.

Je jen na vývojáři, jakou formu zobrazování zvolí. Jako nejpřímější se jeví zobrazovat výstupy do **Output** dialogu. Mírně vyspělejší možností je využití **Error list** dialogu, s dalším členěním podle typu výstupu (Errors, Warnings, Messages). Dle možností kompilátoru je možné dořešit i podrobnější výstupy chyb, jako například čísla řádků s chybami, nebo zvýraznění chyb přímo v editoru.

Vývojář může zvolit i jinou formu výstupu, například zobrazovat výstupy ve vlastním okně, kde může naplno využít možností WPF.

3.3 Rozšíření editoru

Editory kódu v integrovaných prostředích slouží nejen ke vkládání kódu vývojářem, ale poskytují další funkce, které práci usnadňují, urychlují a tím zefektivňují. Tyto funkce představují přidanou hodnotu editorů v IDE oproti zápisu kódu v některém běžném textovém editoru.

Editory tvoří nejviditelnější součást vývojových prostředí a zpravidla sem směřují první kroky při seznamování se s novým prostředím. Dnešní vyspělá IDE disponují editory s pokročilými funkcemi, mezi nimiž standardně nechybí barevné zvýrazňování syntaxe, upozorňování na syntaktické chyby, zvýrazňování chybných gramatických konstrukcí, formátování kódu, apod.

Přes všechny tyto výhody mohou nastat situace, kdy vývojář některou funkci postrádá, ať již jako nezbytnou součást pro svou práci, či jen jako efektní doplněk, který zná třeba z jiného prostředí. Řešení tohoto problému nebývá triviální. Podle otevřenosti IDE se může jednat jen o doplnění funkce do existujícího editoru, úpravu existujících funkcí, nebo vytvoření zcela nového editoru.

Microsoft v nové verzi (2010) Visual Studia představil zcela nový, přepracovaný editor. Zásadním přínosem je použití a zavedení dvou technologií:

- WPF (Windows Presentation Foundation)
- MEF (Managed Extensibility Framework)

První z nich přináší změnu do uživatelského rozhraní. Technologie WPF umožňuje oddělit vizuální prvky editoru od vlastní logiky v pozadí. Editor tak získává mnohem pokročilejší zobrazovací možnosti dovolující široké možnosti modifikace.

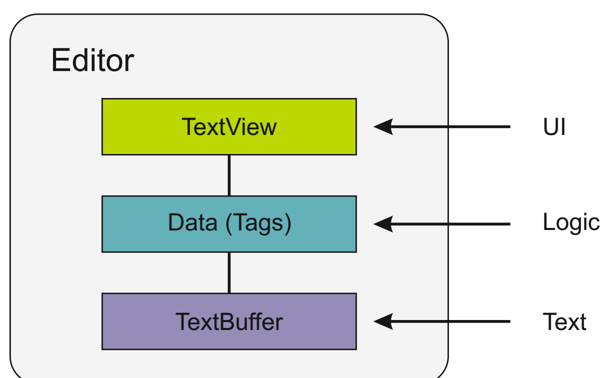
Editor VS je dále postaven na technologii MEF (kapitola 2.2.1) – je složen ze součástí tvořených MEF komponentami. Jednoduše tak lze vytvářet a doplňovat další rozšíření coby MEF komponenty obsahující nové funkce editoru a hostovat je v editoru Visual Studia [14].

Zjednodušeně lze činnost editoru VS popsat modelem na obrázku 9. Vlastní text zde představuje vrstva TextBuffer. Ta obsahuje měnící se text, který poskytuje dále ve formě neměnných časových snímků. Další vrstva reprezentuje logiku dodání dalších informací k textu. Tyto informace představují metadata textu, která slouží jako informace o textu pro uživatele. Uživatel tyto informace vidí ve formě, kterou mu poskytne prezentační vrstva – TextView.

3.3.1 Struktura editoru VS

Editor Visual Studia 2010 se skládá z několika subsystémů [12]:

- **Text Model Subsystem** – reprezentuje vlastní text v editoru a poskytuje mechanismy pro manipulaci s ním. Obsahuje TextBuffer z obrázku 9. Ten je tvořen sekvencí znaků, které mají být v editoru zobrazeny. Nad touto sekvencí model umožňuje vyhledávání a modifikace. Reprezentovaný text představuje „surový“ text bez jakéhokoli formátování nebo dalších vizuálních prvků.
- **Classification Subsystem** – poskytuje logiku pro určení vlastností fontu textu. Role klasifikátorů je rozčlenit úseky textu do klasifikačních tříd, pro které jsou definovány vlastnosti fontu, kterým budou zobrazeny. Tyto informace jsou poskytnuty Text View subsystému, který zařídí vlastní zobrazení.



Obrázek 9: Logický náhled na editor VS

- **Text View Subsystem** – jeho role je naformátovat a zobrazit text a poskytnout mechanismy podporující zobrazování, jako např. vizuální prvky obohacující text, výběry textu, pozice textového kurzoru, apod. Kromě toho zabezpečuje také zobrazování a správu okrajů okna editoru (margins). Ty obsahují například rolovací lišty, čísla řádků, symboly breakpointů.
- **Operations Subsystem** – definuje chování editoru implementací logiky jednotlivých příkazů.

3.3.1.1 Text Model

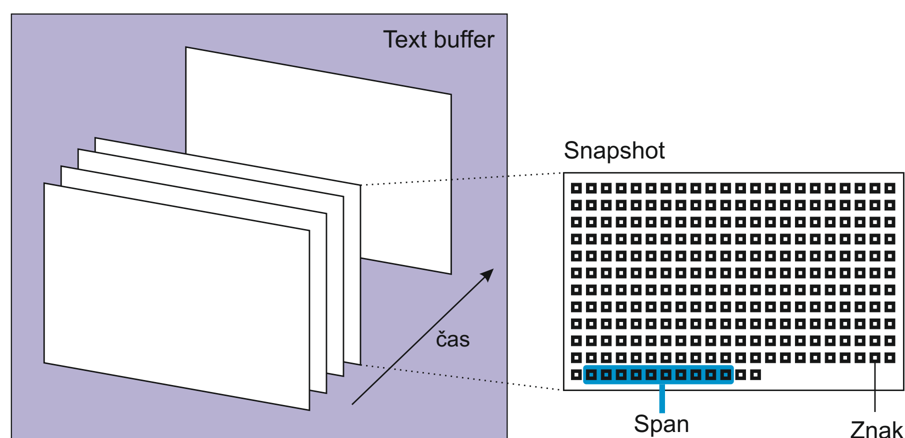
Text model editoru VS tvoří typové třídy, které jsou definovány v knihovnách *Microsoft.VisualStudio.Text.Data.dll* a *Microsoft.VisualStudio.CoreUtilitiy.dll*.

Vlastní text v editoru je reprezentován objektem typu *ITextBuffer*. Ten představuje jakýsi kontejner neustále se měnícího textu (když je v editoru upravován uživatelem). Text je zde reprezentován sekvencí znaků Unicode. Každý editační krok generuje snímek text bufferu, který v modelu zastupuje třída *Snapshot* (*ITextSnaphot*). Snapshot je v čase neměnná verze text bufferu (obrázek 10). Veškeré úpravy textu (formátování, dekorování, . . .) jsou prováděny právě nad snapshoty. Snapshoty řeší i problém souběžnosti, kdy text buffer mohl být používán pouze vláknem, které bylo jeho vlastníkem. Snapshot je možné použít z kteréhokoli vlákna.

Jak bylo řečeno, text ve snapshotu je reprezentován sekvencí znaků. Řádky textu snapshotu jsou dosažitelné pomocí *ITextSnapshotLine* objektů, úseky textu lze referencovat pomocí *SnapshotSpan* objektů.

SnapshotSpan je úsek textu definovaný svými hranicemi – pozicí začátku a konce (0 – délka snapshotu). Pozice znaku je reprezentována objekty *SnapshotPoint*.

Kolekce úseků textu jednoho snapshotu je definována kolekcí *NormalizedSnapshotSpanCollection*.



Obrázek 10: Náhled na základní prvky text modelu

Span je typ obsažený v SnapshotSpan objektech reprezentující textový úsek. Kolekce těchto úseků definovaná typem *NormalizedSpanCollection* uspořádává Span objekty podle pořadí a navíc slučuje překrývající se úseky.

ITrackingPoint objekty představují pozice znaku v text bufferu, které se aktualizují s měnící se polohou znaku v textu. Podobným způsobem fungují *ITrackingSpan* objekty, s tím rozdílem, že pracují nad úseky textu v bufferu.

Rozhraní *ITextBuffer* obsahuje funkce *Insert()*, *Delete()*, *Replace()* pro jednoduché manipulace s textem. Pro větší kontrolu nad prováděnými změnami je možné využít objekty *ITextEdit* získané z ITextBufferu.

3.3.1.2 Classification

Classification subsystem je tvořen sadou typových tříd, které jsou definovány v knihovnách *Microsoft.VisualStudio.Text.Logic.dll*.

Klasifikace je proces dodávání meta-informací k úsekům textu. Tyto údaje pak slouží prezentační vrstvě jako informace, jak daný úsek textu zobrazit. Mechanismus parsování a prohledávání textu za účelem označování spanů textu značkami je nazýván jako tagging. Klasifikaci provádí objekt implementující rozhraní *ITagger*. Ten definuje metodu *GetTags()*, která zajišťuje vlastní algoritmus procházení úseků textu (SnapshotSpanů) a jejich označování speciálními značkami – tagy.

Tagy jsou definovány třídami implementujícími rozhraní *ITag*. Klasifikátor – *ITagger* objekt – v metodě *GetTags()* označuje úseky textu, které kladně vyhodnotí jako úseky textu spadající do určité klasifikační třídy, generickými objekty *TagSpan* s parametrem typu *ITag*. Tyto tagy jsou prezentační vrstvě známy díky systému exportování rozšíření frameworkem MEF, a dokáže je zobrazit podle odpovídajícího formátování (*EditorFormatDefinition*) definovaného pro danou klasifikační třídu.

Kromě klasifikačních tagů (*ClassificationTag*) jsou standardně definovány ještě další typy tagů podporované editorem, které slouží pro jiné funkce (bude popsáno dále):

- *ErrorTag*
- *TextMarkerTag*
- *OutliningRegionTag*
- *SpaceNegotiatingAdornmentTag*

3.3.1.3 Text View

Prezentační vrstva editoru je tvořena subsystémem Text View. Zajišťuje formátování textu, vizuální doplňky textu a další grafické elementy editoru. Formátování probíhá podle klasifikátorů ležících v nižší vrstvě.

Tento subsystém je tvořen sadou typových tříd obsažených v knihovnách *Microsoft.VisualStudio.Text.UI.dll* (pro platformně nezávislé prvky) a *Microsoft.VisualStudio.Text.UI.Wpf.dll* (WPF prvky).

Text view model je reprezentován rozhraním *ITextView*, které umožňuje přístup ke třem bufferům – data buffer, edit buffer a visual buffer. Oproti nižším vrstvám pracuje s pixelovým souřadnicovým systémem s počátkem v levém horním rohu.

Formátovaný text zpracovávaný a zobrazovaný v text view se skládá z řádků reprezentovaných *ITextViewLine* objekty. Tyto objekty poskytují metody a vlastnosti pro mapování mezi znakovým a pixelovým souřadnicovým systémem a pro vizuální doplňky, které mohou text doprovázet.

Další významnou součástí editoru, za kterou zodpovídá text view subsystem jsou okraje (margins). Okraje jsou vizuální elementy editoru, které poskytují uživateli informace, jako např. čísla řádků, nebo značky (glyphs) jako např. breakpointy. Čtyři předdefinované okraje jsou *Top*, *Left*, *Bottom* a *Right*. Tyto představují jakési kontejnery, do kterých jsou pak vloženy jiné margins. Okraje jsou reprezentovány *ITextViewMargin* objekty, které zveřejňují metody a vlastnosti pro řízení velikosti, viditelnosti a dalších vlastností okrajů.

3.3.2 Implementace rozšíření editoru

Využití frameworku MEF při implementaci rozšíření editoru VS umožňuje tuto úlohu rozdrobit do menších celků podle poskytované funkce. Aplikací systému exportů a importů pak lze tyto části navzájem skládat a připojovat k editoru. Zjevná výhoda takové aplikace je rozdělení práce mezi více vývojářů, kteří vzájemně nemusí znát výsledky ani stav svých prací. Rovněž lze již velmi brzy dodat první funkční část rozšíření a začít ji využívat a teprve později doplňovat další a rozšiřovat tak editor o nové funkce postupně [6, 24].

Společným prvkem všech rozšíření cílených pro určitý typ užití – zejména např. pro podporu určitého programovacího jazyka – je přiřazení daného rozšíření do tzv. „typu obsahu“ – content typu.

Content type definuje typ textu, se kterým editor pracuje. Editor pak vyhledá všechny služby nabízené MEF komponentami se stejným content typem, naimportuje je a začne využívat.

Vytvoření nového content typu je prvním krokem při implementování podpory nového jazyka v editoru. Vytvoření proběhne definováním nového content typu s unikátním jménem, definováním proměnné typu *ContentTypeDefinition* a následným exportováním.

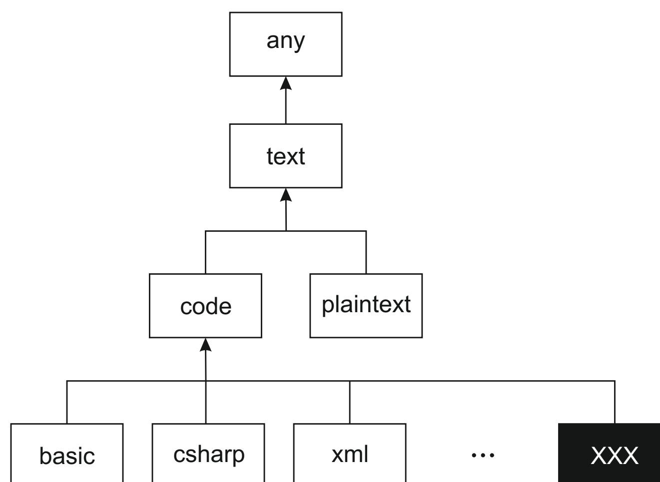
```
[Export]
[Name("XXX")]
[BaseDefinition("code")]
internal static ContentTypeDefinition XXXContentTypeDefinition;
```

Výpis 6: Definice a export content typu

Atributy:

- *Name* – unikátní jméno Content typu
- *BaseDefinition* – jméno content typu, ze kterého je nově definovaný typ odvozen. Je zde možné i vícenásobné odvození.

Visual Studio již má definovanou hierarchii několika content typů pro jazyky, se kterými editor pracuje. Content type pro nový jazyk je možné začlenit kamkoli do této struktury odvozením z již existujícího content typu. Logické je zde zařazení na úroveň ostatních jazyků – odvození z content typu „code“ (Obrázek 11).



Obrázek 11: Část hierarchie Content Types ve Visual Studiu

Editor VS tedy importuje služby, které bude nabízet, na základě content typu. O tom, který content type je právě platný, je rozhodnuto na základě typu textu v editoru. Identifikace probíhá podle přípony souboru, který text reprezentuje (*.cs* pro C#, *.vb* pro

Visual Basic, apod.). Svázání přípony souboru a content typu proběhne exportováním proměnné typu *FileExtensionToContentTypeDefinition* označené těmito atributy:

- *FileExtension* – přípona souboru, se kterou bude content type svázán
- *ContentType* – název content typu

```
[Export]
[FileExtension("xxx")]
[ContentType("XXX")]
internal static FileExtensionToContentTypeDefinition XXXFileExtensionDefinition;
```

Výpis 7: Definice a export napojení na souborovou příponu

Po vytvoření a exportování content typu a jeho asociací s příponou souboru je Visual Studiu a jeho editoru již typ nového jazyka znám a dokáže s ním pracovat – zobrazovat v editoru všechna definovaná formátování, dekorace, a další vizuální prvky specifické pro tento jazyk. Kterýkoli další vývojář může psát rozšíření pro tento content type a exportovat je. Tato rozšíření budou v editoru použita rovnocenně s dalšími, již existujícími rozšířeními.

Poznámka: Totéž platí i pro implementaci rozšíření pro již existující jazyky ve Visual Studiu (C#, Visual Basic, . . .).

Následující výčet představuje pojmy, které reprezentují funkce běžně dostupné v editorech IDE. Tyto funkce jsou jedny z prvních, které by při vývoji podpory editoru pro nový jazyk měly být implementovány. Jak již však bylo uvedeno, díky modularitě dané použitím frameworku MEF, nezáleží na pořadí implementace, a rozhodnutí, které z nich budou implementovány, záleží jen na vhodnosti funkce pro daný jazyk a na uvážení a potřebách vývojáře.

- **Barvení syntaxe** (syntax highlighting) – odlišení lexikálních kategorií kódu barvou textu
- **Zvýraznění vybraného textu** (text highlighting) – zvýraznění všech výskytů vybraného textu
- **Podtrhávání chyb** (error tagging) – podtržení chybných slov, případně slovních konstrukcí
- **Párování závorek** (brace matching) – zvýraznění závorek tvořících pár ke zvolené závorce
- **Odsazování** (smart indent) – automatické odsazování bloků kódu podle úrovně zanoření
- **Formátování** (formatting) – automatické formátování úseků kódu

- **Sbalování kódu** (outlining) – sbalování/rozbalování bloků kódu do jednoho zástupného řádku
- **Intellisense** – nástroj pro syntaktickou nápovědu
  - **Doplňování výrazů** (statement completion) – asistent pro zrychlení psaní kódu
  - **Popis funkcí** (quick info) – bublinová nápověda k funkcím
  - **Našeptávač parametrů** (parameter help) – popis signatur a parametrů metod
  - **Inteligentní značky** (smart tags) – místně nabízená kontextová nabídka
- **Okraje editoru** (margins) – informace v lištách na okrajích okna editoru
  - **Číslování řádků** (line numbers) – číslování řádků kódu pro snadnější orientaci
  - **Značky** (glyphs) – grafické symboly vázané k řádku kódu (breakpointy, „todo" značky, apod.)
- **Dekorace textu** (adornments) – další vizuální grafické elementy svázané s textem (rámečky kolem textu, speciální podtržení – např. gramatické chyby, apod.)

Způsob implementace některých těchto funkcí bude nastíněn v následujících podkapitolách.

Požadavky na všechny funkce editoru:

- poskytování služeb v reálném čase
- poskytování služeb bez příkazu uživatele na jejich spuštění
- poskytování služeb bez ovlivnění výkonu editoru, bez časových prodlev

3.3.2.1 Barvení syntaxe

Barvení syntaxe (v literatuře syntax highlighting, syntax coloring) je na první pohled nejviditelnější funkcí editoru. Jejím úkolem je barevné odlišení jednotlivých lexikálních prvků jazyka podle definovaných klasifikačních tříd. Slouží pro snadnou orientaci v kódu, zlepšuje čitelnost kódu, poskytuje jednoduché ověřování syntaktických chyb při psaní.

Klasifikační třída prvku jazyka je definována deklarací proměnné typu *ClassificationTypeDefinition*, označením nezbytnými atributy, a jejím exportováním.

```
[Export(typeof(ClassificationTypeDefinition))]
[Name("keyword")]
[BaseDefinition("text")]
internal static ClassificationTypeDefinition KeywordDefinition;
```

Výpis 8: Definice a export klasifikačního typu

Atributy:

- *Name* – unikátní jméno klasifikační třídy
- *BaseDefinition* – jméno klasifikační třídy, ze které definovaná třída dědí. Všechny klasifikační třídy dědí ze třídy „text" (není nutno explicitně uvádět).

Způsob, jakým bude editor klasifikační třídu zobrazovat, je definován třídou děděnou ze třídy *ClassificationFormatDefinition* a exportovanou jako typ *EditorFormatDefinition* s potřebnými atributy.

```
[Export(typeof(EditorFormatDefinition))]
[ClassificationType(ClassificationTypeNames = "keyword")]
[Name("keyword")]
[UserVisible(false)]
[Order(After = Priority.Default, Before = Priority.High)]
internal sealed class KeywordFormat : ClassificationFormatDefinition
{
    public KeywordFormat()
    {
        this.ForegroundColor = Colors.Blue;
    }
}
```

Výpis 9: Definice a export formátování

Atributy:

- *Name* – název definice formátu
- *ClassificationType* – názvy klasifikačních tříd, na které bude formát mapován
- *UserVisible* – určuje, zda bude možné formát nastavit uživatelem v dialogu **Options**
- *Order* – priorita vůči jiným klasifikačním třídám

Po definici klasifikačních tříd a formátů pro zobrazení následuje definice vlastního mechanismu pro označování textu klasifikačními tagy. Klasifikátor provádějící označování (tagging) je popsán třídou implementující rozhraní *ITagger<ClassificationTag>*. Její metoda *GetTags()* vrací enumerátor typu *IEnumerable<ITagSpan<ClassificationTag>>* umožňující postupně získávat úseky textu označené klasifikačními tagy. Vlastní logika označování závisí na komplexitě jazyka – od jednoduchého parsování textu podle pevně daných pravidel, až po přenechání tohoto úkolu na zvlášť k tomu uzpůsobené komponentě – lexikálním analyzéru.

Posledním krokem je definice poskytovatele klasifikátoru – třídy implementující rozhraní *ITaggerProvider*.

Implementace funkce barvení kódu je tímto dokončena. Jedná se o poměrně snadnou úlohu, jejíž kritickou částí je implementace algoritmu pro klasifikaci lexikálních jednotek

jazyka (metoda *GetTags*). Podobné principy jsou platné i pro další ze zmíněných funkcí editoru, a proto budou dále popsány již jen stručně.

3.3.2.2 Podtrhávání chyb

Dalších ze standardně nabízených funkcí editoru bývá podtrhávání chyb (též error tagging, error highlighting). Podle úrovně sofistikace funkce se může jednat jen o zvýraznění syntaktických chyb, nebo i o pokročilejší výsledky sémantické analýzy.

Visual Studio zvýrazňuje chyby podtržením slov s výskytem chyby červenou vlnitou čarou. Editor provede podtržení automaticky u všech úseků textu označené tagem typu *IErrorTag*.

Realizace této funkce je velmi podobná implementaci barvení syntaxe. Klasifikátor („tagger“) chyb je zde reprezentován třídou implementující rozhraní *ITagger<IErrorTag>*.

Její metoda *GetTags()* vrací enumerátor typu *IEnumerable<ITagSpan<IErrorTag>>*. V těle této metody je implementace algoritmu pro označování chybných úseků textu. Tuto činnost lze opět přenechat lexeru, u komplexnějších, vyspělejších jazyků, mohou k identifikaci chyb přispívat i struktury ležící nad lexerem – např. syntaktický čí sémantický analyzér.

3.3.2.3 Sbalování bloků kódu

Visual Studio poskytuje v editoru funkci pro sbalování bloků kódu. Tato funkce snižuje množství zobrazeného kódu v editoru a tím zvyšuje přehlednost. Sbalitelné bloky jsou u levého okraje editoru označeny symbolem „minus“, po sbalení je pak blok kódu nahrazen jedním řádkem a označen symbolem „plus“. V jazyce C# je takto defaultně nabízeno sbalení těl jmenných prostorů, tříd, metod, vlastností, a bloků označených „#region/#endregion“.

Editor Visual Studia poskytuje tuto funkcionalitu automaticky pro úseky textu označené tagy typu *IOutliningRegionTag*. Implementace pro nový jazyk je zde opět stejná jako v předchozích případech – zde identifikace bloků textu jako kandidátů na sbalení a jejich označování správnými tagy.

3.3.2.4 Zvýraznění vybraného textu, párování závorek

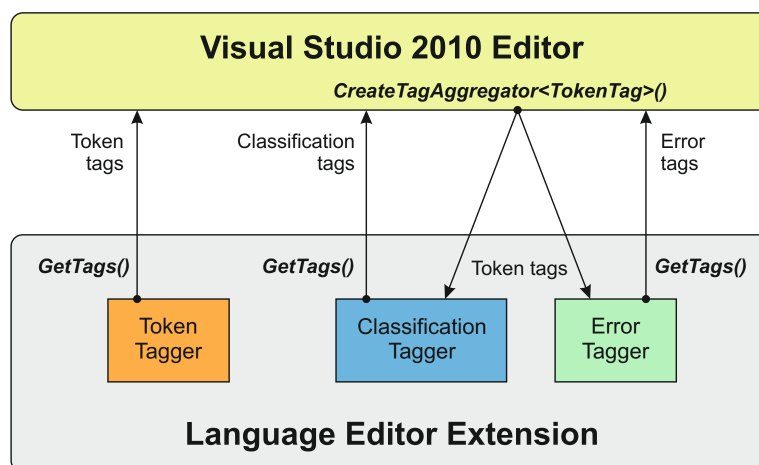
Posledním zde zmíněným typem tagování textu je označování tagem typu *TextMarkerTag*. Slouží k označování úseků textu, které mají být dočasně zvýrazněny. Princip je zde opět shodný jako u předchozích funkcí, zde je navíc podobně jako u barvení syntaxe nutná definice formátování takto označeného textu třídou děděnou ze třídy *MarkerFormatDefinition* a exportovanou jako typ *EditorFormatDefinition* s potřebnými atributy.

Použití tohoto tagování je vhodné např. při implementaci funkce pro označení páru souvisejících závorek, nebo při implementaci funkce, která zvýrazní v editoru všechny výskyty textu, který je uživatelem označen.

3.3.2.5 Využití společného klasifikátoru

Všechny předchozí funkce využívaly ke své činnosti výsledky práce vlastního klasifikátoru (taggeru) pro rozčlenění textu na úseky a jejich označování speciálními tagy. Tuto činnost vnitřně zprostředkovávala k tomu určená komponenta – analyzér (lexikální, syntaktický, . . .).

Při implementaci více funkcí najednou se nabízí možnost využívat společného taggeru, který by jednotlivé lexikální jednotky textu (tokeny) získané z analyzéru označoval například několika obecnými tagy. Proces prvotního rozložení textu na lexikální jednotky bývá nazýván jako tokenizace. Tento tagger je pak exportem začleněn do kontejneru frameworku MEF. Taggery jednotlivých funkcí pak mohou přes tzv. agregátor konzumovat úseky textu označené obecnými tagy a přeznačkovávat je podle svého algoritmu, aniž by muselo dojít k opětovnému využití analyzéru (Obrázek 12) [2, 17]. Tento efektní přístup je možný díky architektuře frameworku MEF a nepochybně přispívá ke zvýšení výkonu editoru.



Obrázek 12: Agregování klasifikátorů

Tagger využívající výsledky obecného taggeru musí provést tyto dva importy:

```
[Import]
internal IBufferTagAggregatorFactoryService aggregatorFactory = null;

[Import]
internal IClassificationTypeRegistryService ClassificationTypeRegistry = null;
```

Výpis 10: Ukázka připojení agregátoru

První z nich – Aggregator Factory Service – umožňuje vytvořit agregátor, který poskytuje tagy vytvořené obecným taggerem.

Druhý poskytuje mechanismus pro asociaci klasifikačních typů s prezentačními formáty.

Tagger, který přebírá z agregátoru již vytvořené tagy, pak jen tyto tagy prochází v těle své metody *GetSpan()* a přeznačkovává je podle svého algoritmu.

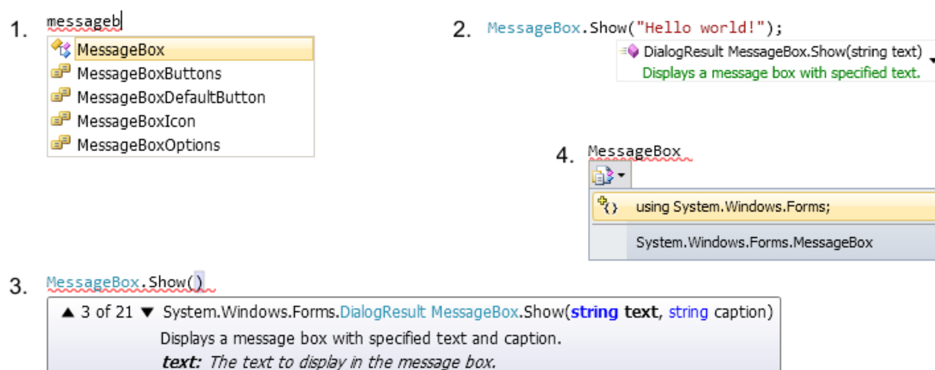
```
public IEnumerable<ITagSpan<ClassificationTag>> GetTags(
      NormalizedSnapshotSpanCollection spans)
{
     foreach (var tagSpan in this.aggregator.GetTags(spans))
    {
           ......
    }
}
```

Výpis 11: Ukázka použití agregátoru

3.3.2.6 Intellisense

Jako „Intellisense" je nazývána sada 4 funkcí editoru, jejichž primárním účelem je zvýšení produktivity při vkládání kódu. Nejznámější z nich je doplňování výrazů (autocompletion, statement completion), se kterým bývá název Intellisense často zaměňován.

Sada těchto funkcí jako celek zprostředkovává vývojáři přístup ke klíčovým slovům, identifikátorům a funkcím, včetně jejich popisu a seznamu parametrů. Za zvýšení produktivity lze považovat zejména snížení počtu nutných vstupů z klávesnice, snížení paměťové náročnosti vývojáře na syntaxi jazyka a tím menší nutnost využití externí dokumentace.



Obrázek 13: Náhled na čtyři funkce Intellisense

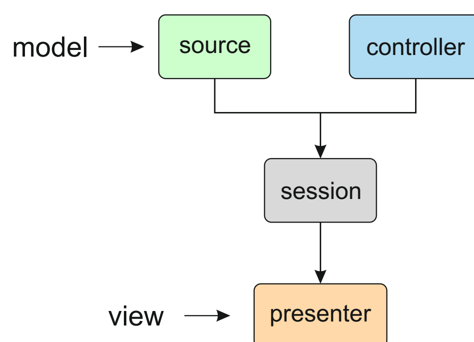
Náhled na 4 funkce Intellisense ukazuje obrázek 13:

1. **Doplňování výrazů** (statement completion)

2. **Popis funkcí** (quick info)
3. **Našeptávač parametrů** (parameter help)
4. **Inteligentní značky** (smart tags)

Model architektury Intellisense je tvořen sadou dedikovaných komponent spolupracujících podle návrhového vzoru MVC – Model-View-Controller (Obrázek 14).

- **Intellisense Source** – poskytuje aktuální obsah, který se má po vyvolání funkce zobrazit (např. seznam možných výrazů k doplnění u funkce statement completion, nebo text v bublinové nápovědě Quick info).
- **Intellisense Controller** – spravuje životní cyklus objektu Intellisense Session.
- **Intellisense Session** – objekt reprezentující stav aktivní funkce Intellisense. Je vytvořen v reakci na akci uživatele.
- **Intellisense Presenter** – zajišťuje zobrazení obsahu funkce Intellisense v editoru.
- **Intellisense Broker** – zajišťuje interakci mezi funkcí Intellisense a ostatními typy komponent.



Obrázek 14: MVC model Intellisense

Obecně lze funkci Intellisense popsat takto: Rozšíření editoru poskytuje pro daný content type Intellisense Controller. Ten naslouchá příkazům uživatele, vyhodnocuje je a v reakci na ně spouští Intellisense Session. Session následně zjistí aktuální obsah k zobrazení od objektu Intellisense Source, který je pak zobrazen v editoru přes Intellisense Presenter.

Pro zavedení funkcionality Intellisense pro podporu nového jazyka je dostačující implementace Intellisense Source a Intellisense Controller pro každou ze čtyř Intellisense funkcí, která bude implementována.

Intellisense Source zastupují rozhraní *ICompletionSource*, *IQuickInfoSource*, *ISignatureHelpSource* a *ISmartTagSource*.

Pro zdroj je nutné definovat také poskytovatele – implementovat rozhraní *ICompletionSourceProvider*, *IQuickInfoSourceProvider*, *ISignatureHelpSourceProvider* a *ISmartTagSourceProvider*.

Intellisense Controller je dále implementací *IIntellisenseController*.

Všechny uvedené implementace jsou exportovány jako MEF komponenty přiřazené do patřičného content typu.

Statement Completion

Automatické doplňování (statement completion) je funkce Intellisense, která zvyšuje produktivitu při vkládání kódu nabízením výrazů vhodných v daném kontextu. Podle míry sofistikovanosti mohou nabízené výrazy obsahovat pevně daný výčet výrazů (např. klíčových slov jazyka), nebo navíc i použité identifikátory a atributy typů získané pomocí reflexe.

Session statement completion je spuštěna (start) v reakci na uživatelovo vložení několika počátečních znaků výrazu z klávesnice, nebo kombinací kláves k tomu určených. Podle již vložené části výrazu může být nabídka filtrována. Session je pak ukončena (commit), vybere-li uživatel některý z nabízených výrazů myší, nebo definovanými klávesami (obvykle Enter, mezerník, Tab, středník, . . .), nebo zrušena (dismiss), jestliže uživatel pokračuje ve vkládání kódu tak, že již žádný výraz v nabídce nevyhovuje, nebo uživatel zruší nabídku např. stiskem klávesy Esc.

Jak již bylo řečeno, zdrojem nabízených výrazů je třída implementující rozhraní *ICompletionSource*. Ta obsahuje metodu *AugmentCompletionSession*, jejímž účelem je právě dodat nabízené výrazy pro kompletaci naplněním kolekce typu *IList<CompletionSet>* a svázat je s úsekem textu, který potencionálně nahradí (applicable to span). Úsek textu je u Intellisense lokalizován podle polohy jeho začátku v bufferu. K tomu slouží třída *SnapshotPoint*.

Pro sledování uživatelova vstupu z klávesnice je v controlleru implementována třída s rozhraním *IOleCommandTarget*. Pokud ta zachytí vstup, který by mohl spustit statement completion session, použije Intellisense Broker k vytvoření nové session a spustí ji. Broker je importován z Visual Studia v době vytvoření controlleru.

IOleCommandTarget ve své metodě *Exec()* implementuje logiku, která určuje, jaká další akce bude vykonána v reakci na další konkrétní vstup z klávesnice.

Uvedený postup je dostačující pro vývoj funkce statement completion pro podporu nového jazyka ve Visual Studiu. Grafické rozhraní je zajištěno automaticky Visual Studiem defaultní implementací Intellisense Presenteru. Vývojář zde může – podobně jako i u ostatních Intellisense funkcí – podle potřeby definovat nový grafický výstup s využitím grafických možností WFP.

Quick Info

Rychlý popis funkcí (Quick Info) je jednoduchá funkce, která uživateli zobrazí v bublinovém okně informace o typu, metodě, apod., na který uživatel nastaví ukazatel myši. Její využití snižuje především frekvenci využívání dokumentace.

Session Quick Info je spuštěna při nastavení kurzoru myši na výraz, pro který je nápověda Quick Info definována. Session končí, opustí-li kurzor tento výraz.

Zdrojem obsahu nápovědy je zde třída implementující *IQuickInfoSource*. Ta pracuje nad již vytvořeným taggerem, který vytváří tagy nad úseky textu. Takto označené úseky přebírá QuickInfoSource pomocí agregátoru (viz. kapitola 3.3.2.5). Metoda *AugmentQuickInfoSession* implementuje logiku, která z agregátoru získá tag úseku textu (span), nad kterým stojí kurzor myši. Podle něj pak vyhledá vhodnou nápovědu a naplní jí kolekci *IList<object>*. Obsahem nemusí být jen text, ale libovolné WPF objekty.

Neustálé překreslování a posun bublinového okna při pohybu myší, kdy ale kurzor setrvává na stejném výrazu, lze řešit definicí tzv. applicableToSpan objektu typu *ITrackingSpan*, pro který k překreslování pak nedochází.

Cílem předchozích odstavců nebylo poskytnout vyčerpávající návod, jak implementovat všechny typy funkcí pro podporu jazyka v editoru. Účelem bylo prozkoumat strukturu editoru Visual Studia, a analyzovat a nastínit základy a principy implementace některých funkcí, zejména těch, které byly použity při vývoji podpory pro jazyk e-PFL.

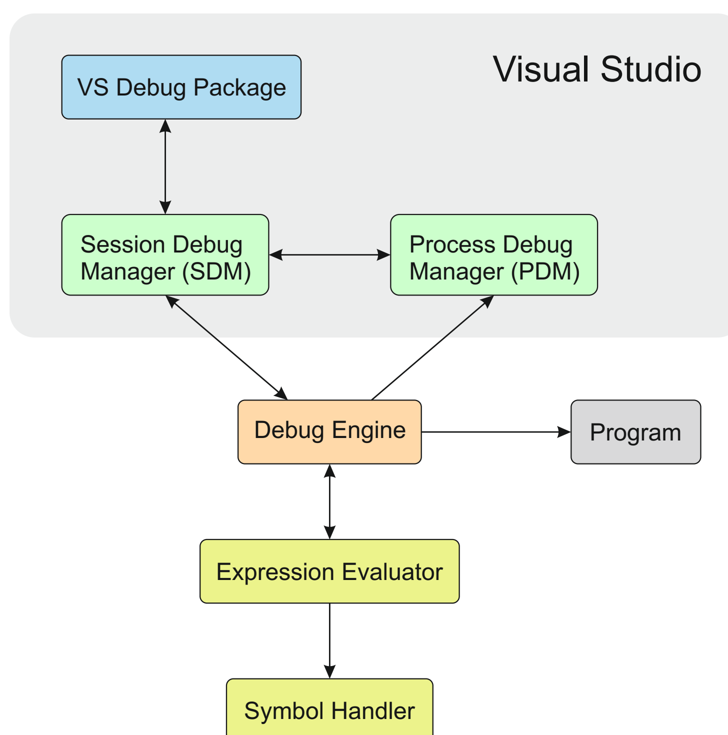
3.4 Debugger

Debuggery jsou počítačové programy, které slouží vývojářům k ladění vyvíjených programů. Primárním cílem ladění je nalezení chyb v programech, odhalení jejich příčiny, a jejich odstranění. Debugger může existovat jako samostatný program, který se po spuštění „napojí “ na laděný program a poskytuje výstupy z jeho běhu. Současná sofistikovaná integrovaná vývojová prostředí nabízejí integrované debuggery umožňující ladit programy v jazycích, které nabízejí.

Debugger integrovaný ve Visual Studiu podporuje ladění pro C#, Visual Basic, C++ a několik skriptovacích jazyků. Skládá se z několika samostatných komponent, z nichž některé jsou společné všem jazykům, jiné specifické pro každý z nich. Při zavádění nového jazyka do prostředí Visual Studia a potřeby ladění programů v něm vyvíjených je možné tyto specifické komponenty doimplementovat a začlenit je do správného kontextu v rámci debuggeru.

Implementace debuggeru je poměrně náročná programátorská úloha [24], která přesahuje rámec tohoto textu. Pro podporu jazyka e-PFL rovněž nebyla provedena implementace debuggeru. Cílem této kapitoly je proto pouze zevrubný úvod do struktury debuggeru Visual Studia a nastínění problematiky jeho rozšíření.

Schematický náhled na debugger Visual Studia ukazuje obrázek 15.



Obrázek 15: Schéma debuggeru VS

- **Debug Package** – běží ve Visual Studio Shell, zabezpečuje přenos výstupů debuggeru do UI Visual Studia.
- **Session Debug Manager** – komponenta poskytující jednotné rozhraní pro správu všech připojených Debug Engine komponent. Funguje jako Debug Engine multiplexer a poskytuje IDE jednotný náhled na debug sessions. Jeho další funkcí je šíření událostí (stop, continue, ...) z IDE do Debug Enginů. Informace o vlastním stavu průchodu programem, vláknem, apod. nejsou SDM známy přímo, ale jsou získány právě prostřednictvím debuggovacích událostí.
- **Process Debug Manager** – spravuje programy a procesy, které jsou (nebo mohou být) laděny. Laděný program musí být během procesu spouštění zaregistrován v PDM. PDM pak zajišťuje přístup k těmto programům pro SDM a pro Debug Engines.
- **Debug Engine** – zodpovídá za monitoring stavu laděného programu a celkově poskytuje Visual Studiu debuggovací rozhraní, kdy veškerá komunikace probíhá přes COM. Ve spolupráci s interpreterem jazyka nebo operačním systémem poskytuje ladící služby, jako např. řízení vykonávání programu, breakpointy, vyhodnocování výrazů. Prostřednictvím komponent Expression Evaluator a Symbol Handler provádí real-time analýzu stavu proměnných a paměti laděného programu.

- **Expression Evaluator** – dynamicky vyhodnocuje proměnné a výrazy na základě znalostí syntaxe jazyka, a umožňuje uživateli vidět jejich stav v určitém časovém okamžiku, byl-li běh programu pozastaven (break mode).
- **Symbol Handler** – přistupuje k symbolickým debuggovacím informacím generovaným překladačem pro vyhodnocení proměnných nebo výrazů, které mapuje na spuštěnou instanci programu. Visual Studio standardně pro tyto informace používá Program DataBase (*pdb*) formát.

Po spuštění programu IDE zjišťuje nastavení pro debugger (struktura *VsDebugTargetInfo2*) zahrnující pracovní adresář, port, ve kterém program poběží, a debug engine, který bude použit. Tyto informace jsou předány do debug package (metoda *LaunchDebugTargets2*). Debug package pak vytvoří instanci session debug manageru (SDM), který zajistí spuštění laděného programu a potřebného debug enginu. Debug engine dále pomocí operačního systému spustí program. Spolu s ním je spuštěno i potřebné run-time prostředí. Následně debug engine vytvoří tzv. program node (*IDebugProgramNode2*) popisující program a oznámí portu, že program byl spuštěn.

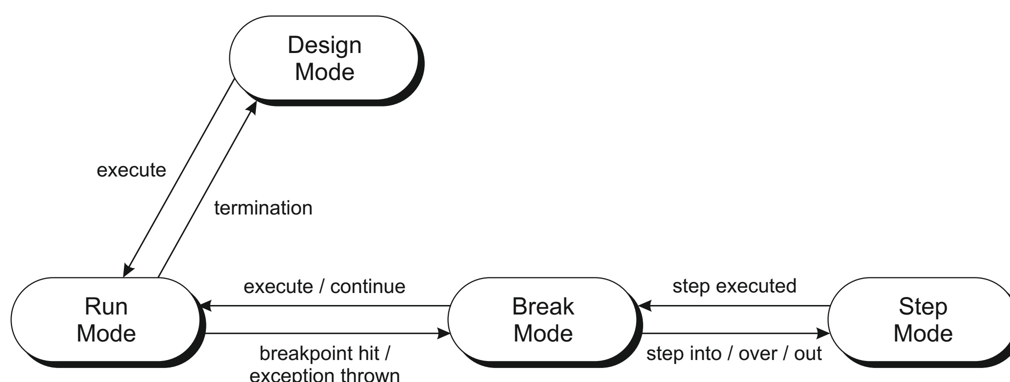
Program dále běží, dokud nenastanou okolnosti, které vyvolají tzv. událost zastavení (stopping event). Tou může být dosažení breakpointu, nebo vyvolání neošetřené výjimky. Debug engine v této chvíli komunikuje pomocí řady rozhraní s SDM s cílem předat SCM kontext, ve kterém se program v daný časový okamžik nachází. Kontext typicky obsahuje název zdrojového souboru s kódem, číslo řádku a sloupce, kde se nachází ukazatel instrukcí. Tyto informace IDE využívá ke zvýraznění řádku kódu, kde došlo k zastavení.

Debugger vstupuje v této situaci do break módu (Obrázek 16) a debug engine čeká na akci uživatele. Pokud ten volí procházení kódu po krocích (step into, step over, step out), SDM volá funkci *IDebugProgram2::Step* a předává jí argumenty *STEPUNIT* (instrukce, výraz, řádek) a *STEPKIND* (typ uživatelovy akce). Po dokončení kroku debug engine vyvolává opět událost zastavení. Rozhodne-li se uživatel pokračovat ve vykonávání programu, SDM zavolá funkci *IDebugProgram2::Execute* a program pokračuje v běhu, dokud nenastanou další okolnosti, které způsobí zastavení.

Během režimu zastavení probíhá vyhodnocování výrazů. Výrazem se zde rozumí řetězec získaný z oken **Watch**, **QuickWatch**, **Immediate Window**, nebo nacházející se v editoru pod kurzorem myši. Pro vyhodnocení výrazu debug engine využívá služeb expression evaluatoru. Výraz je reprezentován objektem *IDebugExpression2*, který obsahuje parsovaný výraz připravený k vyhodnocení v daném kontextu (časovém stavu programu reprezentovaném pomocí *IDebugExpressionContext2*). Vyhodnocený výraz vyjádřený jménem, typem a hodnotou proměnné je zobrazen v některém z oken uživatelského rozhraní.

Debug Package, Session Debug Manager a Process Debug Manager jsou komponenty, které jsou společné napříč všemi debuggery ve Visual Studiu. Tyto páteřní komponenty jsou ve Visual Studiu obsaženy a programátor při vývoji debuggeru pro vlastní jazyk nemusí do jejich implementace zasahovat.

Jeho úlohou však zůstává implementace Debug Enginu a Expression Evaluatoru. Debug engine je komponenta zacílená na specifický run-time – systém, nad kterým program



Obrázek 16: Stavový diagram debugger session

běží. Visual Studio obsahuje debug enginy, které jsou cílené na run-timy: CLR (.NET Framework) a Win32 (Windows). Jestliže nově zaváděný jazyk běží nad těmito run-timy, nemusí programátor implementovat vlastní debug engine, ale pouze expression evaluator.

Debug engine je z programového hlediska serverová COM komponenta. Úkolem vývojáře je zaregistrovat tuto komponentu jako class factory, a dále zaregistrovat ji do Visual Studia. Pomocí série přesně specifikovaných rozhraní musí nastavit porty debuggeru pro získání přístupu k laděným programům, zaregistrovat laděný program k portu tak, aby debugger mohl k programu přistupovat, zajistit vlastní připojení debuggeru a programu, a implementovat notifikační události informující o stavu ladění. Mezi další prvky implementace debug enginu patří vytvoření mechanismu pro kontrolu nad vykonáváním programu, správa breakpointů, pozastavení vykonávání a odpojení debuggeru.

Expression evaluator je součást, kterou je nutné implementovat pro každý jazyk, který je možné v debuggerem ladit. Tvoří pak součást debug enginu, která se stará o vyhodnocování syntaxe a sémantiky daného jazyka. Je opět implementována jako COM komponenta a musí být podobně jako debug engine zaregistrována. Začíná plnit svůj účel v momentě, kdy se laděný program zastaví vstupem do break módu. V této chvíli je množinou rozhraní debug enginu a expression evaluatoru získán kontext, ve kterém budou výrazy vyhodnocovány. Kontext je tvořen především stavem call stacku a místem, kde došlo k zastavení. Vyhodnocovaný výraz je z kontextu parsován a následně expression evaluatorem vyhodnocen. Úkolem vývojáře je implementace tohoto mechanismu pomocí přesně specifikovaných rozhraní. Součástí expression evaluatoru jsou také tzv. type visualizery. Ty fungují jako „překladače“ výsledků evaluace do čtivé podoby, která bude zobrazena v oknech IDE.

4 Implementace podpory jazyka e-PFL ve Visual Studiu

Tato kapitola popisuje některé z výše uvedených prvků podpory nového jazyka ve Visual Studiu v hlubších detailech na konkrétní implementaci podpory pro funkcionální jazyk e-PFL.

4.1 Funkcionální jazyky

Funkcionální programování je programátorská disciplína nad deklarativními jazyky. Základním principem je specifikace cíle výpočtu, ne postupu, jak k němu dojít, jako je tomu u imperativních programovacích jazyků. Na rozdíl od imperativních jazyků, kdy vykonávání programu spočívá v postupné změně stavu vnitřních proměnných, programy ve funkcionálních jazycích vyhodnocují výrazy (funkce) a tato vyhodnocení mohou probíhat i paralelně. Funkce jsou zde popisovány matematickým modelem vycházejícím z lambda-kalkulu. Funkce nemají tzv. vedlejší efekty, tzn., že nemění stav vykonávaného programu. Výstup z funkcí zde závisí čistě na vstupech a opakované volání funkce se stejnými argumenty vede vždy ke stejnému výsledku (referenční transparence). Funkce obvykle nejsou tvořeny složitými konstrukcemi v podobě cyklů apod., ale mohutně se zde využívá rekurze. Funkcionální jazyky také poskytují mnohem lepší možnosti abstrakce než klasické imperativní jazyky, program je možno budovat jako kompozici funkcí. Funkcionální programování je samo o sobě důležité pro některé oblasti informatiky, jako je umělá inteligence, formální specifikace a modelování nebo rychlé prototypování [3].

Problematika vývoje funkcionálního jazyka a jeho podpory v integrovaném vývojovém prostředí spočívá v těchto bodech [22]:

1. Přesná specifikace jazyka pomocí regulární gramatiky.
2. Implementace překladače s ohledem na cílovou platformu – vhodná implementace musí řešit problematiku plynoucí z podstaty funkcionálního jazyka:
   - pořadí vyhodnocování funkcí s ohledem na jejich závislosti
   - vhodná a optimální paralelizace funkcí
3. Volba vhodného a dostatečného rozšíření editoru s ohledem na možnosti jazyka
4. Implementace podpory ladění – debuggování funkcionálních jazyků je obecně obtížná úloha. Náročnost spočívá především v těchto rysech funkcionálních jazyků:
   - mnohonásobná rekurze
   - paralelizace vyhodnocování výrazů
   - postupné vyhodnocování (lazy evaluation)

4.2 Jazyk e-PFL

Funkcionální jazyky mají řadu zajímavých vlastností. Díky nim jsou vhodné pro specifikaci či modelování různých systémů, nebo rychlé vytváření prototypů. Tyto vlastnosti mohou být užitečné zejména pro oblast vestavných systémů. Těchto vlastností využívá Vestavný procesně funkcionální jazyk e-PFL (Embedded Process Functional Language), který primárně slouží k modelování vestavných systémů zejména v prvních fázích vývoje, čímž je využitelný k eliminování různých vývojových rizik [5].

Jazyk e-PFL vznikl postupnou evolucí z jazyků PFL a PPFL. Syntaxe a sémantika vychází z jazyka PFL (vycházejícího z čistě funkcionální podmnožiny jazyka Haskell) rozšířeného zejména o konstrukce pro modelování vestavných systémů.

Jazyk je přesně specifikován gramatikou uvedenou v příloze A.

Pro jazyk e-PFL již byl realizován kompilátor pracující nad touto gramatikou. Kompilátor jazyka je implementován v jazyce C# a lze jej provozovat na platformě .NET. Pro program v jazyce e-PFL je vygenerován cílový kód v jazyce C#. K tomuto vygenerovanému kódu je pak připojeno jednoduché běhové prostředí a tento výsledek lze zkompilovat překladačem jazyka C# a spustit.

4.3 Integrace jazyka e-PFL do prostředí Visual Studia

Integrace jazyka e-PFL do prostředí Visual Studia 2010 byla navržena podle principů popsaných v předchozích kapitolách.

V této práci byly vyřešeny tyto úlohy:

1. Návrh a implementace podpory pro správu projektů v jazyce e-PFL
2. Zakomponování existujícího kompilátoru e-PFL do prostředí Visual Studia
3. Návrh a implementace rozšíření editoru Visual Studia poskytující základní funkce pro podporu vývoje v jazyce e-PFL
4. Řešení pro distribuci kompletní podpory e-PFL na cizí počítač

Tato práce neřeší podporu pro ladění (debuggování) kódu napsaného v jazyce e-PFL. Jak již bylo zmíněno v kapitole 3.4, jedná se o poměrně náročnou úlohu přesahující rozsah této práce. Ve zmíněné kapitole lze nalézt stručný náhled na způsob rozšíření Visual Studia o debugger. V případě jazyka e-PFL je nutné tuto úlohu ještě rozšířit o problematiku ladění funkcionálních jazyků.

Navržená podpora e-PFL ve Visual Studiu není jistě vyčerpávající. Další možnosti rozšíření budou načrtnuty v závěru této práce.

4.3.1 Struktura rozšíření

Implementované řešení se skládá ze tří samostatných projektů:

- *EPFL-Project*

- *EPFL-EditorExtensions*
- *Microsoft.VisualStudio.Project*

Všechny 3 projekty jsou po překladu sestaveny do samostatných DLL knihoven. Z projektů *EPFL-Project* a *EPFL-EditorExtensions* jsou navíc sestaveny VSIX balíčky, které budou následně distribuovány jako hlavní součásti instalace při integraci do Visual Studia na cizí počítače.

Knihovna *EPFL-Project* obsahuje řešení pro správu projektů a jejich překlad a spouštění. Je závislá na knihovně *Microsoft.VisualStudio.Project*, která poskytuje framework pro generování projektů na základě šablon, jejich správu, perzistenci, přidávání dalších prvků, apod. (viz. kapitola 3.1.2). Implementace této knihovny nebyla součástí této práce, ale byla stažena jako zdroj pro podporu vývoje rozšíření ze stránek Microsoftu.

Knihovna *EPFL-ProjectExtensions* je tvořena množinou vzájemně komunikujících tříd, které tvoří systém rozšiřující editor Visual Studia o podporu jazyka e-PFL.

4.3.2 Správa projektů

Pro jazyk e-PFL byly navrženy a implementovány 2 šablony projektů, na jejichž bázi může koncový uživatel zahájit výstavbu projektu.

První šablonou je **Main Project Application**. Bude představovat zřejmě nejčastěji používaný typ projektu. Její součástí je hlavní projektový soubor **Main.pfl** a soubor **Prelude.pfl**, který slouží jako knihovna základních typů a funkcí.

V definici šablony (viz. výpis 13) je nastaveno otevření souboru Main.pfl v editoru po spuštění. Předvyplněným obsahem souboru je konstrukce „import“, která připojuje definice ze souboru Prelude.pfl, a dále začátek konstrukce „main“. První řádek obsahuje komentář vyplněný názvem souboru. Tento název je zde doplněn z atributu *$MainClassName$*. Atribut a jeho hodnota jsou definovány v tagu `<CustomParameter>`.

```
-- e-PFL application file: $MainClassName$
import "Prelude.pfl"

main =
```

Výpis 12: Defaultní obsah souboru Main.pfl

Druhou šablonou je **Library e-PFL Projec**t. Tato šablona zakládá jednoduchý projekt, který je tvořen pouze jediným prázdným souborem **Library.pfl**.

```
<VSTemplate Version="2.0.0" Type="Project"
    xmlns="http://schemas.microsoft.com/developer/vstemplate/2005">
  <TemplateData>
    <Name>Main Project Application</Name>
    <Description>Main E-pfl project application</Description>
    <Icon>MainProject.ico</Icon>
    <ProjectType>E-pfl</ProjectType>
    <SortOrder>10</SortOrder>
```

```
    </TemplateData>
    <TemplateContent>
      <Project File="MainProject.pflproj" ReplaceParameters="true">
        <ProjectItem ReplaceParameters="true" OpenInEditor="true">
          Main.pfl
        </ProjectItem>
        <ProjectItem ReplaceParameters="false">
          Prelude.pfl
        </ProjectItem>
      </Project>
      <CustomParameters>
        <CustomParameter Name="$MainClassName$" Value="Main"/>
      </CustomParameters>
    </TemplateContent>
  </VSTemplate>
```

Výpis 13: Ukázka šablony *.vstemplate* projektu Main Project Application

Obě šablony obsahují dále ikony, které budou zobrazeny v okně **New Project**, a hlavní projektový soubor typu *.pflproj*.

Jako prvek (item), který je možno do projektů v průběhu vývoje přidávat, je definovaný jediný prvek nazvaný **Source File**. Je rovněž definován šablonou a jeho obsah je tvořen souborem *SourceFile.pfl* obsahující v komentáři název souboru zadaný uživatelem v dialogovém okně **Add New Item**. Tento název je sem přenesen pomocí atributu *$safeitemname$*.

Mechanismus správy projektů je tvořen soustavou 3 tříd vybudovaných nad frameworkem *Microsoft.VisualStudio.Projec*t (Obrázek 5):

- *EPFL_ProjectPackage*
- *EPFL_ProjectFactory*
- *EPFL_ProjectNode*

Jejich úkolem je zprostředkovat založení, ukládání a kompletní správu projektů. Jejich činnost je podrobně popsána v kapitole 3.1.2.

Dále byla implementována třída *GeneralPropertyPage* odvozená ze třídy *SettingsPage* (Obrázek 6). Definovány jsou zde 2 vlastnosti projektu:

- **AssemblyName** – vlastnost pouze pro čtení. Zobrazuje název projektu, tak jak byl uživatelem zadán. Je získán z projektového *.pflproj* souboru, kde je definován jako projektová proměnná *$safeprojectname$*.
- **AdditionalInfo** – vlastnost projektu, která je defaultně bez hodnoty. Byla zavedena z důvodu případné nutnosti přidání určitých doplňkových informací k projektu uživatelem.

4.3.3 Připojení kompilátoru

Tato práce se nezabývá problematikou vlastní implementace kompilátoru. Pro jazyk e-PFL již existuje hotový kompilátor (*PPFL.exe*) připravený vedoucím této práce. Jedná se o konzolovou aplikaci, která po spuštění s parametrem, který obsahuje plnou cestu k překládanému souboru *.pfl*, zajistí jeho překlad do kódu jazyka C# a následně do spustitelného *.exe* souboru. Vygenerované soubory jsou uloženy ve stejné složce jako překládaný *.pfl* soubor.

Hlavní myšlenka připojení tohoto kompilátoru do Visual Studia spočívá ve spouštění aplikace *PPLF.exe* s potřebnými argumenty jako externí proces a přesun výstupů na místo očekávané Visual Studiem.

Překlad je řízen tzv. tasky, které jsou definovány soubory *.targets*. V případě jazyka e-PFL se jedná o jeden task. Jeho popis je obsažen v souboru *EPFL.targets*. Definuje jednoduché nasměrování na třídu, která překlad zajišťuje, a způsob plnění hodnot projektových proměnných.

EPFL.targets je definiční soubor společný všech typům projektů a cesta k němu je definovaná v projektových souborech *.pflproj*. Jedná se zde však o absolutní cestu, což může způsobovat problémy při instalaci na cizí počítače. Řešení, které zde bylo použito, spočívá ve vytvoření složky *„E-PFL“* v systémové složce *„ProgramFiles“*. Cesta na tuto složku je uložena v systémové proměnné a její hodnotu tudíž lze snadno získat. Složka *„E-PFL“* byla dále použita jako úložiště vlastního kompilátoru *PPFL.exe*.

```
<Import Project="$(ProgramFiles)\E-PFL\EPFL.targets" />
```

Výpis 14: Ukázka definice cesty k souboru *.targets* ze souboru *.pflproj*

Celá logika spouštění a přesunu výstupů je implementována v metodě *Execute()* třídy *EPFL_CompilerTask*. Tato třída rovněž zajišťuje několik parametrů potencionálně potřebných pro překlad. Těmi jsou naplněny definované vlastnosti (properties) třídy *EPFL_CompilerTask*. Seznam těchto properties je popsán v kapitole 3.2.2. Naplnění vlastností hodnotami je definováno v souboru *EPFL.targets* v tagu `<EPFL_CompilerTask>`.

Tato implementace připojení kompilátorů používá pouze tři z popsaných properties, a sice:

- *ProjectPath* – její hodnota představuje plnou cestu ke složce s překládaným projektem
- *SourceFiles* – kolekce zdrojových souborů k překladu. Její první prvek představuje hlavní soubor vyvíjené aplikace.
- *OutputAssembly* – plný název (včetně cesty) cílové assembly

Kombinací prvních dvou hodnot je odvozen argument použitelný při volání kompilátoru.

Cesta ke kompilátoru uloženého ve složce *„../ProgramFiles/E-PFL"* je získána ze systémové proměnné a proces je spuštěn ve skrytém režimu. Standardní výstup z kompilátoru je zachycován až do ukončení běhu (viz. výpis 15).

```
string argumentString = String.Format("{0}\\{1}", projectPath, sourceFiles[0]);
Process build = new Process();
build.StartInfo.UseShellExecute = false;
build.StartInfo.RedirectStandardOutput = true;
build.StartInfo.CreateNoWindow = true;
string utilsDirectory = Environment.GetFolderPath(Environment.SpecialFolder.
    ProgramFiles);
build.StartInfo.FileName = Path.Combine(utilsDirectory, @"E-PFL\PPFL.exe");
build.StartInfo.Arguments = String.Format("\"{0}\"", argumentString);
build.Start();
string output = build.StandardOutput.ReadToEnd();
build.WaitForExit();
```

Výpis 15: Ukázka algoritmu pro spuštění překladu

Visual Studio ale standardně očekává výstup z překladu ve složce *„obj/Debug"* pod hlavní projektovou složkou. Rovněž název přeloženého souboru musí korespondovat s názvem assembly definovaným uživatelem (obvykle shodný s názvem projektu). Nejsou-li tyto podmínky splněny, pokus o spuštění vyvinuté aplikace z prostředí Visual Studia skončí po překladu chybou.

Aplikace kompilátoru je naproti tomu implementovaná pro spouštění z příkazové řádky a generování výstupních souborů do stejné složky jako zdrojový soubor. Jméno výstupního souboru koresponduje se zdrojovým souborem. Výše uvedený postup je tedy nutné pravidlům Visual Studia přizpůsobit.

Přizpůsobení (viz. výpis 16) spočívá v získání názvu vygenerovaného *.exe* souboru včetně cesty k němu, příprava cesty ke složce, kde Visual Studio výstup očekává, a přesun souboru do této složky.

```
string outputfile = sourceFiles[0].Substring(0, sourceFiles[0].Length - 3) + "exe";
string oldPath = String.Format("{0}\\{1}", projectPath, outputfile);
string newPath = String.Format("{0}\\{1}", projectPath, outputAssembly);
if (File.Exists(oldPath))
{
    File.Move(oldPath, newPath);
}
```

Výpis 16: Ukázka přesunu a přejmenování přeloženého souboru do očekávané složky

Posledním problémem zůstává výpis zpráv kompilátoru v prostředí Visual Studia. Aplikace kompilátoru poskytuje jednoduchý textový výstup zpravující o procesu překladu. Tento výstup je během překladu zachycován a bude vypsán do dialogu **Output** Visual Studia. Referenci na objekt **Output** dialogu lze z Visual Studia získat a plnit jej zachyceným výstupem. Výstup lze dále směrovat na různé záložky Output dialogu

(Info, Warning, Error) podle typu výstupu, nebo dokonce připravit vlastní záložku se specifickým výstupem. Implementovaný kompilátor poskytuje pouze základní výstup o překladu, a proto byl implementován pouze výpis do základní záložky.

```
var outputWindow = Package.GetGlobalService(typeof(SVsOutputWindow)) as
    IVsOutputWindow;
if (outputWindow != null)
{
    Guid guidBuildPane = Microsoft.VisualStudio.VSConstants.
        OutputWindowPaneGuid.BuildOutputPane_guid;
    IVsOutputWindowPane buildPane;
    outputWindow.GetPane(ref guidBuildPane, out buildPane);
    buildPane.OutputString(output);
}
```

Výpis 17: Ukázka směrování výpisu kompilátoru do dialogu Output

4.3.4 Rozšíření editoru o podporu e-PFL

Většina funkcí běžně poskytovaných editory v IDE je použitelná i pro jazyk e-PFL. Přestože jazyk má svá specifika, je přesně definován syntaxí (příloha A), nad kterou lze vybudovat aparát pro poskytování různých funkcí editoru. Celá problematika rozšíření editoru Visual Studia a obecný postup pro implementaci jednotlivých funkcí byl uveden v kapitole 3.3. Tato část se zabývá způsobem implementace funkcí editoru navržených pro jazyk e-PFL.

Pro tuto verzi podpory e-PFL v editoru Visual Studia bylo implementováno barvení syntaxe, zvýrazňování chyb, sbalování kódu, a jedna z funkcí Intellisense – automatické doplňování. Jelikož většina těchto funkcí využívá společný mechanismus pro procházení a analýzu kódu k označování bloků textu metainformacemi, navržené řešení spočívá v implementaci taggeru označujícího úseky textu obecnými tagy (princip na obrázku 12). Takto označený celý text v editoru bude využíván speciálními taggery jednotlivých funkcí. Ty budou z těchto informací čerpat při zpracovávání vlastního algoritmu značení. Navržená architektura je plně v souladu s architekturou MEF a je tímto způsobem záměrně implementována z důvodu případného rozšíření editoru o další funkce. Při něm může být snadno využito již existujících značek namísto implementace vlastního mechanismu značkování bez výrazného narušení výkonu a efektivity editoru.

Celé řešení je uloženo v projektu *E-PFL_EditorExtensions* a rozděleno do několika namespaces.

4.3.4.1 Analyzér zdrojového kódu

Namespace *E-PFL_EditorExtensions.Analyzer* obsahuje třídy podílející se na algoritmu průchodu textem v editoru a jeho rozkladu na lexikální a syntaktické struktury.

Nejvyšší instancí je zde třída *Syntax*, která řídí celý průchod textem. Pro analýzu textu používá algoritmus syntaktické analýzy shora dolů – tzv. rekurzivní sestup. Tato

metoda spočívá v zápisu samostatných procedur pro analýzu každého neterminálního symbolu gramatiky. Překlad se pak spustí voláním procedury odpovídající startovacímu neterminálnímu symbolu [Beneš, Překladače].

Při čtení kódu jsou jednotlivé lexikální symboly získány pomocí objektu třídy *Lex*. Ta postupně čte vstupní sekvenci znaků, a skupinu znaků identifikovanou jako lexikální jednotka vrací coby další přečtený lexikální symbol.

Tento algoritmus a třídy, které ho zpracovávají, byly převzaty z analyzéru v existujícím kompilátoru jazyka e-PFL. Pro potřeby editoru Visual Studia však musely být upraveny. Především označování sekvencí kódu identifikovaných jako lexikální jednotky výčtovým typem *Symbol* (viz. tabulka symbolů – tabulka 1) zde bylo nedostatečné. Namísto něj jsou sekvence označovány objekty třídy *LexUnit*, sloužící jako wrapper pro *Symbol*, a zároveň přidávající další nezbytné informace:

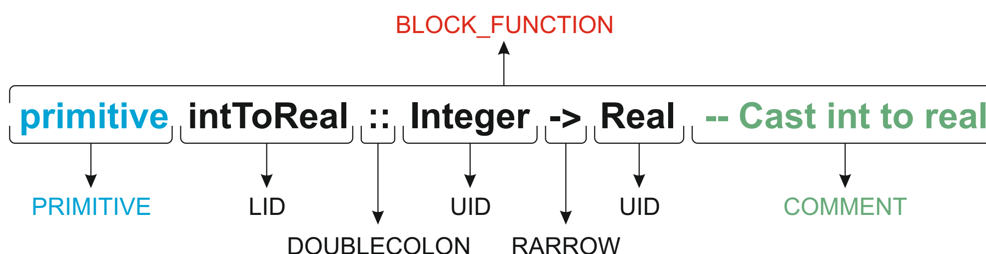
- *Start* – pozice začátku lexikálního symbolu v celé vstupní sekvenci
- *Length* – délka lexikálního symbolu
- *InfoText* – doplňková textová informace

EOF	SEMICOLON	WHEN
LID	UNDERLINE	ENCLOSED
UID	RARROW	INHERITS
DATA	DOUBLECOLON	WHERE
EQUAL	PRIMITIVE	OP
LPAR	INT_VALUE	IMPORT
RPAR	REAL_VALUE	LET
CASE	CHAR_VALUE	IN
OF	STRING_VALUE	COMMENT
COMMA	LSPAR	BLOCK_IMPORT
BAR	RSPAR	BLOCK_DATA
UNIT	LABEL	BLOCK_FUNCTION
MAIN	IF	BLOCK_ERROR
LCPAR	THEN	
RCPAR	ELSE	

Tabulka 1: Tabulka typů lexikálních symbolů

Mimo označování vlastních lexikálních symbolů jsou objekty typu *LexUnit* použity i pro označování vyšších syntaktických struktur jako definice importů, definice dat, a definice funkcí. Tyto vyšší jednotky nemusí být editorem nutně využívány, mohou však sloužit pro některé případné nové funkce editoru při jeho eventuálním dalším rozšíření (Obrázek 17).

Dalším zde upraveným chováním analyzéru bylo vyřešení chování při výskytu syntaktické chyby. Zatímco předchozí řešení používalo systém generování výjimek, zde bylo



Obrázek 17: Ukázka práce analyzéru – označování textu symboly

navrženo označení chybného úseku jednotkou typu BLOCK_ERROR a doplněním informace o druhu chyby. Podle typu syntaktické chyby je takto označován celý řádek kódu, nebo úsek od začátku výskytu chyby až po konec řádku.

Poslední zásadnější úpravou bylo doplnění lexikálního symbolu COMMENT oproti původně implementovanému ignorování komentářů, což bylo výhodné řešení pro kompilátor, ne však pro editor.

Výsledkem práce analyzéru je kolekce lexikálních jednotek, která kompletně pokrývá celý text.

4.3.4.2 Obecný tagger

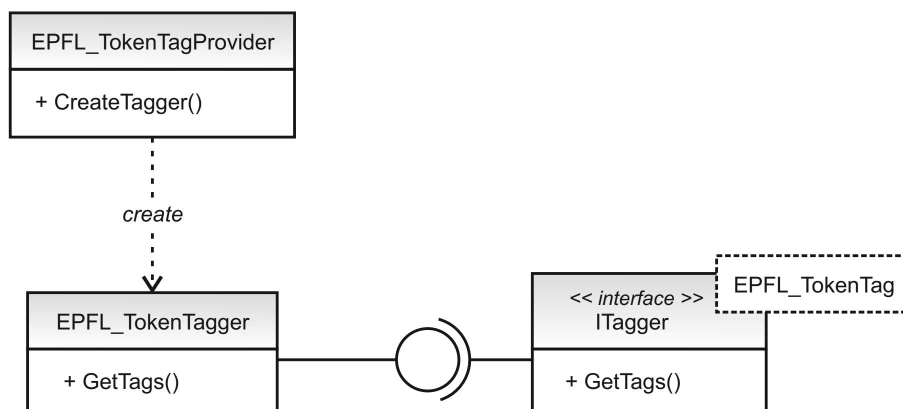
Obecný tagger je struktura poskytující taggovací službu specializovaným taggerům. Jeho úkolem je na žádost editoru analyzovat existující text v editačním okně a označit jej doplňkovými informacemi, které budou sloužit jako vstupní informace specializovaným taggerům.

Editor Visual Studia spouští žádost o tuto službu v okamžiku změny textu. Tagger pak iteračně vrací kolekci označkovaných úseků textu jako návratovou hodnotu volání metody *GetTags()*. Výsledky značkování pak specializované taggery konzumují skrz tzv. agregátor.

Obecný tagger implementovaný pro jazyk e-PFL je dále nazýván jako „token tagger“ a je reprezentován třídou *EPFL_TokenTagger*. Ta implementuje generické rozhraní *ITagger* s parametrem *EPFL_TokenTag*. Token tag zde reprezentuje značku, kterou jsou úseky textu označeny. Tyto tagy jsou dále zpřesněny typem informace (typem značky), které reprezentují. Typy token tagů jsou popsány výčtovým typem *EPFL_TokenTypes*. Přístup k token taggeru je zprostředkován poskytovatelem taggeru – třídou *EPFL_TokenTagProvider*, která je exportována jako komponenta MEF. Celá struktura je vidět na diagramu 18.

Při změně textu v editoru Visual Studia dojde k aktivaci algoritmu v metodě *GetTags()*. Algoritmus pracuje následovně:

Nejprve je získán text aktuálního snímku (*SnapShot*) editoru. Je vytvořen lexikální analyzér (*Lex*) a tento text je mu v konstruktoru předán. Dále je vytvořen syntaktický analyzér (*Syntax*) a je mu předán právě vytvořený lexikální analyzér. Ze syntaktického analyzéru je vyžádána kolekce všech lexikálních jednotek (*LexUnit*) v textu. Tato kolekce je v krocích iterována a informace v lexikálních jednotkách jsou postupně překlápěny



Obrázek 18: Class diagram struktury obecného Token taggeru

do vytvářených značek (*TagSpan<EPFL_TokenTag>*) v textu nad daným snímkem. Tyto značky jsou pak v krocích vráceny jako návratový typ metody (viz. výpis 18).

```
public IEnumerable<ITagSpan<EPFL_TokenTag>> GetTags(
      NormalizedSnapshotSpanCollection spans)
{
   foreach (SnapshotSpan curSpan in spans)
   {
      string editorText = curSpan.Snapshot.GetText();
      Lex lexer = new Lex(new StringReader(editorText));
      Syntax syntaxTagger = new Syntax(lexer);
      List<LexUnit> tokens = syntaxTagger.parse();
      foreach (LexUnit token in tokens)
      {
         if (token.Start >= curSpan.Snapshot.Length)   continue;
         int length = token.Start + token.Length >= curSpan.Snapshot.Length ?
               curSpan.Snapshot.Length − token.Start : token.Length;
         var tokenSpan = new SnapshotSpan(curSpan.Snapshot, new Span(token.Start,
               length));
         if (tokenSpan.IntersectsWith(curSpan))
         {
            EPFL_TokenTypes tagType = (EPFL_TokenTypes)Enum.Parse(typeof(
                  EPFL_TokenTypes), token.LexSymbol.ToString());
            yield return new TagSpan<EPFL_TokenTag>(tokenSpan, new
                  EPFL_TokenTag(tagType, token.InfoText));
         }
      }
   }
}
```

Výpis 18: Metoda *GetTags()* Token taggeru

4.3.4.3 Classification tagger

Klasifikační tagger je specializovaný tagger, který poskytuje editoru úseky textu označené klasifikačními tagy (třída *ClassificationTag*). Je tvořen strukturou tříd obsažených v namespace *E-PFL_EditorExtensions.Classification*.

Struktura tohoto taggeru je podobná struktuře Token taggeru (Obrázek 18). Provider zde reprezentuje třída *EPFL_ClassifierProvider*, samotný tagger pak zprostředkovává třída *EPFL_ClassificationTagger*.

Klasifikační tagger, jako i další specializované taggery, je zde navržen tak, že využívá služeb obecného Token taggeru. Toho je dosaženo využitím agregátoru, který poskytuje tagy generované Token taggerem. Agregátor je poskytnut objektem rozhraní *IBufferTagAggregatorFactoryService*, který je v provideru importován z MEF kontejneru. Stejně tak je zde importováno rozhraní služby pro správu všech klasifikačních typů v editoru (*IClassificationTypeRegistryService*).

```
[Import]
internal IClassificationTypeRegistryService ClassificationTypeRegistry = null;

[Import]
internal IBufferTagAggregatorFactoryService aggregatorFactory = null;

public ITagger<T> CreateTagger<T>(ITextBuffer buffer) where T : ITag
{
   ITagAggregator<EPFL_TokenTag> EPFL_TagAggregator = aggregatorFactory.
        CreateTagAggregator<EPFL_TokenTag>(buffer);
   return new EPFL_Classifier(buffer, EPFL_TagAggregator, ClassificationTypeRegistry)
        as ITagger<T>;
}
```

Výpis 19: Ukázka vytvoření agregátoru ve třídě *EPFL_ClassifierProvider*

Vlastní algoritmus klasifikace je opět v těle metody *GetTags()* (viz. ukázka části kódu – výpis 20). ClassificationTag, kterým je úsek textu, který má být v editoru obarven, je označen typem *IClassificationType*. Tyto klasifikační typy jsou exportovány do MEF kontejneru, odkud jsou podle názvu získávány pomocí služby pro správu klasifikačních typů. Rozhodnutí o označení úseku textu klasifikačním tagem je jednoduše učiněno na základě typu Token tagu. Začátek i délka klasifikačního tagu tagu jsou převzaty z Token tagu.

```
public IEnumerable<ITagSpan<ClassificationTag>> GetTags(
      NormalizedSnapshotSpanCollection spans)
{
   foreach (var tagSpan in this.aggregator.GetTags(spans))
   {
      var tagSpans = tagSpan.Span.GetSpans(spans[0].Snapshot);

      if (EPFL_Language.Keywords.Values.Contains(tagSpan.Tag.Type))
      {
```

```
            yield return new TagSpan<ClassificationTag>(tagSpans[0],
            new ClassificationTag(typeService.GetClassificationType("EPFL_keyword")));
        }
        ......
```

Výpis 20: Část metody *GetTags()* klasifikačního taggeru

Pro jazyk e-PFL jsou klasifikačními tagy označeny tyto Token tagy:

Token tag	Classification tag	Zvýraznění v editoru
Všechny tagy reprezentující klíčová slova jazyka	EPFL_keyword	Modrý text
COMMENT	EPFL_comment	Zelený text
STRING_VALUE	EPFL_stringValue	Červený text

Tabulka 2: Tabulka typů klasifikačních tagů

Statická třída *EPFL_ClassificationDefinition* obsahuje definice klasifikačních typů. Definice formátování jednotlivých typů jsou pak specifikovány v samostatných třídách exportovaných do kontejneru MEF.

Poznámka: V průběhu práce na dalších specializovaných taggerech (Error a Outlining), které využívají vyšších syntaktických struktur, než jsou základní lexikální jednotky, byly definovány typy i formátování i pro tyto struktury. Při definici formátování na podbarvování pozadí takto označovaných úseků textu, lze snadno v editoru vidět rozsah této struktury. Využití této možnosti se hlavně v počátku implementace ukázalo jako velmi výhodné pro odhalení chyb ve značení. Později byly tyto klasifikační typy i formátování odstraněno.

```
[Export(typeof(ClassificationTypeDefinition))]
[Name("EPFL_keyword")]
internal static ClassificationTypeDefinition keyword = null;

[Export(typeof(EditorFormatDefinition))]
[ClassificationType(ClassificationTypeNames = "EPFL_keyword")]
[Name("EPFL_keyword")]
[UserVisible(false)]
[Order(Before = Priority.Default)]
internal sealed class Keyword : ClassificationFormatDefinition
{
    public Keyword()
    {
        this.DisplayName = "EPFL_keyword";
        this.ForegroundColor = Colors.Blue;
    }
}
```

Výpis 21: Ukázka definice klasifikačního typu a definice jeho formátování

4.3.4.4 Error tagger

Error tagger je další specializovaný tagger navržený pro jazyk e-PFL. Jeho úkolem je poskytovat službu, která označuje úseky textu v editoru, které jsou analýzerem vyhodnoceny jako chybné. Tyto úseky musí být označeny značkou *ErrorTag*. Takto označené části textu chápe editor automaticky jako chybné a zvýrazňuje je podtržením červenou vlnitou čarou.

Error tagger je realizován množinou tříd umístěných v namespace *E-PFL_Editor-Extensions.ErrorTagging*. Principiálně se neliší od předchozích taggerů (Obrázek 18). Je zde opět využito agregátoru poskytujícího kolekci Token tagů označujících celý text v editoru. Jednoduchý algoritmus v těle metody *GetTags()* zde překlápí Token tagy typu BLOCK_ERROR na Error tagy. Třída *ErrorTag* umožňuje tagům přiřadit i objekt nesoucí doplňující informace o chybě. Tyto informace jsou pak uživateli zobrazeny, najede-li ukazatelem myši na část kódu označeného jako chybný. Použitým objektem je zde textový řetězec obsažený v atributu *InfoText* Token tagu, který je do něj přenesen analyzérem v době nalezení chyby. Obsažená informace obsahuje stručný důvod ke vzniku chyby a slouží tak uživateli k rychlejšímu zajištění opravy.

```
public IEnumerable<ITagSpan<ErrorTag>> GetTags(
      NormalizedSnapshotSpanCollection spans)
{
   foreach (var tagSpan in this.aggregator.GetTags(spans))
   {
      var tagSpans = tagSpan.Span.GetSpans(spans[0].Snapshot);

      if (tagSpan.Tag.Type == EPFL_TokenTypes.BLOCK_ERROR)
      {
         yield return new TagSpan<ErrorTag>(tagSpans[0], new ErrorTag("Syntax␣
              Error", tagSpan.Tag.InfoText));
      }
   }
}
```

Výpis 22: Metoda *GetTags()* Error taggeru

Samotný tagger je reprezentován třídou *EPFL_ErrorTagger* implementující rozhraní *ITagger<ErrorTag>*. Její poskytovatel je implementován třídou *EPFL_ErrorTaggingProvider*.

4.3.4.5 Outlining tagger

Posledním taggerem navrženým pro jazyk e-PFL je Outlining tagger poskytující sbalování kódu víceřádkových funkcí.

Použité principy jsou i zde shodné s předchozími taggery s využitím agregátoru. Množinu tříd poskytující službu sbalování kódu lze nalézt v namespace *E-PFL_Editor-Extensions.Outlining* a tvoří ji poskytovatel taggeru *EPFL_OutliningTaggingProvider* a vlastní tagger *EPFL_OutliningTagger* implementující rozhraní *ITagger<OutliningRegionTag>*.

Metoda *GetTags()* překlápí Token tagy typu BLOCK_FUNCTION na tagy třídy *OutliningRegionTag*. Takto označené úseky editor automaticky vyhodnocuje jako „sbalitelné“ a zobrazuje nalevo od jejich prvního řádku ikonku „plus“, kterou uživatel může funkci sbalit do jednoho řádku.

Přestože prosté překlopení Token tagů do outlining tagů je fungujícím řešením, ukázalo se, že vykazuje v některých situacích značnou nestabilitu. Řešením bylo přeznačení začátků/konců úseků sbalitelných funkcí při překlápění na pozice začátků/resp. konců řádků, na kterých se nacházejí.

Dále zde bylo implementováno ignorování jednořádkových funkcí, jejichž sbalení by nemělo přínos. Rovněž jsou z outlining bloku vyřazeny prázdné řádky, které mohou být analyzérem přiřazeny k bloku funkce.

Třída *OutliningRegionTag* umožňuje mimo jiné definovat parametrem *CollapsedForm* objekt, kterým bude nahrazen sbalený úsek. Navržené řešení tento parametr plní textovým řetězcem obsahujícím název funkce. Ten je převzat z atributu *InfoText* Token tagu, který je do něj přenesen analyzérem v době syntaktické analýzy funkce.

```
test  x a = z where                ⇒              Function 'test' ...
                z = x + y
                    where
                    y = a + x
```

Výpis 23: Ukázka víceřádkové funkce jazyka e-PFL před a po sbalení

4.3.4.6 Intellisense

Pro podporu jazyka e-PFL byla implementována jediná funkce z rodiny funkcí Intellisense, a sice doplňování výrazů (statement completion).

Funkce funguje podobně jako u jiných jazyků, zde ovšem nabízí k doplnění pouze klíčová slova jazyka e-PFL. Seznam klíčových slov, kterým je nabídka pro doplňování plněna, je definován ve statické třídě *EPFL_Language* jako kolekce objektů typu *Completion*.

Celé řešení tvoří třídy *EPFL_CompletionHandlerProvider* a *EPFL_CommandHander* představující Controller, a *EPFL_CompletionSourceProvider* a *EPFL_CompletionSource*, které tvoří Source (viz. kapitola 3.3.2.6).

Třídy tvořící controller zajišťují vytvoření a zánik session (*ICompletionSession*), která zprostředkovává zobrazení nabídky výrazů pro doplnění. Session je spuštěna, napíše-li uživatel v editoru znak, který je písmenem. Hlavní logika je řízená metodou *Exec()*. Třídy tvořící completion source dodají seznam klíčových slov pro doplnění. Controller ve svém atributu *ApplicableTo* udržuje sekvenci znaků dosud napsaných v editoru, které budou doplňovaným výrazem nahrazeny. Controller také seznam slov pro doplnění filtruje podle atributu *ApplicableTo*.

Session úspěšně zaniká (commit), napíše-li uživatel celé klíčové slovo sám (nabízený seznam má pouze 1 položku a ta je rovna hodnotě *ApplicableTo*), nebo vybere-li některou z nabízených položek a volbu potvrdí klávesami RETURN, TAB, SPACE nebo některou z kláves s interpunkčními znaky.

Zrušení session (dismiss) lze provést klávesou ESC.

4.3.5 Deployment

Rozšíření Visual Studia může být distribuováno jako tzv. VSIX balíček. Jedná se o takový druh jednotky nasazení, které Visual Studio rozeznává a automaticky jej instaluje do složky určené pro rozšiřující balíčky, odkud je pak ihned začne využívat *(%LocalAppData%/Microsoft/VisualStudio/10.0/Extensions/<Company>/<Product>/ <Version>)*.

Prakticky se jedná o *.zip* soubor dodržující standard Open Packaging Convention. V souboru musí být obsaženy tyto součásti:

- [Content_Types].xml – soubor obsahující seznam typů souborů obsažených v balíčku
- extension.vsixmanifest – dokument popisující balíček
- podpůrné soubory balíčku – obrázky, ikony, licenční ujednání, apod.
- vlastní soubory, které balíček doručuje

Po instalaci je možné spravovat instalovaná rozšíření z nástroje **Extension Manager**, který je součástí Visual Studia.

Z projektů *E-PFL_Project* a *E-PFL_EditorExtensions* z řešení pro jazyk e-PFL jsou při sestavení generovány samostatné VSIX balíčky. Jelikož se zde jedná o komplexní řešení pro podporu daného jazyka, je výhodné je sloučit v jeden balíček a tím zjednodušit celou instalaci.

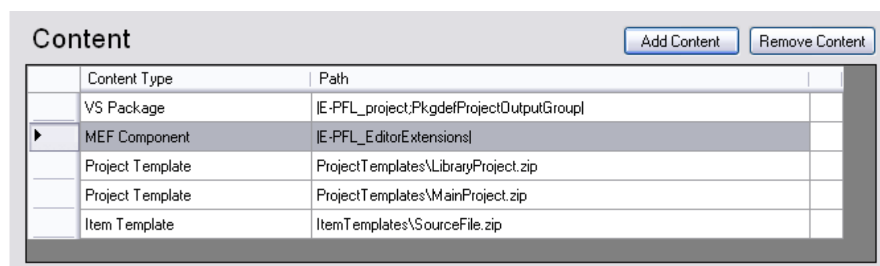
Nastavení vlastností balíčku VSIX je uloženo v souboru *source.extension.vsixmanifest*, který je součástí každého balíčku. Visual Studio poskytuje pro tento typ souboru zvláštní designer a veškeré nastavení se tak významně zjednodušuje.

Mezi základní možnosti nastavení patří:

- Název rozšíření, jeho verze, autor, a popis
- Výběr edice Visual Studia, kde bude rozšíření možné nasadit
- Možnost přiložení souboru s licenčními ujednáními
- Možnost přiložení vlastní ikony a obrázku náhledu (budou zobrazeny v Extension Manageru)
- URL, kde se lze dozvědět o rozšíření více

Vlastní obsah balíčku je definován v sekci *Content*. Binární soubory projektu, ze kterého se VSIX tvoří, jsou zde předvyplněny automaticky, další je v případě potřeby možné přidat. Pro každou položku je nutné specifikovat druh přidaného obsahu.

V případě řešení pro e-PFL byly přidány zip soubory jednotlivých projektových šablon a itemů, a dále projekt *E-PFL_EditorExtensions* jako MEF komponenta (viz. Obrázek 19).



Obrázek 19: Nastavení obsahu VSIX balíčku pro e-PFL

Veškeré položky rozšíření jsou tímto obsaženy v jednom VSIX balíčku.

Jedinými samostatnými součástmi jsou už jen soubor kompilátoru *PPFL.exe* a soubor *EPFL.targets* s parametry pro překlad. Na tyto soubory se řešení obsaženo v balíčku odvolává pomocí absolutních cest a budou tedy umístěny na výhodném a snadno dosažitelném místě – ve složce *ProgramFiles/E-PFL*.

5 Závěr

Rozšíření integrovaného vývojového prostředí o podporu nového jazyka není snadnou programátorskou úlohou. Ať už se jedná o jakkoli vyspělou úroveň podpory, vždy jde o komplexní úlohu, kladoucí na programátora určité nároky, zejména na znalost cílového prostředí a na znalost specifik této úlohy. Jelikož se nejedná o standardní úkol, který je běžně v praxi řešen, zdroje těchto znalostí nebývá snadné nalézt.

Podobně tomu bylo i v době vzniku této práce. Přestože prostředí Visual Studia je dostatečně známé a v literatuře dobře dokumentované, verze 2010 přináší změny právě na poli možností jeho rozšíření. Jedná se především o uvedení frameworku MEF jako základní architektury pro implementaci rozšíření IDE. Jak česká, tak zahraniční literatura je v současné době na tyto informace poměrně skoupá, proto hlavní zdroj informací pro vývojáře zabývající se rozšířením Visual Studia 2010 o podporu nových jazyků představují zejména internetové stránky MSDN, stránky poskytovatelů frameworků a diskuzní fóra.

Podpora pro jazyk e-PFL analyzovaná, implementovaná a dokumentovaná v této práci ukázala postup při práci na rozšíření Visual Studia. Výsledkem je taková úroveň podpory jazyka e-PFL, která umožňuje běžnou základní práci na projektech založených na jazyce e-PFL – kompletní správa projektů, jejich překlad a spouštění, standardní podpora v editoru.

Implementovaná podpora samozřejmě není vyčerpávající. Díky nově představené technologii MEF může evoluce podpory jazyka e-PFL ve Visual Studiu snadno pokračovat. Další navrhovaná rozšíření by mohla směřovat především do implementace podpory ladění, do rozšíření funkce automatického doplňování výrazů Intellisense o nabídku použitých typů, funkcí a proměnných, nebo do dalších funkcí editoru implementovaných pro potřeby specifik jazyka e-PFL.

6 Literatura

[1] Managed Extensibility Framework. Codeplex MEF Community Site [Online].
URL <http://mef.codeplex.com/>

[2] Apiki, S.: Extending Visual Studio 2010 to Support Additional Programming Languages). MSDN Library [Online].
URL <http://www.devx.com/VS_2010/Article/45058>

[3] Beneš, M.: Funkcionální programování. Učební opora [Online].
URL <http://www.cs.vsb.cz/behalek/education/pp/texty/fp/index.html>

[4] Beneš, M.: *Překladače*. Vysoká škola báňská, Ostrava, 1999, učební opora.

[5] Běhálek, M.: *Vestavný procesně funkcionalní jazyk*. Disertační práce, VŠB-TU Ostrava, 2009.

[6] Granger, C.: VSX103: Lighting up the new Visual Studio 2010 Editor with Rich Extensions. Microsoft video presentation [Online].
URL <http://channel9.msdn.com/Blogs/VSIPMarketing/VSX103-Lighting-up-the-new-Visual-Studio-2010-Editor-with/-Rich-Extensions>

[7] Granger, C.: VSX212: Adding a Language Service into Visual Studio 2010. Microsoft video presentation [Online].
URL <http://channel9.msdn.com/blogs/vsipmarketing/vsx212-adding-a-language-service-into-visual-studio-2010>

[8] Hodges, D.: VS Extensibility Architecture: Intro & Advanced Topics. Microsoft video presentation [Online].
URL <http://channel9.msdn.com/Blogs/AnthonyC/VS-Extensibility-Architecture-Intro--Advanced-Topics>

[9] MacDonald, M.: *Pro WPF in C# 2010: Windows Presentation Foundation in .NET 4.0*. New York, NY, USA: Apress, 2010, ISBN 978-1-4302-7205-2.

[10] Mackey, A.: *Introducing .NET 4.0 With Visual Studio 2010*. New York, NY, USA: Apress, 2010, ISBN 978-1-4302-2455-6.

[11] Microsoft: Creating a Basic Project System. MSDN Library [Online].
URL <http://msdn.microsoft.com/en-us/library/cc512961.aspx>

[12] Microsoft: Extending the Editor. MSDN Library [Online].
URL <http://msdn.microsoft.com/en-us/library/dd885242.aspx>

[13] Microsoft: IronPython Integration. MSDN Code Samples Gallery [Online].
URL <http://code.msdn.microsoft.com/IPyIntegration>

[14] Microsoft: Language Service Overview (Managed Package Framework). MSDN Library [Online].
URL <http://msdn.microsoft.com/en-us/library/bb166360.aspx>

[15] Microsoft: Managed Extensibility Framework Overview. MSDN Library [Online].
URL <http://msdn.microsoft.com/en-us/library/dd460648.aspx>

[16] Microsoft: Managed Package Framework for Projects. Codeplex OpenSource Community Site [Online].
URL <http://mpfproj10.codeplex.com/>

[17] Microsoft: Ook Language Integration. MSDN Code Samples Gallery [Online].
URL <http://code.msdn.microsoft.com/ookLanguage>

[18] Microsoft: Project Types Architecture. MSDN Library [Online].
URL <http://msdn.microsoft.com/en-us/library/bb166186.aspx>

[19] Microsoft: Template Parameters. MSDN Library [Online].
URL <http://msdn.microsoft.com/en-us/library/eehb4faa.aspx>

[20] Microsoft: Visual Studio Development Environment Model. MSDN Library [Online].
URL <http://msdn.microsoft.com/en-us/library/bb165114.aspx>

[21] Novák, I.; Velvárt, A.; Granitz, A.; aj.: *Professional Visual Studio 2010*. Indianopollis, IN, USA: Wiley Publishing, 2010, ISBN 978-0470548653.

[22] Petricek, T.; Skeet, J.: *Real World Functional Programming*. Greenwich, GB: Manning Publications, 2010, ISBN 978-1-933988-92-4.

[23] Randolph, N.; Gardner, D.; Anderson, C.; aj.: *Visual Studio 2010 and .NET 4 Six-in-One*. Indianopollis, IN, USA: Wiley Publishing, 2010, ISBN 978-0-470-49948-1.

[24] Sullivan, B.: VSX205: Visual Studio 2010 Debugger Extensibility. Microsoft video presentation [Online].
URL <http://channel9.msdn.com/Blogs/VSIPMarketing/VSX205-Visual-Studio-2010-Debugger-Extensibility>

A Příloha 1: Gramatika jazyka e-PFL

```
program ⟶ (dataDef | fncDef | importFile)* [mainExpr] EOF

importFile ⟶ IMPORT STRING_VALUE SEMICOLON

dataDef ⟶ DATA UID [(LID)* EQUAL dataCons] SEMICOLON

dataCons ⟶ (UID (dataConParam)* (BAR UID (dataConParam)*)*)+

dataConParam ⟶ UID | LID | LPAR dataConParam RPAR | listType

fncDef ⟶ [PRIMITIVE] (LID | OP) DOUBLECOLON processType
   SEMICOLON | [PRIMITIVE] (LID | OP) (LID)* EQUAL expr
   [WHERE LCPAR fncDef (fncDef)* RCPAR] SEMICOLON

processType ⟶ varType [RARROW processType]

varType ⟶ LID | LID (algType | parType | listType) | algType
   | parType | listType

typeExpr ⟶ varType [RARROW typeExpr]

type ⟶ LID | algType | parType | listType

algType ⟶ UID (type)*

parType ⟶ LPAR typeExpr (COMMA varType)* RPAR

listType ⟶ LSPAR typeExpr RSPAR

expr ⟶ application

application ⟶ expr0 (expr0)* [OP application]

expr0 ⟶ LPAR expr (COMMA expr)* RPAR
    | LID
    | UNIT
    | UID
    | INT_VALUE
    | REAL_VALUE
    | CHAR_VALUE
    | STRING_VALUE
    | CASE expr OF LCPAR (pattern RARROW expr)+ RCPAR
```

```
    | LABEL ...
    | IF expr THEN expr ELSE expr
    | LSPAR listExpr RSPAR

listExpr ⟶ LPAR [expr (COMMA expr)*] RPAR

pattern ⟶ UID (simplePattern)* OP pattern
    | simplePattern OP pattern

simplePattern ⟶ LID
    | UNDERLINE
    | INT_VALUE
    | REAL_VALUE
    | CHAR_VALUE
    | STRING_VALUE
    | UID
    | LPAR pattern (COMMA pattern)* RPAR
    | LSPAR [pattern (COMMA pattern)*] RSPAR

mainExpr ⟶ MAIN EQUAL expr SEMICOLON
```

Vysvětlivky:

`(...)*`	0 nebo více opakování výrazu
`(...)+`	1 nebo více opakování výrazu
`[...]`	0 nebo 1 opakování výrazu

B Příloha 2: Obsah přiloženého CD a postup instalace

Na CD přiloženém k této práci se nacházejí následující složky:

- *DP*
- *Source*
- *Install*

Složka *DP* obsahuje soubor s touto diplomovou prací ve formátu PDF.

Složka *Source* obsahuje zdrojové kódy výsledného řešení podpory pro jazyk e-PFL. Jedná se o řešení (solution) pro Visual Studio 2010. Pro jeho správnou činnost je nutné nainstalovat Visual Studio 2010 SDK.

Složka *Install* obsahuje instalační soubory sestaveného řešení.

Postup instalace

Pro instalaci podpory jazyka e-PFL ve Visual Studiu 2010 je třeba provést následující:

1. Ze složky *Install/ProgramFiles* na přiloženém CD zkopírovat složku *E-PFL* včetně jejího obsahu do složky *ProgramFiles* na cílovém počítači.
2. Spustit soubor *E-PFL_project.vsix* ze složky *Install* z přiloženého CD.
3. Po provedení instalace spustit Visual Studio.
4. Otevřít Extension Manager a přesvědčit se, že rozšíření e-PFL je nainstalováno a povoleno (enabled).