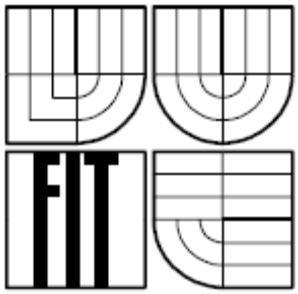

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií

.NET CodeDom

Semestrální práce do předmětu
Vybrané problémy informačních systémů

Rudolf Schönecker Červen 2006

Obsah

1. Úvod

Styl i metodika programování se během posledních dvaceti let měnila radikálním způsobem. V poslední dekádě 20. století se prosadila myšlenka objektového programování a spolu s ní i jazyk C++. Nové modelovací jazyky a techniky se draly do popředí, vznikala zde korelace mezi návrhem, který byl vyjádřen modelovacími jazyky (povětšinou grafickými), a mezi zdrojovým kódem, který psán v jazyce C++ trpěl veškerými neduhy vycházejícími z poděděných archaismů starších procedurálních jazyků.

Z našeho pohledu je důležitá právě vazba mezi zdrojovým kódem na jedné straně a modelem daného informačního systému na straně druhé. Tato vazba o sobě dávala znát zejména při změnách modelu či potřebné restrukturalizaci kódu uvnitř zdrojových souborů. Objevovala se samozřejmě řešení, která se pokoušela různými technikami dané obtíže řešit. Společným rysem těchto řešení bylo použití souboru se zdrojovým kódem jakožto nosného kontejneru základní informace.

Rok 2001 dal vznik nové metodologii vývoje - modelem řízené architektuře (MDA). Základním principem a nositelem informace přestal být kód, stal se jím model. Na model je možno nahlížet abstrakcemi různých úrovní, je-li vyjádřen rigorózně v některém modelovacím jazyce, můžeme jej podrobit transformacím, rozvíjet jej, a až v poslední fázi použít generativních prostředků pro převod modelu na kód příslušné architektury v jistém jazyce. Asociace mezi třídami v podobě šipek grafu UML diagramu a jiné ekvivalenty jinak složitých vazeb v kódu, umožnily transformace, které by, vzhledem k často komplikovaným a rozsáhlým vazbám v kódu, nebylo možno reálně provést.

Vývojové nástroje bývají zrcadlem používané metodologie vývoje a nejinak je tomu u MDA. Potřeba spravovat model v podobě UML diagramů, fragmentů kódů psaných v různých jazycích, dotýkajících se rozličných oblastí informačních systémů, ale přesto popisujících z hlediska sémantiky jedinou věc, vedla ke vzniku Code Document Object Model (CodeDOM) jako sjednocení různých náhledů (v různých jazycích a různými prostředky) na ta samá data (model).

Snahou návrhářů vývojových studií je poskytnout aplikačním programátorům nástroje podporující nejnovější poznatky z oblasti metodologie vývoje aplikací. Skutečný stav podpory MDA ve vývojových studiích, konkrétně VS 2005 a Delphi 2006, se zdá být povzbudivým - samotná vývojová studia jsou již částečně postavena na bázi frameworku, který je společný vývojářům aplikační domény, a ve velké míře využívají možností právě CodeDOM, jakožto nástroje pro spravování objektového modelu dokumentu a zdrojového kódu.

V následujících částech textu se zaměřím na Microsoft .NET iniciativu CodeDOM. Text budu psát z pohledu programátora přehledově znalého architektury platformy .NET a jazyka C#. Představím CodeDOM z hlediska jeho lokace uvnitř .NET Frameworku 2.0, uvedu strukturu CodeDOM, vybrané třídy, rozhraní a metody, jimiž lze s CodeDOM pracovat. Zmíním se o možném použití CodeDOM v praxi, nepodporovaných vlastnostech a z toho plynoucích nedostatcích, dále pak o iniciativách vývojářů a cestách, kterak usnadnit práci s třídami CodeDOM a jejich prvky.

2. Objektový model dokumentu

Objektový model dokumentu představuje způsob popisu HTML či XML dokumentu ve smyslu objektově-orientovaného přístupu. K tomuto modelu následně mohou přistupovat různé prostředky pomocí poskytovaných rozhraní. Laskavý čtenář již jistě mnohokrát využil výhod přístupu JavaScriptu či VBScriptu k objektům modelu HTML dokumentu.

Model dokumentu je zpřístupněn v podobě stromu. Jednotlivé uzly stromu jsou reprezentovány objekty – kontejnery. Tím, že samotný model dokumentu je přístupný jako abstraktní datový typ, určují daná rozhraní veškerou možnou funkcionalitu při přístupu k modelu. Navíc je model nezávislý na platformě a použitém programovacích jazyce.

Definice DOM vychází z W3C specifikace a vztahuje se výhradně na oblast tvorby dynamických webových dokumentů. Myšlenka využít některých zajímavých vlastností původního objektového modelu dokumentu však byla rozvíjena dále, zejména metodologií MDA.

3. CodeDOM

MDA uchopila DOM jakožto nosný koncept. Dokumentem je nyní zdrojový kód v libovolném programovacím jazyce. Omezíme-li se na prostředí Microsoft .NET, jedná se o programovací jazyk podléhající Common Language Specification (CLS).

Code Document Object Model (CodeDOM), je pojem označující objektový model dokumentu zdrojového kódu. CodeDOM pomocí jednotlivých objektů (kontejnerů) odráží kompletní hierarchii jazykových elementů uvnitř zdrojového kódu (tj. jmenných prostorů, tříd, metod, příkazů, výrazů, komentářů, aj.).

Koncept modelu byl vytvořen na základě výběru jisté podmnožiny jazykových elementů některých podporovaných jazyků .NET. CodeDOM tedy není sjednocením všech elementů. Často se setkáme s tím, že chceme-li vyjádřit některé specifické rysy jazyka, nemusí být odpovídající třídy pro model k dispozici. Někdy schází i základní prvky, jako třeba unární operátor. Většinou je ale možné nahradit daný prvek bez ztráty jazykové neutrality.

3.1. Funkční struktura CodeDOM

CodeDOM je logicky a funkčně rozdělen do dvou částí (dvou jmenných prostorů):

1) `System.CodeDOM` – je tvořeno množinou tříd, které můžeme využít při vytváření instancí - elementů CodeDOM grafu, jež bude odrážet strukturu zdrojového kódu (v jistém .NET jazyce). Mapování jazykových konstrukcí na elementy CodeDOM grafu není přímé, CodeDOM graf je na konkrétním programovacím jazyce nezávislý.

2) `System.CodeDom.Compiler` – definuje třídy pro generování zdrojového kódu z CodeDOM grafů a struktur, a prostředky pro kompilaci zdrojového kódu v podporovaných jazycích. .NET Framework přímo podporuje generátory a překladače jazyků C#, Visual Basic a JScript, přičemž podporu dalších jazyků je možno přidat skrze implementaci příslušných rozhraní providerů.

Někdo by mohl namítnout, že dostupné nástroje pro praktické využití CodeDOM nestačí. Je totiž hezké mít k dispozici prostředky pro tvorbu elementů CodeDOM grafu, způsob, jak pro již existující CodeDOM graf vygenerovat kód v libovolném jazyce .NET, či jak daný CodeDOM graf zkompilovat, ale pro automatizované zpracování je rovněž potřeba umět načíst již existující zdrojový kód a transformovat jej do podoby CodeDOM grafu.

Zde přicházejí v úvahu parsery konkrétních .NET jazyků, které, napojeny na wrapery CodeDOM modelu, jsou schopny zdrojový kód v daném jazyce do podoby CodeDOM grafu převést. Tyto parsery nejsou součástí namespace `System.CodeDom`, neboť parser jazyka je specifickou záležitostí, která se řadí k prostředkům konkrétního jazyka. Je s podivem, že nejsou součástí providera příslušného jazyka, kde bychom je očekávali s největší pravděpodobností. Ve Visual Studiu je koncept CodeDOM využit např. při serializaci kódu. Pro některé jazyky, které Microsoft podporuje v Visual Studiu .NET 2005, lze nalézt tyto parsery podle [10] v namespace Microsoft.VisualStudio.Dll.

Dále jsou zde snahy využít již existujících parserů (např. z implementace Mono .NET) a na jejich základě postavit překladače zdrojového kódu do podoby CodeDOM grafu. Příkladem může být CS CodeDOM Parser na [13]. Bohužel, tyto převodníky většinou podporují pouze jistou podmnožinu jazyka, navíc jsou často vystavěny na základě parserů starších verzí .NET jazyků (zde např. C#). Jako nejjednodušší cesta se tedy jeví prozkoumat jmenné prostory programovacích studií Microsoftu - Microsoft.VisualStudio.Dll pro jazyky C#, C++, VB a JScript, či odpovídající namespace firmy Borland v Delphi 2006 pro jazyky Delphi, C# a C++.

Další otázka, která laskavého čtenáře jistě napadla, je: jakým způsobem mapovat specifické jazykové kontrukce v jazycích .NET, které nemají přímý ekvivalent v podobě objektu CodeDOM elementu. Odpovědí jsou tzv. code snippets. Jedná se o prvky, které zapouzdří výraz, příkaz, či obecnou jazykovou konstrukci. Tím, že využijeme code snippets, můžeme nepřeveditelné prvky jazyka přímo umístit do CodeDOM grafu. Zápor této pomůcky se projeví při generování kódu nebo jeho kompilaci. Univerzální nástroje (generátor, překladač) pro práci s CodeDOM grafem nebudou rozumět obsahu code snippet a nebudou schopny s ní přímo pracovat. Code snippets jsou jazykově závislé.

3.2. Namespace System.CodeDOM

Nejprve si představíme namespace `System.CodeDOM`. Kontejnerové třídy v `System.CodeDOM`, které lze využít pro tvorbu elementů CodeDOM grafu, můžeme rozdělit do těchto oblastí:

- **obecné kontejnery grafu**
  (`CodeCompileUnit`, `CodeObject`, `CodeSnippetCompileUnit`)

- **deklarace namespace**
  (`CodeNamespace`*, `CodeNamespaceImport`*)
- **deklarace typů**
  (`CodeTypeDeclaraction`*)
- **parametry typů**
  (`CodeTypeParameter`*)
- **reference typů**
  (`CodeTypeReference`*, `CodeTypeReferenceOptions`)
- **členské prvky typů**
  (`CodeTypeMember`*, `CodeMemberMethod`, `CodeMemberField`, `CodeMemberProperty`, `CodeContructor`, `CodeTypeContructor`, `CodeEntryPoint`, ..)
- **atributy členských prvků**
  (`MemberAttributes`, `CodeAttributeDeclaration`*, `CodeAttributeArgument`*, `CodeParameterDeclaractionExpression`*, `CodeDirectionExression`, `FieldDirection`)
- **příkazy**
  (`CodeStatement`*, `CodeExpressionStatement`, `CodeVariableDeclarationStatement`, `CodeAssignStatement`, `CodeBinaryOperatorExpression`, `CodeComment`, `CodeConditionStatement`, `CodeIterationStatement`, …)
- **výrazy**
  (`CodeExpression`*, `CodeCastExpression`, `CodeArrayIndexerExpression`, `CodeObjectCreateExpression`, `CodeMethodInvokeExpression`, …)
- **snippets literálů**
  (`CodeSnippetCompileUnit`, `CodeSnippetTypeMember`, `CodeSnippetStatement`, `CodeSnippetExpression`)
- **zbývající**
  (`CodeObject`, `CodeLinePragma`, `CodeDirective`*, `CodeRegionDirective`, …)

Nemá smysl zde uvádět deklarace všech tříd, jejich metod a vlastností. Jejich kompletní přehled lze nalézt v [9]. Gramatiku popisující CodeDOM graf v EBNF včetně demonstračního CodeDOM provideru pro jazyk CSharp naleznete na stránkách [6].

3.3. Namespace System.CodeDOM.Compiler

Namespace `System.CodeDOM.Compiler` deklaruje tato tři základní rozhraní:

1) `ICodeCompiler` – rozhraní překladače pro překlad zdrojového kódu nebo CodeDOM grafu. Implementací rozhraní `ICodeCompiler` mají vývojáři možnost přeložit assembly v podobě zdrojového kódu, jeho části, či CodeDOM grafu. Rozhraní navíc umožňuje nastavit volby překladu, získat přístup k postupným výsledkům překladu (chybová a varovná hlášení, aj.), a to prostřednictvím parametrů – objektů – typu `CompilerParameters` a `CompilerResults`. Rozhraní by mělo být implementováno v případě, že žádáme automatický překlad nového jazyka s podporou CodeDOM.

* existuje i Collection dané třídy

Rozhraní deklaruje tyto metody:

- `CompileAssemblyFromDom`† - přeloží assembly v podobě CodeDOM stromu
- `CompileAssemblyFromFile`† - přeloží assembly ze zdrojového souboru
- `CompileAssemblyFromSource`† - přeloží assembly z textového řetězce

poznámka: od verze .NET 2.0 jsou již `ICodeGenerator` a `ICodeCompiler` součástí třídy `CodeDomProvider`, jestliže chceme použít vlastní code provider, stačí překrýt metody uvedené třídy

2) `ICodeGenerator` – rozhraní pro generování kódu z CodeDOM grafu. Implementováním tohoto rozhraní umožníme dynamické generování kódu v příslušném jazyce.

   Mezi důležité metody patří:

   - `GenerateCodeFromCompileUnit` – generuje kód pro specifický element CodeDOM grafu (genericky), výsledek směřuje do objektu typu `TextWriter`
   - `GenerateCodeFromExpression` - generuje kód pro element výrazu
   - `GenerateCodeFromNamespace` - generuje kód pro element namespace
   - `GenerateCodeFromStatement` - generuje kód pro element příkazu
   - `GenerateCodeFromType` - generuje kód pro element deklarace typu
   - `Supports` – vrací hodnotu indikující, jestli generátor podporuje vlastnosti jazyka specifikované v parametru – objektu – typu `GeneratorSupport`

   poznámka: konkrétní objekt implementující ICodeGenerator lze získat voláním metody `CreateGenerator` třídy `CodeDomProvider`.

3) `ICodeParser` – rozhraní pro převod zdrojového kódu na `CodeCompileUnit`. Chceme-li vytvořit např. designery kódu, serializery, atd., implementujeme v rámci tohoto rozhraní metodu

   - `Parse` – převádí vstupní textový stream typu `TextReader` na `CodeCompilerUnit`

Namespace `System.CodeDOM.Compiler` dále deklaruje třídy (pouze výběr):

- `CodeCompiler` – příklad implementovaného rozhraní `ICodeCompiler`
- `CodeDomProvider` – abstraktní třída pro implementaci `CodeDomProvider`
- `CodeGenerator` – abstraktní třída, příklad impl. rozhraní `ICodeGenerator`
- `CodeGeneratorOptions` – volby nastavení pro generátor kódu
- `CodeParser` – prázdná implementace rozhraní ICodeParser
- `CompilerError` – chybová či varovná hlášení
- `CompilerInfo` – nastavení informací o jazyce
- `CompilerParameters` – parametry používané při spuštění překladače
- `CompilerResults` – výsledky překladu

† podporována je i varianta Batch pro dávkovou aplikaci konkrétní metody na pole vstupů

4. Demonstrační příklad

V následující kapitole bude uveden příklady, jak vytvořit konkrétní CodeDOM graf, který bude reprezentovat jednoduchý program, dále jak tímto způsobem vytvořený CodeDOM graf zkompilovat a jak zpětně vygenerovat kód příslušného programu v .NET jazycích C# nebo VB. Příklady kódu budou vycházet z následující programové ukázky:

```
namespace PrikladFaktorial
{
    class FaktorialClass
    {
        static long Faktorial(long num)
        {
            long result = 1;
            while (num > 1)
            {
                result *= num;
                num--;
            }
            return result;
        }

        static void Main(string[] args)
        {
            Console.WriteLine("Faktorial 6 je {0}.", Faktorial(6));
        }
    }
}
```

Při vytváření příkladu jsem vycházel z příkladů ve výborně zpracovaných článcích [4] a [8] a programových ukázek k jednotlivým elementům v [9]. Příklady jsou odzkoušeny na Microsoft .NET 2.0 ve Visual Studiu 2005 (licence VUT Brno).

4.1. Vytvoření CodeDOM grafu

Nejdříve vytvoříme namespace s názvem MyNamespace a vložíme do něj prvky reprezentující import namespace `System`:

```
CodeNamespace MyNameSpace = new CodeNamespace("PrikladFaktorial");
MyNameSpace.Imports.Add(new CodeNamespaceImport("System"));
```

Nyní vytvoříme třídu s názvem Myclass, která bude mít viditelnost public. Využít můžeme třídu CodeTypeDeclaration, která slouží obecně pro reprezentaci tříd, struktur, rozhraní nebo výčtů. Tu vložíme jako nový typ do MyNameSpace.

```
CodeTypeDeclaration MyClass = new CodeTypeDeclaration("FaktorialClass");
MyClass.IsClass = true;
MyClass.Attributes = MemberAttributes.Public;
MyNameSpace.Types.Add(MyClass);
```

Vytvoříme metodu Faktorial a nastavíme jí atributy public a static, vytvoříme a přidáme parametr num typu long.

```
CodeMemberMethod MyMethod = new CodeMemberMethod();
MyMethod.Attributes = MemberAttributes.Public | MemberAttributes.Static;
MyMethod.Name = "Faktorial";
MyMethod.ReturnType = new CodeTypeReference("System.Int64");
MyClass.Members.Add(MyMethod);

CodeParameterDeclarationExpression MyNum =
      new CodeParameterDeclarationExpression("System.Int64", "num");
MyNum.Direction = FieldDirection.In;
MyMethod.Parameters.Add(MyNum);
```

Několikrát budeme potřebovat konstantní celočíselný výraz 1, pojmenujeme si ho jako Const_1.

```
CodeExpression Const_1 = new CodePrimitiveExpression(1);
```

Přidáme příkaz pro deklaraci lokální proměnné result typu long a její inicializaci hodnotou 1. Dle dokumentace využijeme property InitExpression.

```
CodeVariableDeclarationStatement MyDeclStm =
      new CodeVariableDeclarationStatement("System.Int64", "result");
MyDeclStm.InitExpression = Const_1;
MyMethod.Statements.Add(MyDeclStm);
```

Nyní si definujeme pomocné odkazy na existující proměnné result a num, které budeme dále využívat.

```
CodeVariableReferenceExpression ref_result =
      new CodeVariableReferenceExpression("result");
CodeArgumentReferenceExpression ref_num =
      new CodeArgumentReferenceExpression("num");
```

Cyklus vytvoříme pomocí elementu CodeIterationStatement. CodeDOM má pouze jednu vnitřní formu cyklu podobající se for-cyklu známému např. z jazyka C. Jak uvidíme později při generování, jednotlivé generátory pro reprezentaci obecné formy volí různé typy cyklů. Praktická zkušenost jistě velí zeptat se, jak specifikovat častou situaci, kdy není zadána některá část cyklu (inicializace/podmínka/inkrementace). [9] doporučuje vložit Code Snippet s prázdným řetězcem – obsahem.

```
CodeIterationStatement MyLoop = new CodeIterationStatement(
     new CodeSnippetStatement(""),
     new CodeBinaryOperatorExpression(
            ref_num,
            CodeBinaryOperatorType.GreaterThan,
            Const_1),
     new CodeAssignStatement(
            ref_num,
            new CodeBinaryOperatorExpression(ref_num,
            CodeBinaryOperatorType.Subtract,
            Const_1));,
     new CodeStatement[] {
            new CodeAssignStatement(
                   ref_result,
                   new CodeBinaryOperatorExpression(
                          ref_result,
                          CodeBinaryOperatorType.Multiply,
```

```
                              ref_num)) });
MyMethod.Statements.Add(MyLoop);
```

Posledním příkazem funkce Faktorial je návrat výsledné hodnoty.

```
CodeMethodReturnStatement ReturnStm =
      new CodeMethodReturnStatement(ref_result);
MyMethod.Statements.Add(ReturnStm);
```

Nyní přidáme metodu Main. Jedná se o vstupní metodu celého programu, použijeme proto element CodeEntryPointMethod, který je jinak plně ekvivalentní původní CodeMemberMethod.

```
CodeEntryPointMethod MyMainMethod = new CodeEntryPointMethod();
MyClass.Members.Add(MyMainMethod);
MyMainMethod.Attributes = MemberAttributes.Public |MemberAttributes.Static;
```

Nakonec přidáme příkaz výpisu výsledku volání funkce Faktorial v těle funkce Main.

```
CodeMethodInvokeExpression methodInvoke = new CodeMethodInvokeExpression(
     new CodeTypeReferenceExpression("Console"),
     "WriteLine",
     new CodeExpression[] {
           new CodePrimitiveExpression("Faktorial 6 je {0}."),
           new CodeMethodInvokeExpression(
                 new CodeMethodReferenceExpression(
                       new CodeTypeReferenceExpression("FaktorialClass"),
                       "Faktorial"),
                 new CodeExpression[] {
                       new CodePrimitiveExpression(6)} ) } );
MyMainMethod.Statements.Add(methodInvoke);
```

4.2. Generování kódu

Následuje kód pro vygenerování obsahu CodeDOM grafu z MyNameSpace. Tento příklad demonstruje některé možnosti nastavení objektu typu CodeGeneratorOptions. Použity jsou vlastnosti BracingStyle udávající C podobu odsazování bloků na novém řádku, či IdentString, která vyjadřuje míru odsazení bloku. K dispozici máme dále např. BlankLinesBetweenMembers nebo ElseOnClosing.

```
StringBuilder MyCode = new StringBuilder();
StringWriter sw = new StringWriter(MyCode);

CodeGeneratorOptions cop = new CodeGeneratorOptions();
cop.BracingStyle = "C";
cop.IndentString = "  ";

VBCodeProvider provider = new VBCodeProvider();
provider.GenerateCodeFromNamespace(MyNameSpace, sw, cop);
```

Uvedený příklad generuje kód do sw, který je poté přístupný prostřednictvím volání MyCode.ToString():

```
Imports System
```

```
Namespace PrikladFaktorial

  Public Class FaktorialClass

    Public Shared Function Faktorial(ByVal num As Long) As Long
      Dim result As Long
      result = 1

      Do While (num > 1)
        result = (result * num)
        num = (num - 1)
      Loop
      Return result
    End Function

    Public Shared Sub Main()
      Console.WriteLine("Faktorial 6 je {0}.", FaktorialClass.Faktorial(6))
    End Sub
  End Class
End Namespace
```

V předchozím příkladě je použit VBCodeProvider. Kdybychom jej zaměnili za CSharpCodeProvider, získali bychom následující kód. Všimněme si zejména rozdílného přístupu providerů při generování cyklu:

```
namespace PrikladFaktorial
{
  using System;

  public class FaktorialClass
  {

    public static long Faktorial(long num)
    {
      long result;
      result = 1;
      for (
      ; (num > 1); num = (num - 1))
      {
        result = (result * num);
      }
      return result;
    }

    public static void Main()
    {
     Console.WriteLine("Faktorial 6 je {0}.", FaktorialClass.Faktorial(6));
    }
  }
}
```

4.3. Kompilace CodeDOM grafu

Zde je ukázka kompilace CodeDOM grafu vygenerovaného pro příklad faktoriálu. Nejprve jsou zadány volby kompilace prostřednictvím parametru typu CompilerParameters, poté je vytvořen prvek typu CodeCompileUnit, do kterého je vloženo namespace

vygenerovaného CodeDOM grafu, a který lze podrobit kompilaci pomocí metody CompileAssemblyFromDom příslušného providera konkrétního .NET jazyka.

```
bool includeDebugInformation = true;
string OutputFileName = "faktorial.exe";

CompilerParameters options =
      new CompilerParameters(
            new string[] { "System.dll"  },
            OutputFileName,
            includeDebugInformation);
options.GenerateExecutable = true;

CodeCompileUnit compunit = new CodeCompileUnit();
compunit.Namespaces.Add(MyNameSpace);

CompilerResults cr = provider.CompileAssemblyFromDom(
                                    options,
                                    new CodeCompileUnit[] { compunit });
```

5. Wrappers pro CodeDOM

Tvůrci CodeDOM při návrhu CodeDOM modelu usilovali především o univerzálnost a možnost snadného propojení kódu pro generování jednotlivých elementů, přičemž stránka lidského zápisu, či kontrola kontextu, do kterého je daný element vsazen, byla již podružná. Mnohdy je možné pomocí legálních prostředků vygenerovat chybný model. Dokumentace navíc tuto skutečnost nijak nepostihuje a vše je tedy odvislé od znalostí a zkušeností vývojáře, experimenty s modelem.

Zde se otvírá prostor pro nejrůznější „nástavby“ nad CodeDOM, pro tzv. wrapery, které si kladou za cíl především zpříjemnit ruční zápis výstavby CodeDOM grafu z jednotlivých elementů a omezit kontext, ve kterém lze jednotlivé elementy použít tak, aby byl za každých okolností vygenerován legální CodeDOM model.

5.1. MbUnit - Refly

V této kapitole uvedu Refly jako jednoho ze zástupců klasických wrapperů nad CodeDOM. Refly, součást projektu MbUnit. Je dostupný na [3] a to jak v podobě zkompilované assembly, tak v podobě zdrojových kódů.

Cílem Refly je zjednodušit generování kódu pomocí CodeDOM a to prostřednictvím pojmenovaných členů, prvků a vlastností a jednoduššího vytváření jmenných prostorů, tříd, metod a ostatních prvků. Refly rovněž nabízí prvky, které v CodeDOM přímo vyjádřeny nejsou (např. while, či foreach cyklus, ThrowIfNull, atd.).

Původní ukázka kódu generujícího příklad Faktoriál by zapsaná pomocí Refly mohla vypadat následovně:

```
NamespaceDeclaration ns = new NamespaceDeclaration("PrikladFaktorial");
ns.Imports.Add("System");
ClassDeclaration fclass =
```

```
     ns.AddClass("FaktorialClass",System.Reflection.TypeAttributes.Public);

// metoda faktoriál
MethodDeclaration faktorial = fclass.AddMethod("Faktorial");
faktorial.Signature.ReturnType = new TypeTypeDeclaration(typeof(long));
faktorial.Attributes = MemberAttributes.Public | MemberAttributes.Static;
ParameterDeclaration num =
     faktorial.Signature.Parameters.Add(typeof(long), "num");

// tělo metody faktorial
VariableDeclarationStatement result =
     Stm.Var(typeof(long), "result", Expr.Prim(1));
faktorial.Body.Add(result);
IterationStatement whilestm =
     Stm.While(new BinaryOpOperatorExpression(
                         Expr.Arg(num),
                         Expr.Prim(1),
                         CodeBinaryOperatorType.GreaterThan));
whilestm.Statements.AddAssign(Expr.Var(result), new
BinaryOpOperatorExpression(Expr.Var(result), Expr.Arg(num),
CodeBinaryOperatorType.Multiply));
whilestm.Statements.AddAssign(Expr.Arg(num),
      new BinaryOpOperatorExpression(
            Expr.Arg(num), Expr.Prim(1), CodeBinaryOperatorType.Subtract));
faktorial.Body.Add(whilestm);
faktorial.Body.Return(Expr.Var(result));

// metoda Main
MethodDeclaration main = fclass.AddMethod("Main");
main.Body.Add(Expr.Type(typeof(Console)).Method("WriteLine").Invoke(
      new Expression[] {
             Expr.Prim("Faktorial 6 je {0}."),
             Expr.Type(fclass).Method(faktorial).Invoke(Expr.Prim(6)) }
      )
);
```

Seznamovacím informačním zdrojem k Refly je [3], kde je na velice jednoduchém příkladě demonstrována ekvivalence generujícího kódu zapsaného v CodeDOM a v Refly. Laskavý čtenář může dále využít vygenerované dokumentace ze zdrojových kódů či zdrojové kódy samotné. Ty doporučuji zejména při zájmu o poznávání samotného CodeDOM a korektního přístupu k jednotlivým jeho třídám a elementům.

6. Závěr

Současný vývoj rozsáhlých informačních systémů je založen na podpůrných nástrojích, utilitách a vývojových prostředích, které vývoj usnadní, pomohou vývojářům spravovat model daného informačního systému a souběžně jeho zdrojový kód. Většina těchto nástrojů využívá specifik a výhod serializace a generování kódu. Tento příspěvek si kladl za cíl seznámit laskavého čtenáře s alternativou modelu dokumentu obsahujícího zdrojový kód v prostředí .NET, CodeDOM. Chtěl ukázat, jakým způsobem lze daný model používat, a co vše tento model a s ním spojené části frameworku umožňují.

Originální dokumentací k .NET CodeDOM je pouze automaticky vygenerovaný dokument v MSDN, doplněný o několik poznámek, ukázek kódu. Situaci naštěstí zachraňují články a poznámky bloggerů. Příspěvek jsem pojal jako úvodní materiál, který by měl zájemci o studium CodeDOM modelu poskytnout úvodní informace a nasměrovat jej na existující zdroje při dalším poznávání. Příslušné odkazy na články a ostatní materiály jsou uvedeny na konci tohoto příspěvku.

Reference

[1] Bonansea, G.: Compiling with CodeDom
http://www.codeproject.com/dotnet/CompilingWithCodeDom.asp

[2] Dunn, C.: The CodeDOM and the Delphi IDE
http://blogs.borland.com/corbindunn/archive/2004/09/30/1493.aspx

[3] Halleux, J.: Refly, makes the CodeDom'er life easier
http://www.codeproject.com/csharp/refly.asp
http://www.mertner.com/confluence/display/MbUnit/Home

[4] Harrison, N.: Using the CodeDOM
http://www.ondotnet.com/pub/a/dotnet/2003/02/03/codedom.html

[5] Jones, A. R.: Generate .NET Code in Any Language Using CodeDOM
http://www.devx.com/dotnet/Article/15678

[6] Kahu Research: CodeDOM Grammar & Samle CodeDOM Provider
http://kahu.zoot.net.nz/codedom/

[7] Kloeten, O.: Commonly Used .NET Coding Patterns in CodeDom
http://www.codeproject.com/csharp/CodeDomPatterns.asp

[8] Korzeniowski, B. J.: Microsoft .NET CodeDom Technology
http://www.15seconds.com/issue/020917.htm

[9] Microsoft: MSDN
http://www.microsoft.com/msdn

[10] Mojica, J.: CodeDOM
http://www.oreillynet.com/windows/blog/2004/10/codedom.html

[11] Novotný, O.: Next Generation Tools for Object-Oriented Development
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnmaj/html/nexgen.asp

[12] Steinert, A. J.: Bring the Power of Templates to Your .NET Applications with the CodeDOM Namespace
http://msdn.microsoft.com/msdnmag/issues/03/02/CodeDOM/

[13] Zderadicka, I.: CS CodeDOM Parser
http://www.sweb.cz/ivan.zderadicka/csparser.html

Odkazy na články a příspěvky na internetu byly platné v červnu 2006.