Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Bakalářská práce

Využívání .NET assembly z neřízeného C++

Plzeň, 2015 Vojtěch Kinkor

Prohlášení

Prohlašuji, že jsem bakalářskou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 19. 6. 2015

Vojtěch Kinkor

Abstract

Using .NET Assemblies in Unmanaged C++

The aim of this thesis is to investigate options of using .NET assemblies from applications programmed in unmanaged C++ and propose an optimal method for their interconnection. This can be helpful for extending already existing C++ applications which cannot be rewritten for .NET platform. In addition part of this thesis presents implemented utility for automatic generation of C++ wrappers for .NET classes and so in conclusion to simplify this interconnection. Presented solution offers up to 5× better performance than competitive tools.

Abstrakt

Využívání .NET assembly z neřízeného C++

Cílem této práce je prozkoumat možnosti využívání .NET assemblies z aplikací napsaných v neřízeném C++ a navrhnout optimální metodu jejich propojení. Toto propojení může usnadnit rozšiřování stávajících C++ aplikací, které není možné přepsat pro platformu .NET. Součástí práce je návrh nástroje pro automatické generování C++ wrapperů pro .NET třídy a tím pádem usnadnění tohoto propojení. Výsledné řešení nabízí až 5× vyšší výkon než stávající nástroje.

Obsah

1 Úvod

.NET je moderní softwarová platforma, která slouží pro vývoj a běh aplikací. Vývojářům nabízí širokou knihovnu funkcí pokrývající mnoho oblastí – například grafická uživatelská rozhraní, řízení přístupu k datům, přístup k databázím, síťovou komunikaci, aj. [1, 2] a díky tomu zjednodušuje a zrychluje vývoj. Není proto divu, že .NET je v aktuální době velmi oblíbenou a prosazovanou možností pro tvorbu aplikací na platformě Microsoft Windows. Zároveň však existuje velké množství softwaru, který je vyvíjen v jazyce C/C++ a je dlouhodobě v provozu. Může se jednat jak o různé interní systémy, tak o běžné uživatelské programy. Přepsání celého programu pro .NET je často nerealizovatelné nebo může být velmi nákladné.

Cílem této práce je prozkoumat oblast propojení programů napsaných v jazyce C++ s komponentami vytvořenými nad platformou .NET a srovnat stávající možnosti. Zaměří se především na technologii COM a hostování jádra platformy .NET a bude je porovnávat s dalšími nalezenými metodami například z hlediska multiplatformnosti. Na základě tohoto srovnání bude vybrána vhodná metoda a vytvořen nástroj usnadňující transparentní využívání funkcionality .NET assembly. Uživateli má přinést možnost automaticky generovat tzv. C++ wrappery pro jednotlivé .NET třídy.

V následující kapitole bude nastíněno pozadí dané problematiky, tedy zejména informace o platformě .NET a nativních aplikacích. Kapitola 3 se věnuje jednotlivým metodám propojení, popisuje nutné náležitosti propojení a pomocí ukázek ukazuje rozdíly mezi metodami. V kapitole 4 se věnuji různým úskalím a výběru vhodné metody pro následné použití. Kapitola 5 rozsáhleji popisuje vybranou metodu a návrh její implementace. V kapitolách 6 a 7 se zabývám vytvořeným nástrojem, respektive jeho testováním a porovnáním s jinými metodami z různých hledisek.

2 Základní informace

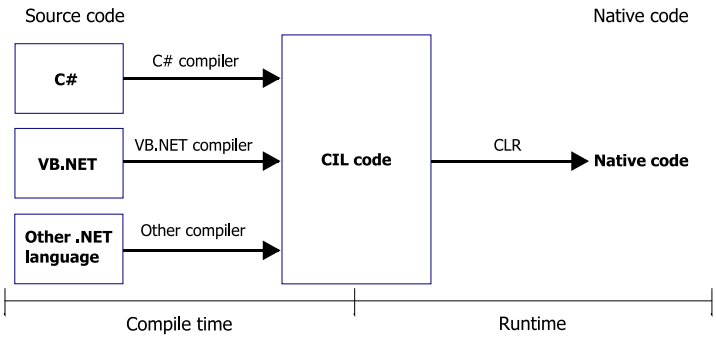
2.1 Platforma .NET

Platforma .NET je vyvíjená mnoha firmami[1] a její součásti byly standardizovány společnostmi ECMA a později i ISO. Jejím hlavním představitelem a první implementací je Microsoft .NET Framework určený pro osobní počítače se systémem Microsoft Windows [1].

Programy pro .NET mohou být napsány v různých programovacích jazycích, které podporují překlad do tzv. CIL kódu – *Common Intermediate Language*. Nejvíce prosazovaným jazykem je C#, který vznikal zároveň s platformou .NET. Vedle něj však existuje množství dalších, spravovaných firmou Microsoft nebo i vyvíjených komunitou.

Common Language Infrastructure, CLI je specifikace kódu aplikací a prostředí pro jejich běh. CIL kód je procesorově a platformně nezávislý kód vytvořený právě dle CLI. Programy se typicky spouštějí v tzv. CLR prostředí – *Common Language Runtime,* které je implementací prostředí podle CLI a je součástí .NET Frameworku. Takto spouštěný kód je dle zavedené terminologie označován jako managed – **řízený kód**. Prostředí poskytuje kódu různé služby a například se stará o alokování a uvolňování paměti [2]. Celý proces je znázorněn v obrázku 1.

Sestava jednoho či více CIL kódu (spolu s *manifestem* – metadaty, která danou sestavu popisují) se označuje jako **.NET assembly** [3].



Obrázek 1 – Překlad aplikace ze zdrojového do nativního kódu. (zdroj: [4])

[1] Mj. Microsoft, Hewlett-Packard, Intel, IBM, Novell, Sun Microsystems

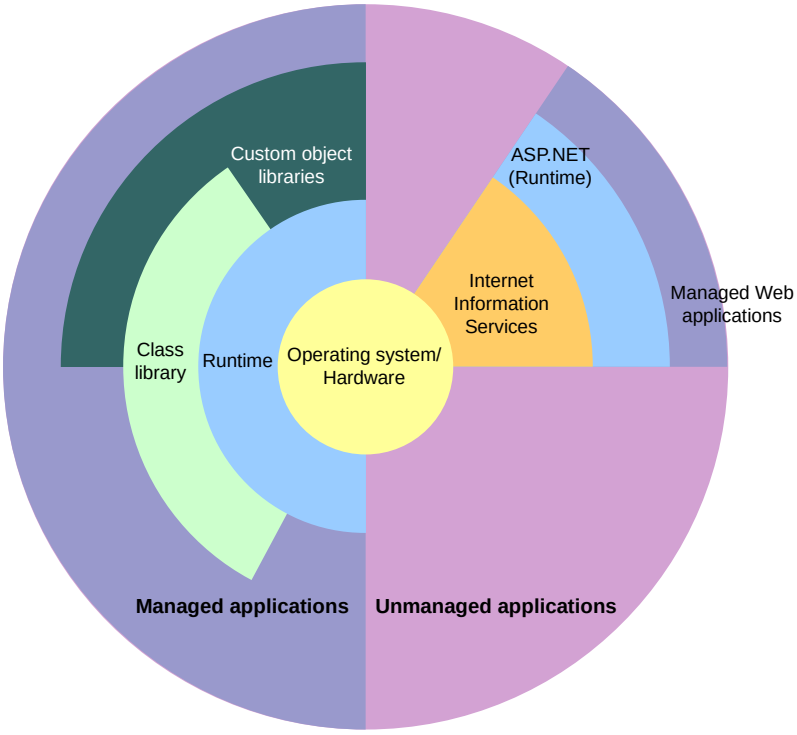
2.2 Neřízený kód

Pokud naopak nevyužijeme možností platformy .NET a vytvoříme program, který se během komplikace přeloží na nativní strojový kód pro jednu konkrétní platformu, mluvíme o tzv. unmanaged code – **neřízeném kódu**. Může se jednat například o aplikace napsané v jazyce C++ využívající Windows API. Programátor si musí sám zajistit správu paměti, správný přístup ke službám apod. Výhodou je větší kontrola nad aplikací a při správném použití může dosáhnout lepšího výkonu.

2.3 Propojení neřízeného a řízeného kódu

Přestože je platforma .NET v současné době velmi oblíbená, stále existuje velké množství softwaru vyvíjeného v C++, u kterého není přepsání do jiného jazyka reálné. Pokud je potřeba takový software rozšířit, je možné pokusit se jej propojit s komponentami vytvořenými jako .NET assembly. Je tedy i možné si takto zpřístupnit velké množství již hotových funkčních knihoven, které pro .NET existují. Právě tomuto propojení se tato práce věnuje.

Na obrázku 2 je znázorněn kontext neřízených a řízených aplikací (včetně různých součástí).



Obrázek 2 – kontext řízených a neřízených aplikací. (zdroj: [5])

2.4 Platforma Mono

Mono[2] je open-source projekt s cílem vytvořit multiplatformní implementaci .NET Frameworku na základě ECMA standardů pro CLI prostředí (pojmenované *Mono Runtime*) a C# kompilátor. Je dostupný pro různé platformy a operační systémy, mimo jiné Linux, Unix, Mac OS X i Microsoft Windows [6].

Aktuální verze (4.10.0 k červnu 2015) implementuje většinu funkcionality z .NET Frameworků ve verzích 1.1 až 4.5, s výjimkami jako např. WPF (moderní GUI subsystém) nebo částmi, které jsou platformně závislé (správa OS,…) [7]. V současnosti také dochází k „adopci" některých částí .NET Frameworku, které byly zveřejněny jako open-source.

Kromě implementace *.NET Framework Class Library* (knihovna funkcí nabízených .NET Frameworkem) zahrnuje projekt též vývoj doplňujících knihoven, mimo jiné GTK# pro vývoj GUI v GTK pro OS Linux/Unix. Na platformě Mono jsou založeny např. komerční nástroje Xamarin[3] (vývoj zejména mobilních aplikací) a Unity[4] (herní engine) [8].

Mono je binárně kompatibilní s .NET Frameworkem, z čehož plyne možnost spouštět programy zkompilované pomocí Microsoft i Mono C# kompilátoru na obou platformách [6].

[2] Viz `http://www.mono-project.com/`
[3] Viz `http://xamarin.com/`
[4] Viz `http://unity3d.com/`

3 Možnosti propojení .NET assembly s neřízenou C++ aplikací

Propojení .NET assembly a C++ lze obecně chápat dvojím způsobem:

1. **Používání řízených .NET assembly z neřízené C++ aplikace.**
2. Používání neřízených C++ knihoven z řízené .NET aplikace.

Tato kapitola, potažmo celá práce, se zabývá pouze prvním způsobem. Možnosti tohoto propojení lze rozdělit do 4 kategorií, lišících se přístupem a požadavky [9]:

1. C++ Interop, označovaná někdy zkratkou IJW (*It Just Works*).
2. Platform Invoke (zkráceně P/Invoke).
3. COM API – standardizované rozhraní pro přístup ke sdíleným objektům.
4. Hostování CLR.

První dvě možnosti Microsoft označuje jako přístup pomocí *Flat API* [10]. Obecně jde o princip zveřejnění metod/funkcí v knihovnách psaných v jazycích C nebo C++ a jejich následné přímé volání bez dalších mezivrstev. Příkladem je Windows API (například knihovna `kernel32.dll`).

V příkladech budu vycházet z následující třídy [11] napsané v C# a zkompilované jako .NET assembly `YahooAPI.dll`:

```
1  // třída pracující s informacemi o akciích pomocí veřejného Yahoo! Finance API
2  public class YahooAPI {
3      public double GetBid(string symbol) {...} // vrátí cenu poptávky akcií
4      public double GetAsk(string symbol) {...} // vrátí cenu nabídky akcií
5      ...
6      public static string info() {...} // vrátí text "YahooAPI library"
7  }
```

Jednoduchá .NET assembly používaná v dalších ukázkách.

3.1 C++ Interop / IJW

Jedním z podporovaných jazyků platformy .NET je C++/CLI, jazyk vycházející z C++. Oproti modernějšímu jazyku C# ztrácí na přehlednosti a jednoduchosti. Výhodou je ale možnost kompilovat kód v tzv. „mixed mode“ režimu. V něm může assembly obsahovat jak řízený, tak neřízený kód. Volání mezi nimi je z hlediska programového kódu „přímé“ (např. vytvoření řízeného objektu uvnitř neřízené funkce). Tato technika se označuje jako C++ Interop, případě zkratkou IJW – *It Just Works* [10].

Této možnosti lze využít, pokud je klientská aplikace napsána v C++ a můžeme do ní nadále zasahovat. Použitím Microsoft Visual C++ kompilátoru s volbou `/clr` se vytvoří aplikace jako tzv. *mixed mode obraz*. Výsledkem je aplikace, která právě může obsahovat řízený i neřízený kód. V takové aplikaci je poté možné přímo volat komponenty z připojených .NET assemblies (viz ukázka 1) [12].

Toto řešení je výhodné, pokud klientskou aplikaci lze překompilovat v tomto režimu. Volání řízeného kódu je intuitivní, bez složitých konstrukcí a má nízkou režii. Jedná se o doporučovanou možnost i z hlediska výkonu a bezpečnosti [10].

```
1  #using "YahooAPI.dll" // import řízené assembly
2  ...
3  int main() {
4      msclr::auto_gcroot<YahooAPI^> api = gcnew YahooAPI();
5      cout << msclr::interop::marshal_as<std::string>(YahooAPI::info()) << endl;
6      cout << "Ask: " << api->GetAsk(gcnew System::String("MSFT")) << endl;
7      cout << "Bid: " << api->GetBid(gcnew System::String("MSFT")) << endl;
8      ...
```

Ukázka 1 – Použití C++ Interop / IJW a jazyka C++/CLI.

3.2 Platform Invoke

Platform Invoke je obecný postup pro volání metod/funkcí z DLL knihoven, které v sobě obsahují informace o poskytovaném API. Jedná se o standardní a velmi často využívaný mechanismus [10].

3.2.1 Vytvoření C++/CLI .NET assembly obsahující unmanaged API

Tato možnost vychází z možnosti kompilace v „mixed mode“ režimu, ale přistupuje k problému ze strany knihovny. Podmínkou je .NET assembly napsaná v jazyce C++/CLI s možností upravovat její kód.

Cílem je exportovat z knihovny funkce, které chceme mít přístupné v klientské

aplikaci. Toho lze dosáhnout dvěma způsoby [13]. První z nich spočívá v přidání označení `dllexport` do původní řízené knihovny ke všem třídám nebo funkcím, které chceme exportovat. Nelze však zveřejnit funkce, které používají .NET konstrukce v deklaraci (například parametry a návratové hodnoty metod nebo třídní proměnné). V takovém případě je nutné použít druhý způsob a vytvořit obalující třídy a funkce, které budou sloužit jako přemostění[5]. V obou případech je pro použití v klientské C++ aplikaci nutné deklarovat importované funkce (resp. třídy) s označením `dllimport` (viz ukázky 2 a 3).

```
#ifdef _SERVER // při kompilaci knihovny
  #define _LNK __declspec(dllexport)
#else // při kompilaci klientské aplikace
  #define _LNK __declspec(dllimport)
#endif

class _LNK YahooAPI {
    public:
        double GetBid(const char* symbol);
        double GetAsk(const char* symbol);
        ...
```

Ukázka 2 – Společný hlavičkový soubor `YahooAPI.h` pro knihovnu i klientskou aplikaci.

```
#pragma comment(lib, "YahooAPI")
#include "YahooAPI.h"
...
int main() {
    YahooAPI *api = new YahooAPI();
    cout << YahooAPI::test() << endl << endl;
    cout << "Ask: " << api->GetAsk("MSFT") << endl;
    cout << "Bid: " << api->GetBid("MSFT") << endl;
    ...
```

Ukázka 3 – Použití knihovny pomocí P/Invoke v klientské aplikaci.

Hlavní nevýhodou tohoto řešení je omezení se na jazyk knihovny C++/CLI a nutnost její úpravy. Výhodou je celistvost výsledné knihovny, která může nabízet zároveň řízené i neřízené API rozhraní.

Samotný CIL kód umožňuje exportování řízených funkcí a jejich volání z neřízeného kódu; toho se i využívá v C++/CLI. Bohužel ostatní .NET jazyky, tedy např. C#, samy o sobě žádnou podobnou možnost nenabízejí. Je však možné vytvořit knihovnu a následně upravit dekompilovaný CIL kód. Postup byl několikrát ověřen a existují i hotové nástroje umožňující zautomatizování celého procesu [14, 15, 16]. Nejedná se však o oficiálně popsaný a podporovaný postup.

[5] Tzn. např. vytvořit a exportovat zcela novou třídu pouze obalující původní funkčnost.

3.2.2 Vytvoření C++/CLI mostu k .NET assembly

Jedná se o speciální případ předchozí možnosti. Spočívá ve vytvoření DLL knihovny, která bude sloužit pouze jako most či obalující mezivrstva (*wrapper*) mezi neřízenou C++ aplikací a řízenou .NET assembly. Musí být vytvořena v jazyce C++/CLI a může přistupovat k libovolným .NET assemblies. Rozsah rozhraní, které bude exportované, závisí na programátorovi.

Podoba mostu je naznačena v ukázkách 4 a 5. Použití v klientské aplikaci je následně obdobné jako v ukázce 3 (pouze budeme přistupovat k třídě `YahooAPIWrapper` namísto `YahooAPI`).

```
1  #ifdef _SERVER // při kompilaci wrapperu
2    #define _LNK __declspec(dllexport)
3  #else // při kompilaci klientské aplikace
4    #define _LNK __declspec(dllimport)
5  #endif
6
7  class YahooAPIManagedWrapper; // třída obsahující odkaz na řízenou knihovnu
8
9  class _LNK YahooAPIWrapper {
10     private:
11         YahooAPIManagedWrapper* _private;
12     public:
13         double GetBid(const char* symbol);
14         double GetAsk(const char* symbol);
15         ...
```

Ukázka 4 – Společný hlavičkový soubor `YahooAPIWrapper.h` pro most (bridge) a klientskou aplikaci.

```
1  #define _SERVER
2  #using "YahooAPI.dll"
3  #include "YahooAPIWrapper.h"
4  ...
5  YahooAPIWrapper::YahooAPIWrapper() {
6      _private = new YahooAPIManagedWrapper();
7      _private->api = gcnew YahooAPI(); // instance řízené knihovny
8  }
9  double YahooAPIWrapper::GetBid(const char* symbol) {
10     return _private->api->GetBid(gcnew System::String(symbol));
11 }
12 ...
```

Ukázka 5 – Soubor `YahooAPIWrapper.cpp` definující most ke knihovně `YahooAPI`.

Jedná se o poměrně univerzální řešení, při kterém není potřeba modifikovat .NET assembly a v určitých situacích ani klientskou aplikaci (případ různých aplikací založených na dynamickém načítání pluginů). Drobnou nevýhodou je nutnost distribuovat s aplikací další knihovnu.

3.3 Přístup pomocí COM API

Component Object Model je technologie vytvořená firmou Microsoft umožňující vytvářet a přistupovat k různým komponentám pomocí binárního rozhraní. Technologie je definována obecně a nezávisle na programovacím jazyku. COM objekty spolu komunikují pomocí pevně definovaného rozhraní, k samotnému objektu nemají přístup. Všechny COM objekty jsou zaneseny v registrech a spravovány operačním systémem [17].

Tato technologie se používá pro zajištění meziprocesové komunikace a propojení softwarových komponent. Přestože byla vyvinuta již na počátku 90. let, je v systémech Microsoft Windows stále velmi používána a díky vývoji disponuje nyní vysokou výkonností.

.NET Framework obsahuje technologii nazvanou **COM Interop** umožňující vzájemnou komunikaci mezi COM a .NET objekty. Snaží se o automatickou konverzi mezi datovými typy obou technologií a správný překlad návratových hodnot a výjimek [10].

Součástí .NET Frameworku je také nástroj `RegAsm.exe`, který umožňuje zaregistrování .NET assembly do systému jako COM objekt a vytvoření tzv. *type library* – binární knihovny popisující obsah COM objektů. Zaregistrovaný objekt je ve skutečnosti CLR prostředí volající kód z příslušné .NET assembly. Type library je následně možné importovat do C++ programu a transparentně volat pomocí názvů deklarovaných v .NET assembly. Standardně jsou zveřejněna pouze programová rozhraní (interface), pro přímé zveřejnění tříd je nutné označit je pomocí speciálních metadat (viz ukázka 6 a použití v ukázce 7) [18, 19].

Částečnou nevýhodou COM API je abstrakce tříd během komunikace – k objektům je nutno přistupovat pomocí rozhraní a z toho vyplývají různá omezení (např. nemožnost volat statické metody nebo změna názvů přetěžových metod). Některé problémy, jako například převod datových typů (*marshalling*), částečně odstraňuje právě COM Interop, avšak za cenu vyšší režie. Výsledkem je však univerzální řešení, které je pro programátora dostatečně transparentní. Další podstatnou nevýhodou je nutnost udržovat v systému záznamy o COM objektech, čímž se omezuje přenositelnost aplikace. Řešením může být technologie *Registration-Free COM*, která umožňuje přístup k objektům bez registrace do systému [20].

```
[ComVisible(true), ClassInterface(ClassInterfaceType.AutoDual)]
public class YahooAPI {
    ...
```

Ukázka 6 – Označení třídy v knihovně `YahooAPI` pro použití pomocí COM API.

```
#import <mscorlib.tlb> raw_interfaces_only
#import "YahooAPI.tlb"
...
int main() {
    CoInitialize(0); // nutná inicializace COM API

    YahooAPI::_YahooAPIPtr api(__uuidof(YahooAPI::YahooAPI));
    cout << api->test() << endl;
    cout << "Ask: " << api->GetAsk("MSFT") << endl;
    cout << "Bid: " << api->GetBid("MSFT") << endl;

    CoUninitialize(); // "ukončení" COM API komunikace
    ...
```

Ukázka 7 – Použití knihovny v klientské aplikaci pomocí COM API;
Type library `YahooAPI.tlb` je vygenerována nástrojem `RegAsm`.

3.4 Hostování CLR

3.4.1 Hostování Microsoft .NET CLR

Nepříliš využívanou možností je hostování prostředí CLR uvnitř procesu klientské aplikace. Jedná se o poměrně pokročilé řešení, které umožňuje nastavení velkého množství parametrů a vlastností CLR, včetně například vytvoření vlastního správce vláken nebo paměti (*garbage collectoru*) [21]. Za dobu vývoje platformy .NET vzniklo několik různých API pro hostování CLR v jazycích C a C++, mimo jiné pomocí COM objektů. Ne všechny však podporují běh .NET assembly vytvořených v různých verzí .NET Frameworku [22].

Ve všech případech je však vytvoření CLR prostředí poměrně rozsáhlá rutina, kterou musí vždy programátor zajistit. Obecný postup je přibližně následující:

1) Vytvoření vnějšího prostředí.
2) Načtení CLR s vybranou verzí .NET Frameworku.
3) Získání rozhraní pro komunikaci s CLR.
4) Spuštění CLR.
5) Načtení požadované .NET assembly do výchozí „domény“ CLR.
6) Získání rozhraní pro komunikaci s třídou uvnitř assembly.
7) Spouštění kódu uvnitř třídy, vytváření jejích instancí.

Pokud je naším cílem pouze volání řízeného kódu ve standardním CLR prostředí, pak se jeví hostování CLR jako zbytečně komplikované řešení (viz i ukázka 8).

```
1  double GetBid(char *name) {
2      SAFEARRAY *psaArgs = NULL;
3      variant_t vtRet;
4
5      // Create a safe array to contain the arguments of the method.
6      psaArgs = SafeArrayCreateVector(VT_VARIANT, 0, 1);
7      LONG index = 0;
8      SafeArrayPutElement(psaArgs, &index, &variant_t(name));
9
10     // Invoke the "GetBid" method from the Type interface.
11     ICLR().spType->InvokeMember_3(bstr_t("GetBid"),
                         static_cast<BindingFlags>(BindingFlags_InvokeMethod
                         | BindingFlags_Instance
                         | BindingFlags_Public), NULL, vtObject, psaArgs, &vtRet);
12
13     return vtRet.dblVal;
14 }
```

Ukázka 8 – Obalující funkce pro vyvolání metody `GetBid` na již vytvořené objektu v připraveném CLR prostředí (bez ošetření návratových hodnot).

Ocenit můžeme větší kontrolu nad tvorbou objektů (instancí třídy); samotné volání a především konverze datových typů je však značně složitější, než u předchozích možností. Vývoj komplikuje i nepříliš rozsáhlá dokumentace a omezené možnosti ladění kódu.

3.4.2 Využití platformy Mono a „embedování kódu“

Podobně jako .NET Framework i Mono má možnost hostovat CLI prostředí v C/C++ aplikacích, označováno jako „embedování kódu“ [23]. Princip fungování je v zásadě shodný, API je však jednodušší a přehlednější (za cenu menší kontroly nad během CLI). Práce s objekty a převodem datových typů je na první pohled intuitivnější, než u hostovaného .NET CLR.

I tato možnost je stejně jako samotné Mono multiplatformní a je možné ji využít ve spojení s kompilátory Microsoft C/C++ i GCC/G++. Je třeba si však uvědomit (zejména u platformy Windows), že je nutné spolu s aplikací distribuovat i knihovny Mono.

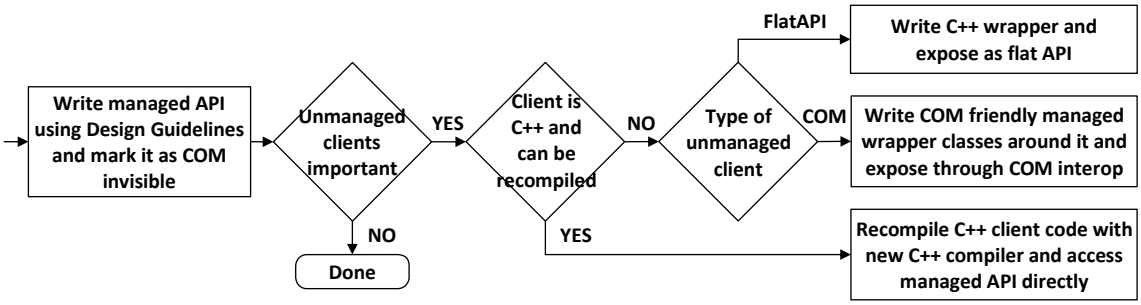
Bohužel možnost embedování kódu stojí poměrně na okraji vývoje projektu, a proto k ní existuje jen velmi krátký návod na použití, který zdaleka nepopisuje všechny možnosti. Zároveň je potřeba se vypořádat s různými nástrahami, jako je například nutnost použití konfiguračního souboru pro načítanou assembly (byť prázdného) nebo chybějící odkaz na knihovnu `libmono` na platformě Windows.

```
1  double GetBid(char *name) {
2      void *args[1];
3      args[0] = mono_string_new(ICLR().domain, name);
4      MonoMethodDesc* method_desc;
5      MonoMethod* method;
6
7      // Find method in class
8      method_desc = mono_method_desc_new("YahooAPI:GetBid(string)", 0);
9      method = mono_method_desc_search_in_class(method_desc, ICLR().klass);
10     mono_method_desc_free(method_desc);
11
12     // Invoke it
13     MonoObject* object = mono_runtime_invoke(method, instance, args, NULL);
14
15     // Get return value
16     return *(double *) mono_object_unbox(object);
17 }
```

Ukázka 9 – Obalující funkce pro vyvolání metody `GetBid` na již vytvořené objektu v prostředí Mono pomocí embedování kódu.

4 Srovnání možností

Následující kapitola se věnuje zejména různým kritickým oblastem a porovnání jednotlivých možností. Porovnávat lze z hlediska různých kritérií (zajímavým kritériem může být rychlost jednotlivých řešení). Je ale především nutné vycházet z možností klientské aplikace a řízené knihovny. Pro základní výběr poskytuje Microsoft na stránkách MSDN následující rozhodovací strom:



Obrázek 3 – Rozhodovací strom pro výběr technologie. (zdroj: [10])

4.1 Kritické oblasti

Všechny možnosti mají několik společných kritických oblastí. Jejich řešení většinou vychází ze stejných podmínek a proto i možnosti jsou vždy principiálně obdobné.

4.1.1 Převod datových typů – marshalling

Marshalling je proces převodu dat mezi různými formáty stejného typu dat. Základní datové typy, jako jsou různě velké číselné typy (v .NET `Byte`, `Int16` až `Int64`, `Double` apod.) jsou obvykle převeditelné bez jakéhokoliv zásahu do samotných dat. Takovým typům se říká *blittable*. Opačným případem jsou *non-blittable* typy, které nelze takto převést. Jedná se o různé interpretace znaků, řetězců, polí až objektů a struktur. .NET obsahuje několik nástrojů, jak marshalling usnadnit (např. atributy `MarshalAs` použitelné pro COM API a různé funkce třídy `Marshal`).

4.1.2 Obsluha výjimek

.NET používá pro obsluhu výjimek jiné cesty než C++ a vzájemná kooperace je obvykle obtížná. Nejvíce možností má programátor při použití C++ Interop / IJW, které mu

umožňuje odchytávat i řízené výjimky. Na úrovni neřízeného C++ kódu lze výjimku odchytit pomocí konstrukce `catch(...)`, nelze však zjistit o výjimce žádné podrobnosti. Proto je nejlepší volbou u možností založených na C++ Interop odchytávat řízené .NET výjimky a v případě nutnosti je předávat do klientské aplikace jako výjimky neřízené. V případě COM API je možné odchytávat výjimky typu `_com_error`, pomocí kterých lze získat informace o původní řízené výjimce.

4.1.3 „Callback" funkce

Občas potřebujeme u nějaké komponenty zaregistrovat funkci, která bude zavolána, když nastane konkrétní událost. Takové funkci se říká *callback*, v názvosloví .NET případně *delegate* a *event handler*. Opět mohou nastat dvě situace – můžeme registrovat neřízenou callback funkci do serverové .NET knihovny, nebo může načtená .NET knihovna registrovat svoji řízenou funkci do neřízené klientské (a v tomto případě hostitelské) aplikace.

Samotná implementace se v podstatě neliší od použití v rámci jednoho programu/programovacího jazyku. Problémy mohou nastat především v převodu datových typů. C++ Interop / IJW nabízí opět nejjednodušší přístup. V P/Invoke lze využít nástrojů třídy `Marshal`, konkrétně funkci `GetDelegateForFunctionPointer`. Pro použití v COM API je možné v .NET knihovně použít atribut `MarshalAs` a převést tím delegate na odkaz na funkci, marshalling parametrů je prováděn automaticky.

4.2 Přehled výhod a nevýhod jednotlivých možností

4.2.1 C++ Interop / IJW

+ Jednoduché použití
+ Snadné rozšíření o další funkcionalitu
− Nutné celou aplikaci překompilovat jako C++/CLI

4.2.2 Platform Invoke

+ Lze použít pro pluginy (bez překompilování klientské aplikace)
− Nutné ručně vytvořit obalující funkce/třídy

4.2.3 Přístup pomocí COM API

+ Automatické generování obalujících struktur

- Nutná registrace COM objektu v systému nebo použití Registration-Free technologie
- Omezení vyplývající z technologie
- Vyšší režie

4.2.4 Hostování CLR

+ Přímá kontrola nad assembly
+ Multiplatformní (Mono)

- Komplikované použití

4.3 Metoda vhodná pro generování C++ wrapperu

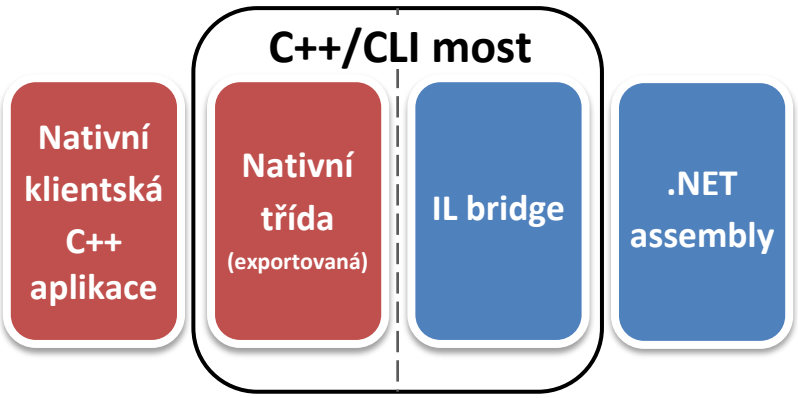
Každá ze zkoumaných možností přináší jak výhody, tak nevýhody. První metoda je však ze své podstaty (tj. přímý přístup k .NET assembly z klientské aplikace) pro potřeby zadání nevhodná, znamená příliš velký zásah do původní neřízené C++ aplikace.

Přístup pomocí COM API se zdá být na první pohled jako nejjednodušší možnost, avšak tato možnost je vykoupena omezenější použitelností a vyšší režií, protože v tomto případě při volání metody musí dojít navíc zejména k vytvoření COM Interop rámce a tzv. CCW – *COM callable wrapperu* (který ověřuje a zajišťuje např. marshalling, existenci objektů, odchyt chyb a výjimek, správu vláken, atd.), na rozdíl od přístupu skrze *Flat API* (tedy možností C++ Interop / IJW a Platform Invoke) [10].

Ze zbývajících možností se jeví jako nejvýhodnější *Platform Invoke*, konkrétně ve variantě používající C++/CLI most. Proto jsem se rozhodl tuto možnost podrobně prozkoumat a implementovat nástroj generující C++ wrappery v této podobě.

5 Realizace propojení pomocí C++/CLI mostu

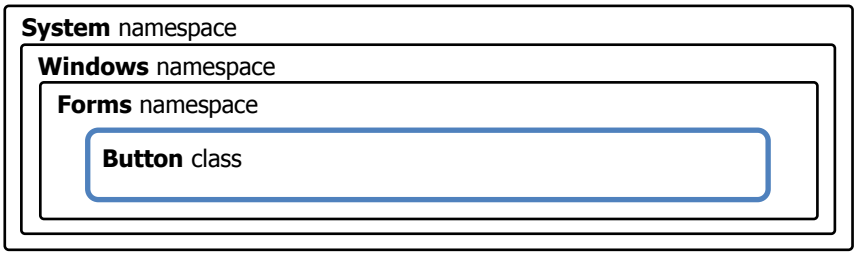
Základní podoba tohoto propojení již byla naznačena v kapitole 3.2.2. Cílem je vytvoření mezivrstvy mezi neřízenou C++ aplikací („klient“) a řízenou .NET assembly („server“; typicky nějaká knihovna funkcí), jak je znázorněno na obrázku 4. Tato mezivrstva musí být napsána v jazyce C++/CLI s využitím technologií *C++ Interop* (z pohledu volání řízené assembly) a *Platform Invoke* (z pohledu volání mostu z neřízené aplikace).



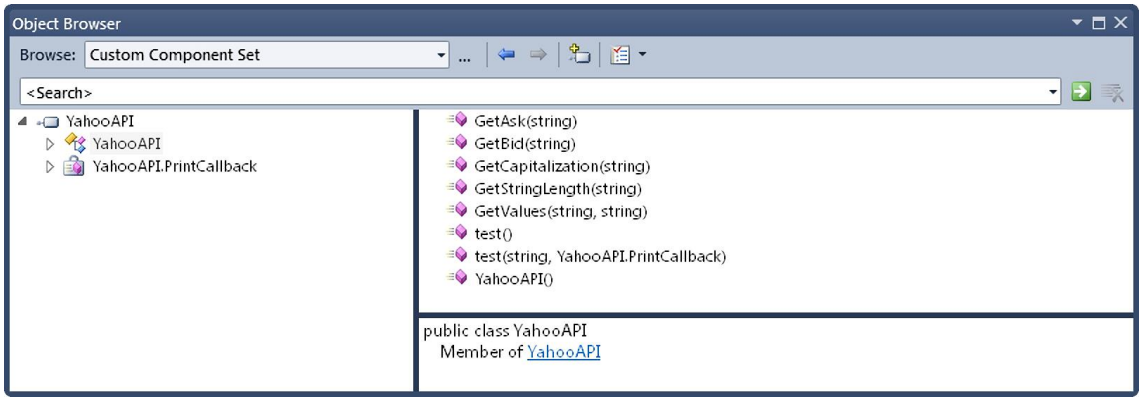
Obrázek 4 – Kontext C++/CLI mostu s naznačeným rozhraním mezi neřízenou a řízenou částí.

5.1 Struktura .NET assembly

.NET assembly si lze představit jako seskupení různých částí, které jsou napsané pomocí některého z jazyků platformy .NET (nejčastěji tedy C#). Jedná se zejména o třídy, které mohou obsahovat všechny klasické prvky objektově orientovaného programování – tedy typicky různé metody (mj. konstruktory) a třídní proměnné (v rámci .NET hovoříme o tzv. *fields* a *properties*). Dále může assembly obsahovat například struktury nebo výčtové typy. Jednotlivé části mohou být seskupené do tzv. *namespaces* (pojmenované prostory tříd a dalších objektů), jedná se však především o kosmetické seskupování, který si lze představit jako prefix všech názvů (znázorněno v obrázku 5). Assembly v sobě nesou metadata, díky kterým je možné snadno zobrazit informací o jejích součástech (viz obrázek 6), a také reference na další assembly, které jsou pro činnost této potřeba.



Obrázek 5 – Umístění třídy `Button` v rámci vnořených namespace `System.Windows.Forms`. Lze chápat jako třídu s názvem `System.Windows.Forms.Button`.



Obrázek 6 – Testovací assembly `YahooAPI` zobrazená pomocí nástroje z Visual Studio 2010.

5.2 Návrh struktury C++/CLI mostu

Smyslem C++/CLI mostu je transparentně zpřístupnit jednotlivé součásti assembly pro použití v neřízené aplikaci tak, aby bylo v ideálním případě možné jednoduše vytvářet samostatné instance tříd, volat jejich metody, umožnit operace mezi nimi (tedy například aby bylo možné předat objekt jako parametr metody), atd. Každá instance řízené třídy by měla být zastoupena jednou instancí neřízené *obalující* třídy.

Struktura mostu by měla odpovídat struktuře .NET assembly. Jazyk C++, pomocí kterého bude klientská aplikace s mostem komunikovat, toto umožňuje zajistit.

Výsledkem po zkompilování C++/CLI mostu je dynamicky linkovaná knihovna (soubor s příponou `.dll`), která bude klientskou aplikací načítána.

Jak již bylo zmíněno i v ukázkách jednotlivých možností propojení (kapitola 3), všechny třídy a funkce, které jsou v tomto mostu označené pro „export“, musejí být deklarovány pouze pomocí jazyka C++.

V následujících kapitolách je popsán návrh mostu z hlediska kódu.

5.2.1 Rozdělení mostu do souborů

Každá součást assembly (tj. například třída), ke které chceme vytvořit C++/CLI most, je zastoupena vlastní sadou souborů. Jazyk C++/CLI, stejně jako C++, používá hlavičkové (`.h`) a zdrojové (`.cpp`) soubory. Most je vždy tvořen minimálně jedním – *hlavním* – hlavičkovým souborem, který obsahuje deklarace prvků exportovaných do DLL. Tento soubor je připraven tak, aby byl použitelný jak během kompilaci C++/CLI mostu, tak jako „popis" načítané knihovny pro klientskou aplikaci (viz ukázky na straně 8).

5.2.2 Namespace

Jazyk C++ podporuje namespace ve stejné podobě, jakou používá i .NET. V mostech je tedy vytvořena struktura shodná s původní assembly.

5.2.3 Kolize názvů

Při tvorbě mostu mohou nastat kolize mezi názvy uvnitř původní .NET assembly a právě tvořeným mostem. Jedním z možných postupů pro zamezení této situace je přidání prefixů či postfixů k názvům všech součástí (řízená třída `System.Windows.Forms.Button` by byla v mostu zastoupena například pomocí obalující třídy `System::Windows::Forms::ButtonWrapper`), přičemž není nutné měnit strukturu namespace. Alternativou je právě změnění této struktury – je možné původní namespace přejmenovat, případně přidat k názvu prefix/postfix.

Další možností je vytvoření nové namespace, která bude celý most obalovat (řízená třída `System.Windows.Forms.Button` by tedy byla v mostu například pod názvem `Wrapper::System::Windows::Forms::Button`). Tento postup se zdá být nejvýhodnější z hlediska použití v klientské aplikaci – v C++ je možné použít direktivu `using` (v tomto případě `using namespace Wrapper`) a poté používat prvky dané namespace bez explicitního uvádění jejího názvu. Z tohoto důvodu byla zvolena právě tato možnost.

5.2.4 Třídy

Most pro třídu řízené .NET assembly je rozložen do tří souborů.

Hlavní hlavičkový soubor obsahuje především deklaraci exportované obalující třídy (tj. deklaraci všech metod, které budou použitelné v klientské aplikaci). Součástí této třídy musí být odkaz na pomocnou třídu, kterou označuji jako *IL bridge* (ve smyslu mostu mezi nativním a řízeným IL kódem). Také jednou z metod je konstruktor s parametrem třídy IL bridge – slouží pro vytvoření obalující třídy v případě, že je řízený objekt návratovou hodnotou nějaké další metody (této, nebo i jiné třídy). Viz ukázka 10.

Dalším souborem je právě **hlavičkový soubor s definicí IL bridge** třídy. Jejím jediným členem je odkaz na řízený objekt umístěný ve speciální třídě `auto_gcroot`, která pomáhá se správou paměti nad používanými řízenými objekty v neřízeném prostředí. Viz ukázka 11.

```
1  #pragma once
2  #include <string>
3
4  #ifndef _LNK
5    #define _LNK __declspec(dllimport) // při kompilaci mostu
6  #endif
7
8  namespace Wrapper { // obalující namespace
9
10     class YahooAPI_IL; // dopředná deklarace pomocné třídy = IL bridge
11
12     class _LNK YahooAPI { // exportovaná obalující třída
13         public:
14             YahooAPI_IL* __IL; // odkaz na IL bridge
15
16             YahooAPI();
17             YahooAPI(YahooAPI_IL* IL); // konstruktor předávající IL bridge
18             ~YahooAPI();
19
20             double GetAsk(std::wstring symbol);
21             ...
22     };
23
24 }
```

Ukázka 10 – Hlavní hlavičkový soubor pro most třídy.

```
#pragma once
#pragma managed
#include <msclr\auto_gcroot.h>
#using "YahooAPI.dll" // import řízené assembly

namespace Wrapper {

    class YahooAPI_IL {
        public:
            msclr::auto_gcroot<::YahooAPI^> __Impl; // odkaz na řízený objekt
    };

}
```

Ukázka 11 – Hlavičkový soubor s IL bridge uchovávající odkaz na instanci řízené třídy.

Posledním souborem mostu třídy je **zdrojový soubor** (`.cpp`), který obsahuje definici třídy z hlavního hlavičkového souboru, tj. kód exportovaných metod.

Součástí konstruktorů je vytvoření instance IL bridge a vytvoření nové instance řízeného objektu. Zároveň je zde i destruktor, který naopak instanci IL bridge z paměti odstraní. O odstranění řízeného objektu se pak postará *garbage collector* navázaný na třídu `auto_gcroot`. Viz ukázka 12 (její součásti jsou popsány dále).

```
#pragma managed
#include "marshaller_ext.h" // rozšíření marshallingu - funkce _marshal_as
#define _LNK __declspec(dllexport)
#include "Wrapper_YahooAPI_IL.h"
#include "Wrapper_PrintCallback_IL.h"

namespace Wrapper {

    YahooAPI::YahooAPI() { // konstruktor
        __IL = new YahooAPI_IL; // vytvoření IL bridge
        __IL->__Impl = gcnew ::YahooAPI(); // instance řízené knihovny
    }

    YahooAPI::YahooAPI(YahooAPI_IL* IL) { // konstruktor s IL bridge
        __IL = IL; // přiřazení IL bridge s řízeným objektem
    }

    YahooAPI::~YahooAPI() { // destruktor
        delete __IL;
    }

    double YahooAPI::GetAsk(std::wstring symbol) { // metoda
        ::System::String^ __Param_symbol = _marshal_as<::System::String^>(symbol);
        ::System::Double __ReturnVal = __IL->__Impl->GetAsk(__Param_symbol);
        return __ReturnVal;
    }
    ...
}
```

Ukázka 12 – Definice exportované třídy.

Uvnitř každé metody je typicky potřeba vykonat jeden až tři kroky:

1) Převést parametry metody na správný datový typ.
2) Vykonat příslušnou akci související s řízenou třídou.
3) Převést návratovou hodnotu na správný datový typ a vrátit ji.

Převodu datových typů se dále věnuje kapitola 5.3.

Vykonáním příslušné akce je myšleno například zavolání metody nebo pozměnění některého členu řízeného objektu (přiřazení hodnoty *field*). Během této akce se typicky přistupuje k instanci řízeného objektu skrze IL bridge (řádek 24 ukázky 12). Metody mohou sloužit k několika účelům:

Volání metod řízeného objektu

Jedná se o nejtypičtější případ, neboť cílem je transparentně zpřístupnit metodu řízené třídy. Viz obalující metoda `GetAsk` v ukázce 12 (řádky 22 až 26) volající metodu `GetAsk` řízeného objektu (řádek 24).

Práce s „fields“ a „properties“

Fields je označení pro třídní proměnné, které mohou být omezeny pouze zvoleným datovým typem nebo klíčovými slovy (například `readonly`). *Properties* jsou prvky, které zajišťují přístup k typicky privátním třídním proměnným a mohou obsahovat určitou programovou logiku (například omezení číselné hodnoty). Přístup z hlediska C++/CLI mostu je v obou případech shodný. Použití je naznačeno v ukázkách 13 a 14.

```
1  private string firstName;  // field - privátní proměnná
2  public string LastName; // field – veřejná proměnná
3  public string Name { get { return firstName+LastName; } } // property; jen čtení
4  public string Nickname { set; } // auto-property; jen zápis
```

Ukázka 13 – Fields a properties v rámci jazyka C#.

```
std::wstring GetLastName() { // získá hodnotu field
    ::System::String^ __ReturnVal = __IL->__Impl->lastName;
    std::wstring __ReturnValMarshaled = _marshal_as<std::wstring>(__ReturnVal);
    return __ReturnValMarshaled;
}
void SetLastName(const std::wstring& value) { // nastaví hodnotu field
    ::System::String^ __Param_value = _marshal_as<::System::String^>(value);
    __IL->__Impl->lastName = __Param_value;
}

std::wstring GetName() { // získá hodnotu property
    ::System::String^ __ReturnVal = __IL->__Impl->Name;
    std::wstring __ReturnValMarshaled = _marshal_as<std::wstring>(__ReturnVal);
    return __ReturnValMarshaled;
}
```

Ukázka 14 – Zpřístupnění fields/properties v rámci C++/CLI mostu.

5.2.5 Výčtové typy

Jedná se o hodnotový datový typ, oproti struktuře je ale jednodušší. Výčtový typ (`enum`) může obsahovat pouze položky jednoduchých číselných datových typů, například `int`, `byte`, `short`, apod., je tedy tzv. *blittable* – převeditelný bez jakékoliv konverze.

V tomto případě vytvoření mostu neznamená propojení s .NET assembly, ale jen definování kompletního výčtového typu v C++ s korespondujícími hodnotami všech položek. Hlavní hlavičkový soubor bude tedy obsahovat pouze výčtový typ zapouzdřený v odpovídající namespace. Viz ukázka 15.

```
namespace Utilities {
    enum LineEnding {
        None = 0,
        Unix = 10,
        Mac = 13,
        Dos = 3338
   };
}
```

Ukázka 15 – „Most“ pro výčtový typ.

5.2.6 Datové struktury

Datové struktury se v .NET chovají jako hodnotové datové typy, tj. přiřazením do proměnné se obsah struktury zkopíruje. Zároveň však struktury mohou obsahovat metody a *properties* s definovanou logikou, kterou při vytváření odpovídající datové

struktury v C++ není možné zajistit (především z důvodu, že není možné z assembly tuto logiku získat). Jako funkční alternativa se nabízí použití stejného postupu jako při vytváření mostů tříd. Výsledkem je vytvoření obalující třídy uchovávající odkaz na řízenou „instanci“ struktury. Bohužel se tímto postupem vytrácí možnost snadno kopírovat obsah struktury mezi proměnnými v klientské aplikaci.

5.2.7 Generické datové typy

V rámci .NET je možné vytvářet datové typy, které jsou parametrizovány jinými datovými typy – označují se jako generické. Pro představu si můžeme vzít třídu `List<T>`, která představuje dynamicky alokovaný seznam položek typu `T`. Například seznam čísel bude vytvořen již „konkretizovanou“ třídou `List<int>`. Parametr `T`, v tomto případě `int`, je následně možné uvnitř třídy používat. V případě našeho seznamu čísel to znamená, že metoda `Add` pro přidání položky do seznamu má parametr typu `int`.

Jazyk C++ používá namísto generických datové typů tzv. šablony – *templates*. Jejich použití je ale odlišné a není možné jednoduše vytvořit univerzální most pro nekonkretizovaný generický datový typ. To je poměrně zásadní problém ve chvíli, kdy nějaká metoda z „obyčejné“ (negenerické) třídy vrací třeba již zmíněný seznam čísel – `List<int>`. V tomto případě je ale již možné vytvořit most pro generickou třídu s jedním konkrétním parametrem. Pokud bude v jiné metodě použit například seznam znaků – `List<char>` – tak bude nutné vytvořit další most pro tuto variantu.

Toto řešení jsem se rozhodl použít v rámci nástroje, kdy generované mosty pro generické třídy budou v názvu obsahovat i parametry oddělené podtržítky. Pro `List<int>` bude tedy vytvořena obalující třída s názvem `List__int`.

5.3 Datové typy a marshalling

Důležitou součástí mostu je zajištění správného převodu datových typů mezi nativním a řízeným prostředím. Typicky je využíváno marshallingu u všech metod (včetně konstruktorů) pro zpracování vstupních parametrů a výstupních hodnot.

Knihovny pro jazyk C++/CLI s sebou přináší několik funkcí, které pomohou s převodem základních typů. Jedná se zejména o funkci `marshal_as`, která je definována jako dále rozšiřitelná šablona.

Všechna rozšíření této funkce jsou v rámci wrapperu uložena v hlavičkovém souboru `marshaller_ext.h`. Některá rozšíření ale nebylo možné jednoduše přidat k existujícím funkcím `marshal_as`, byla proto vytvořena nová pomocná funkce `_marshal_as`, pomocí které se v mostu marshalling provádí.

Ve výchozím stavu zajišťuje funkce `marshal_as` převod zejména mezi různými typy řetězců oběma směry. V rámci mostu jsem se rozhodl pro používání třídy `std::wstring`, která umožňuje snadnou práci s unicode řetězci.

Dalším důležitým datovým typem jsou pole. Z hlediska výkonu a efektivnosti by bylo nejlepším možným řešením „přemapování“ paměti pole (data jsou v takovém případě v paměti jen jednou a přistupuje k nim jak řízená, tak nativní aplikace). Taková možnost má však dvě zásadní nevýhody – lze ji použít pouze pro jednoduché (*blittable*) datové typy a je nutné zajistit správné uvolnění paměti pro *garbage collector*. Také je nutné zajistit tzv. přišpendlení – *pinning*. Slouží k tomu, aby se objekt nemohl v paměti přesunout, pokud se spustí *garbage collector*. To má ale různá další omezení a není například možné takto přišpendlený ukazatel použít jako návratovou hodnotu. Vzhledem k těmto okolnostem a také snaze o transparentnost mostu (tak, aby v klientské aplikaci nebylo nutné řešit „nic navíc“) je nejjednodušší a vždy funkční možností obsah pole kopírovat.

Nejsnáze použitelným ekvivalentem řízeného statického pole (v rámci C++/CLI pod názvem `cli::array<>`) je třída `std::vector<>`, která reprezentuje dynamicky alokované pole (dochází zde tedy k mírné změně funkcionality). Překopírování dat se skládá pouze z průchodu vstupním polem, během kterého je v každé iteraci aktuální prvek převeden na správný datový typ a přidán do výstupního pole (viz ukázka 16).

```
1  template<typename TTo, typename TFrom>
2  inline cli::array<TTo>^ marshal_as(const std::vector<TFrom>& from) {
3      size_t len = from.size();
4      cli::array<TTo>^ __ReturnVal = gcnew cli::array<TTo>(len);
5
6      for(size_t i = 0; i < len; i++) {
7          __ReturnVal[i] = _marshal_as<TTo>(from[i]);
8      }
9
10     return __ReturnVal;
11 }
```

Ukázka 16 – Rozšíření marshallingu pro převod nativního pole (`std::vector<>`) na řízené pole (`cli::array<>`).

Marshalling jsem se rozhodl dále rozšířit o práci se základními řízenými kolekcemi, jako je například `List`, `LinkedList`, `HashSet` apod., opět ve spojení s třídou `std::vector`. Tento převod není nutný – jednou z metod těchto tříd bývá zkopírování hodnot v kolekci do nového statického pole (pojmenovaná `ToArray`); je tedy možné vytvořit pro dané kolekce odpovídající mosty a používat kolekce skrze ně.

V mostech tříd mohou být metody, které přebírají jako parametr instanci další jiné třídy. Při zpracování parametrů je nutné odstranit jejich „obal“ a předat dále pouze řízený objekt – k tomuto účelu je vždy v každém hlavičkovém souboru s IL bridge přidáno další rozšíření marshallingu, které právě toto zajišťuje pro danou konkrétní třídu. Právě z důvodu, aby marshaller mohl získat odkaz na původní řízený objekt, je IL bridge v obalující třídě označen jako veřejný člen.

6 Nástroj pro generování C++/CLI mostu

Tvorba C++/CLI mostů je poměrně rutinní činností, je tedy možné vytvořit nástroj, který bude toto generování obstarávat. V následující kapitole se věnuji právě vytvořenému nástroji.

6.1 Návrh funkcionality

Nástroj musí umožňovat pohodlné generování C++/CLI mostů pro .NET assembly a měl být mít uživatelsky přívětivé rozhraní. Mělo by být možné snadno přidávat a odebírat assemblies a generovat mosty pro různé jejich kombinace (tzn. i generovat mosty pro více assemblies najednou). Ke každé assembly by se měl zobrazovat přehled součástí a měl by být možný jejich výběr. Uživatele by měl nástroj informovat o dokončení nebo chybě během generování mostů. Vygenerované soubory by měly být čitelné a v ideálním případě obsahovat komentáře získané z původní assembly.

6.2 Implementace

Pro vytvoření nástroje byl zvolen jazyk C#, který umožňuje jednoduchou a rychlou tvorbu aplikací s grafickým uživatelským rozhraním a zároveň poskytuje nástroje pro zjišťování informací o .NET assembly (viz kapitola 6.5).

6.3 Uživatelské rozhraní

Vytvořený nástroj má jednoduché uživatelské rozhraní, jehož dominantní částí je stromové zobrazení přidaných assemblies, které u každé položky zobrazuje ikonu odpovídajícího typu a zaškrtávací pole. Při zaškrtnutí položky dojde automaticky k zaškrtnutí všech vnořených položek. Tímto způsobem může uživatel snadno vybrat všechny součásti, pro které chce generovat C++/CLI most.

Dále rozhraní obsahuje textové pole spolu s tlačítkem pro výběr výstupní cesty (tj. kam budou soubory mostu uloženy) a tlačítka pro přidání resp. odebrání assembly a spuštění generování. Assembly je možné přidat také přetažením příslušného souboru na okno aplikace. U horního okraje aplikace se nachází jednoduché menu.

Typický postup práce s nástrojem je následující:

1) Spuštění nástroje.
2) Přidání .NET assembly pomocí tlačítka nebo přetažením na okno aplikace.
3) Výběr požadovaných součástí pomocí zaškrtávacích polí.
4) Stisknutí tlačítka „Generovat“.

6.4 Struktura aplikace

Aplikace byla psána s cílem vytvořit univerzální nástroj umožňující generovat různé soubory (tj. nejen C++/CLI mosty) v návaznosti na .NET assembly. Je členěna do samostatných souborů s oddělenou logikou generování od uživatelského rozhraní.

Skládá se ze dvou grafických oken (formulářů):

- `FormMain.cs` – hlavní okno aplikace; jeho součásti byly popsány v předchozí kapitole.
- `FormInfo.cs` – statické okno s informacemi o aplikaci a pár tipech k používání. Dále je zde uveden kontakt na autora aplikace a odkaz na repozitář projektu v rámci služby GitHub[6], obsahující zdrojové kódy k aplikaci.

Dále obsahuje tyto soubory:

- `FixedTreeView.cs` – grafická komponenta stromového zobrazení – opravuje problém s dvojklikem v původní implementaci (problém se vyskytoval ve spojení s událostí `AfterCheck` po zaškrtnutí položky stromu).
- `IEnumerableExtension.cs` – rozšíření tříd implementujících rozhraní `IEnumerable`. Přidává metodu `ForEach` pro vykonání akce nad každou položkou seznamu.
- `TypeConverter.cs` – třída `TypeConverter` sloužící pro generování názvů datových typů v řízeném a neřízeném kódu. Používá se například pro parametry metod, návratové hodnoty, třídní proměnné, apod. Příklad: pro datový typ „řetězec“ vytvoří dvojici `std::wstring` (pro použití v nativním kódu) a `::System::String^` (pro řízený kód). Dále poskytuje různé informace informace, např. zda je nutné provádět při převodu marshalling.

[6] `https://github.com/` – Platforma pro hostování open-source projektů.

- `Utils.cs` – obsahuje statické funkce pro úpravu názvů do podoby použitelné v C++/CLI (nahrazení nepovolených znaků, převedení názvu namespace do podoby používající „čtyřtečky“, …), a funkce pro získání pomocných názvů (například pro třídu obsahující IL bridge, obalující namespace, lokální proměnné používané během marshallingu apod.).
- `XmlDocHelper.cs` – pomocná třída zpracovávající XML soubory s anotacemi pro .NET assembly (tzv. dokumentační komentáře). Takové soubory lze typicky získat během kompilace assembly[7]. Třída umožňuje najít pro vybranou assembly dokumentační komentáře k třídám, metodám, apod. Tyto komentáře jsou přidávány do hlavních hlavičkových souborů jednotlivých obalujících tříd mostu (tj. do souborů, které obsahují deklarace exportovaných tříd a následně se používají v klientské aplikaci).
- `Generator.cs` – abstraktní třída `Generator` definující strukturu generátorů. Od této třídy jsou odvozeny všechny konkrétní generátory, kterým poskytuje jedinou službu – zápis jejich výstupu do souboru.
- Skupina souborů `Wrapper*.cs` – jedná se o sadu generátorů vytvořených pro generování C++/CLI mostů. Podrobný popis naleznete v kapitole 6.6.

6.5 Procházení .NET assembly

Nezbytným předpokladem pro generování C++/CLI mostů je možnost procházení struktury .NET assembly, tj. schopnost načítat metadata obsahující informace o součástech assembly a dále zjišťovat podrobnosti nutné k implementaci. Tato technika se nazývá reflexe a .NET framework pro ni poskytuje různé nástroje v rámci namespace `System.Reflection`. V rámci vytvořeného nástroje se reflexe používá ve dvou situacích.

Nejprve se uplatní při přidání assembly do stromového zobrazení postupně v několika úrovních:

1) Získání základních informací pomocí třídy `System.Reflection.Assembly` (název, verze, …) a získání rozhraní pro načítání dalších podrobností. V tomto kroku se assembly přidá se do stromu jako nová kořenová položka.

[7] Například pomocí parametru `/doc:file.xml` pro C# kompilátor `csc`.

2) Získání všech **součástí** assembly – každá součást je reprezentována objektem třídy `Type` a může se jednat například o třídu, strukturu, výčtový typ apod. Tyto součásti jsou přidány do stromu jako podpoložky assembly.

3) Získání **členů** každé součásti. Jedná se o metody, třídní proměnné, položky výčtových seznamů, atd. Tyto členy jsou v rámci reflexe zastoupeny vždy konkrétní třídou – například pro metody se jedná o třídu `MethodInfo`. Z těchto tříd je možné získat podrobné informace, například název a umístění v rámci namespace, seznam parametrů, návratový typ, … Do stromu jsou přidány jako podpoložky *součástí*.

Dále se reflexe použije během samotného generování C++/CLI mostů. K tomu se hodí zejména informace získané ve třetí úrovni, tedy informace o členech jednotlivých součástí, pomocí kterých lze vytvořit např. obalující metody se stejným *podpisem* (názvem a parametry), jaký měla původní řízená metoda.

6.6 Generování C++/CLI mostů

Generování je zajištěno již zmíněnou sadou generátorů, přičemž každý generátor se stará o jinou část mostu. Generátory mají strukturu předepsanou třídou `Generator` a mohou mít definované následující metody:

- `AssemblyLoad` – volána pro každou nově zpracovávanou assembly.
- `EnumLoad` – volána pro vytvoření mostu pro výčtové typy.
- `ClassLoad` – volána pro vytvoření mostu pro třídy.
- `GeneratorFinalize` – volána po předání všech dat určených ke zpracování, tj. pro dokončení generování.

Generování je spuštěno po kliknutí na tlačítko „Generate“. Vytvořené generátory jsou nejprve přidány do seznamu („zřetězeny“). Následně probíhá procházení jednotlivých zaškrtnutých položek ve stromovém zobrazení. Pro každou zaškrtnutou položku je zavolána příslušná metoda všech generátorů v seznamu. Nakonec je na všech generátorech zavolána metoda `GeneratorFinalize` a tím je proces generování ukončen. V případě, že se v průběhu generování vyskytne chyba, je vyhozena výjimka, kterou grafické rozhraní oznámí uživateli srozumitelnou hláškou.

Do metod `EnumLoad` a `ClassLoad` se předává kromě samotného „typu“ (instance třídy `Type`) také seznam *fields* (pro výčtové typy a struktury), respektive seznamy *fields*, konstruktorů a metod (pro třídy) vybraných uživatelem v grafickém rozhraní.

Během generování se všechny získané „typy“ (kromě jednotlivých součástí assembly i parametry metod, návratové hodnoty, atp.) zpracovávají pomocí vytvořené třídy `TypeConverter`, která poskytuje varianty jednotlivých typů použitelné v řízeném a neřízeném kódu. Viz popis třídy v kapitole 6.4.

Pro samotné generování C++/CLI byla vytvořena sada následujících generátorů:

- `WrapperHeaderGenerator` – generuje hlavní hlavičkové soubory mostů.
- `WrapperSourceGenerator` – generuje zdrojové soubory s definicemi k hlavním hlavičkovým souborům.
- `WrapperILBridgeGenerator` – generuje hlavičkové soubory s IL bridge a rozšířeními pro marshalling.
- `WrapperILDelegateGenerator` – oddělený speciální případ předchozího generátoru – vytváří most pro callback funkce (delegáty).
- `WrapperMockupGenerator` – pomocný generátor, který zajišťuje existenci chybějících tříd vytvořením prázdných.
- `WrapperProjectGenerator` – generátor, který pro most vytvoří projektový soubor (`.vcxproj`) umožňující snadné zkompilování[8].

6.7 Rozšiřitelnost aplikace

Nástroj byl psán tak, aby jej bylo možné použít i pro generování jiných souborů než jen C++/CLI mostů. Postačí vyměnit stávající sadu generátorů za jinou.

Funkčnost generátorů lze v případě potřeby snadno dále rozšiřovat. Současná sada byla vytvořena na základě předem určené požadované funkčnosti, je tedy možné, že pro některé situace nebude funkční nebo jen vyhovující. Viz i závěry testování v následující sekci.

[8] Projektové soubory lze zkompilovat například pomocí IDE Visual Studio nebo nástrojem `msbuild`.

7 Testování funkčnosti

V této kapitole se zaměřím především na testování funkčnosti především zvolené metody, které probíhalo během vývoje nástroje i poté. Podívám se blíže na nově objevené kritické oblasti a pokusím se o porovnání s jinými metodami propojení.

7.1 Modelová třída `YahooAPI`

Jako první pro testování jsem si zvolil jednoduchou třídu `YahooAPI`, která byla vytvořena v rámci hledání vhodné metody propojení .NET assembly s řízeným C++ (viz strana 5). Třída obsahuje 6 poměrně jednoduchých metod a jednu metodu obsahující callback funkci. Pro samotný most byly vygenerovány 4 soubory se zdrojovým kódem.

Testování spočívalo v měření času trvání běžících metod spuštěných v mnoha iteracích po sobě (konkrétně byla každá metoda spuštěna 10 000 000×). Pro vyloučení chyby proběhlo každé měření vícekrát a pro porovnání použit průměr.

Jako použitelné metody této třídy jsem vybral tyto:

1. `string test()` – vrátí vždy stejný řetězec.
2. `int GetStringLength(string str)` – vrátí délku zadaného řetězce.
3. `void test(string mode, PrintCallback callback)` – zavolá `callback` s řetězcem zkombinovaným s parametrem `mode`.

Porovnával jsem vygenerovaný C++/CLI most s přístupem pomocí COM API. Pro srovnání byla přidána i aplikace napsaná v jazyce C# se stejnou funkčností.

		Metoda 2		
	Metoda 1	300 znaků	600 znaků	**Metoda 3**
C++/CLI most	2,271 s	3,131 s	4,487 s	7,003 s
COM API	4,014 s	14,806 s	22,980 s	17,893 s
C#	0,065 s	0,039 s	0,039 s	3,101 s

Tabulka 1 – Výsledky testování třídy `YahooAPI`.

Z tabulky je jednoznačně vidět, že výkonnost obou způsobů propojení C++ a .NET je diametrálně horší než C# aplikace, která má ke třídě přímý přístup a není tedy zatížena přechodem mezi řízeným a neřízeným kódem a marshallingem.

Pokud se zaměříme pouze na srovnání C++/CLI mostů a propojení pomocí COM API, je ve všech případech C++/CLI most rychlejší (u druhé metody až pětkrát).

Je třeba si však uvědomit, že se tento test zaměřil pouze na opakované volání jedné metody a také byl zatížen nutností marshallingu textových řetězců.

7.2 Modelová assembly `TestSuite`

`TestSuite` je assembly, která začala vznikat během vývoje nástroje a byly do ní postupně přidávány a testovány nově podporované prvky generování. Obsahuje tyto součásti:

- Výčtový typ.
- Jednoduchou strukturu.
- Třídu obsahující *fields* a *properties*.
- Generickou třídu.
- Třídu s mnoha metodami vracejícími různá pole a kolekce; metodami pro práci s předchozími prvky; metodou s callback funkcí.

Tato třída je tedy vhodná pro porovnání funkčnosti a možnostech použití jednotlivých wrapperů. Vzhledem k tomu, že pomocí této třídy byl nástroj tvořen, je v C++/CLI mostu obsaženo a zpřístupněno vše, co se v assembly nachází. Jedinou výjimkou je generická třída, která je obsažena pouze v konkretizované variantě (viz rozbor v kapitole 5.2.7).

Pokud pro porovnání opět vezmu přístup pomocí COM API s automaticky vygenerovanými wrappery, jsem již během registrace COM komponenty do systému informován o nedostupnosti některých částí – konkrétně není možné zpřístupnit:

- Obsaženou generickou třídu.
- Metodu, která vrací instanci této generické třídy.
- Čtyři metody, které přebírají nebo vracejí generické seznamy čísel a řetězců.
- Dvě metody vracející vnořená pole (tzv. *jagged*).

Dále je nutné počítat s nezařazením všech statických prvků, které vychází z principu fungování COM objektů (tj. přístupu ke všem objektům pomocí rozhraní). Toto je záležitost vzniklá už během kompilování assembly a uživatel o ní není informován. Pro testování byly proto všechny statické prvky pro COM API nahrazeny nestatickými.

Vzhledem k tomu, že testování třídy `YahooAPI` bylo zatíženo marshallingem řetězců, rozhodl jsem se otestovat stejnou metodikou tři metody z této assembly:

1. `int GetInt()` – vrátí vždy stejné číslo.
2. `int[] GetInts1()` – vrátí statické pole čísel.
3. `Stuff GetObject()` – vrátí novou instance jednoduché třídy.

	Metoda 1	**Metoda 2**	**Metoda 3**
C++/CLI most	0,373 s	2,307 s	7,952 s
COM API	0,581 s	4,377 s	19,647 s
C#	0,036 s	0,145 s	0,137 s

Tabulka 2 – Výsledky testování assembly `TestSuite`.

Jak je vidět z výsledků v tabulce 2, výkonnost jednotlivých řešení se od prvního testování významně neliší. Opět vidíme, že čistě řízená aplikace napsaná v C# nabízí nedosažitelně lepší výkon. V rámci srovnání prvních dvou metod odchází jako vítěz opět C++/CLI.

7.3 Knihovna DotNetZip

Knihovna DotNetZip (také známá jako Ionic Zip Library) je .NET open-source knihovna obsahující sadu nástrojů pro práci s komprimovanými soubory ve formátu ZIP. Byla vybrána jako příklad reálně použitelné knihovny v klientských neřízených C++ aplikací. Umožňuje uživateli vytvářet nové ZIP archivy, nebo manipulovat s již existujícími (měnit jejich strukturu, přidávat nebo mazat obsažené soubory, apod.) [24].

Tato knihovna začala být testována v závěru vývoje nástroje s původním cílem otestovat funkčnost základních funkcí. Viz příklad 17 ukazující jednoduchou C++ aplikaci využívající knihovnu pro vytvoření zip archivu s jedním souborem.

```cpp
#pragma comment(lib, "Wrapper")
#include <stdlib.h>
#include "Wrapper_Ionic_Zip_ZipFile.h"

using namespace Wrapper::Ionic::Zip;

int main()
{
   ZipFile *zip = new ZipFile();
   zip->AddFile(L"somefile.txt");
   zip->Save(L"out.zip");

   return 0;
}
```

Ukázka 17 – Klientská C++ aplikace využívající knihovnu DotNetZip pomocí C++/CLI mostu.

Během testování nebylo na počátku možné z různých příčin vytvořit most pro celou knihovnu a bylo nutné vybírat pouze jednodušší součásti. Problémy byly zejména s různými datovými strukturami (například `System.DateTime`), které nebylo možné v IL bridge uchovávat jako hodnotový datový typ – problém byl vyřešen změněním na uchovávání referencí na řízené struktury. Dále bylo dopracováno hledání závislostí na dalších assemblies a tedy tvoření mostů pro všechny použité datové typy. Bylo zjištěno, že v jazyce C++/CLI se namísto metod `Dispose` (sloužících pro odstranění objektů) používá destruktor a je tedy nutné generování těchto metod pozměnit. Změněno bylo také pojmenovávání jednotlivých souborů vygenerovaného mostu – v původní verzi se v názvu souborů používal pouze název samotné třídy (resp. struktury nebo výčtového typu). Tím mohlo dojít ke kolizím mezi jednotlivými assemblies. Ve finální verzi nástroje se pro pojmenování používá název tříd včetně namespace.

Výsledná verze nástroje po všech provedených úpravách dokáže pro celou tuto knihovnu vygenerovat most, který je následně zkompilovatelný a použitelný v klientské C++ aplikaci.

Knihovna DotNetZip v sobě zahrnuje i COM rozhraní [25] a je připravena tak, aby byla pomocí COM API kompletně přístupná. Výkonnostní porovnání je tedy možné, avšak jednalo by se opět o opakování stejně koncipovaných testů. Vzhledem k povaze knihovny nemá také tak velký význam – v klientské aplikaci pravděpodobně nebudeme vytvářet desítky milionů ZIP souborů během krátké doby.

7.4 Zhodnocení řešení

Kromě zmíněné knihovny DotNetZip byla funkčnost ověřena ještě s dalšími dostupnými .NET knihovnami (například matematickou knihovnou Math.NET[9] [26]) a bylo tak ověřeno, že nástroj je v praxi použitelný.

Práce s C++/CLI mostem je subjektivně pohodlnější než práce s COM API, zejména díky hlavnímu hlavičkovému souboru, který je přehlednější než soubory vygenerované pro COM. C++/CLI most také lépe kopíruje strukturu původní .NET třídy a podporuje i prvky, které během automatického generování wrapperů pro COM API není možné zpřístupnit (zejména se jedná o generické třídy a statické metody). Pokud se tedy programátor rozhoduje mezi těmito dvěma možnostmi, zdá se být použití C++/CLI mostů lepší volbou. Vytvořený nástroj může být v takovém případě značným usnadněním práce.

Je však zřejmé, že pokud bude klientská aplikace využívat .NET assembly ve velké míře a provádět mnoho přístupů do řízených tříd a struktur, bude režie představovat zásadní zpomalení aplikace. Hodí se tedy spíše do aplikací, kde nebude probíhat tak intenzivní přecházení mezi řízeným a neřízeným kódem.

[9] V rámci knihovny Math.NET bylo nalezeno pár částí, které nástroj neumí v tuto chvíli správně vygenerovat (například metody vracející třídu `Func<>`). Tyto části byly prozatím z generování odebrány, avšak není problém je do nástroje později doimplementovat.

8 Závěr

Tato práce měla za cíl prozkoumat oblast propojení programů napsaných v jazyce C++ s .NET assembly a srovnat jednotlivé možnosti. Další částí bylo vybrání vhodné metody a vytvoření nástroje, který by realizaci tohoto propojení uživatelům usnadnil.

Byly prozkoumány celkem 4 možnosti propojení neřízených C++ aplikací a řízených .NET assembly. Ke všem možnostem byly vytvořeny ukázkové programy, na základě kterých byly možnosti porovnány. Byly zjištěny přínosy, ale samozřejmě i různá úskalí jednotlivých metod. Nakonec byla vybrána jedna konkrétní možnost tohoto propojení – C++/CLI mosty – a na základě ní navrhnut postup automatického generování C++ wrapperů a taktéž implementován nástroj, který toto generování zajišťuje.

Nástroj byl vytvořen s cílem přinést uživateli možnost automaticky generovat wrappery pro .NET assembly. Nabízí jednoduché intuitivní rozhraní, pomocí kterého lze snadno vybírat jednotlivé .NET třídy z různých assembly, ke kterým má být wrapper vytvořen. Výstupem nástroje sada zdrojových souborů představující C++/CLI mosty, které lze snadno zkompilovat a použít v C++ aplikaci.

Dále bylo provedeno testování vytvořeného nástroje a navrhnutého řešení v porovnání s jinými metodami. Porovnáváno bylo především s metodou propojení pomocí COM API, oproti kterému přináší vytvořené řešení jednoznačné výhody.

Všechny body zadání tedy byly splněny a dosažené výsledky jsou uspokojující. Vytvořený nástroj by bylo vhodné dále testovat a rozšířit jej o všechny konstrukce, které se mohou v .NET assembly nacházet. Nástroj je také možné dále rozšiřovat o jiné metody propojení, například s ohledem na multiplatformnost. Dále by šlo upravovat funkcionalitu nástroje a přidat tak např. možnost měnit strukturu namespace vygenerovaného wrapperu.

Reference

[1] KAČMÁŘ, Dalibor. *Programujeme .NET aplikace ve Visual Studiu .NET*. Praha: Computer Press, 2001. ISBN 80-7226-569-5.

[2] MSDN Library. *Getting Started with the .NET Framework*. [Online] [Citace: 10. 12. 2014]. Dostupné z: http://msdn.microsoft.com/en-us/library/hh425099(v=vs.110).aspx

[3] MSDN Library. *Assemblies in the Common Language Runtime*. [Online] [Citace: 10. 12. 2014]. Dostupné z: http://msdn.microsoft.com/en-us/library/hk5f40ct(v=vs.90).aspx

[4] Wikipedia: the free encyclopedia. *Common Language Runtime*. [Online] [Citace: 11. 12. 2014]. Dostupné z: http://en.wikipedia.org/wiki/Common_Language_Runtime

[5] MSDN Library. *Overview of the .NET Framework*. [Online] [Citace: 26. 12. 2014]. Dostupné z: http://msdn.microsoft.com/en-us/library/zw4w595w.aspx

[6] Mono. *About Mono*. [Online] [Citace: 20. 12. 2014]. Dostupné z: http://www.mono-project.com/docs/about-mono/

[7] Mono. *Compatibility*. [Online] [Citace: 9. 6. 2015]. Dostupné z: http://www.mono-project.com/docs/about-mono/compatibility/

[8] Mono. *Companies using Mono*. [Online] [Citace: 20. 12. 2014]. Dostupné z: http://www.mono-project.com/docs/about-mono/showcase/companies-using-mono/

[9] MSDN Blogs – Community Goodies. *Interop with Native C++*. [Online] 14. 7. 2010 [Citace: 6. 12. 2014]. Dostupné z: http://blogs.msdn.com/b/msdnforum/archive/2010/07/14/interop-with-native-c.aspx

[10] MSDN Library. *An Overview of Managed/Unmanaged Code Interoperability*. [Online] [Citace: 6. 12. 2014]. Dostupné z: http://msdn.microsoft.com/en-us/library/ms973872.aspx

[11] Pragmateek. *Using C# from native C++ with the help of C++/CLI (fixed and enhanced)*. [Online] 7. 3. 2013 [Citace: 6. 12. 2014]. Dostupné z: http://pragmateek.com/using-c-from-native-c-with-the-help-of-ccli-v2/

[12] BASU, Abhinaba. MSDN Blogs – I know the answer (it's 42). *C++/CLI and mixed mode programming*. [Online] 14. 11. 2012 [Citace: 6. 12. 2014]. Dostupné z: http://blogs.msdn.com/b/abhinaba/archive/2012/11/14/c-cli-and-mixed-mode-programming.aspx

[13] MSDN Library. *Exporting from a DLL*. [Online] [Citace: 6. 1. 2015]. Dostupné z: http://msdn.microsoft.com/en-us/library/z4zxe9k8.aspx

[14] CodeProject. *Simple Method of DLL Export without C++/CLI*. [Online] 28. 6. 2009 [Citace: 12. 12. 2014]. Dostupné z: http://www.codeproject.com/Articles/37675/Simple-Method-of-DLL-Export-without-C-CLI

[15] CodeProject. *How to Automate Exporting .NET Function to Unmanaged Programs*. [Online] 22. 11. 2006 [Citace: 12. 12. 2014]. Dostupné z: http://www.codeproject.com/Articles/16310/How-to-Automate-Exporting-NET-Function-to-Unmanage

[16] Web – Robert Giesecke. *Unmanaged Exports*. [Online] 9. 7. 2009 [Citace: 12. 12. 2014]. Dostupné z: https://sites.google.com/site/robertgiesecke/Home/uploads/unmanagedexports

[17] Microsoft. *COM: Component Object Model Technologies*. [Online] [Citace: 6. 12. 2014]. Dostupné z: https://www.microsoft.com/com/default.mspx

[18] TROELSEN, Andrew. *COM and .NET Interoperability*. Berkeley, CA: Apress, 2002. ISBN 1590590112.

[19] MSDN Library. *Exposing .NET Framework Components to COM*. [Online] [Citace: 12. 12. 2014]. Dostupné z: http://msdn.microsoft.com/en-us/library/zsfww439.aspx

[20] MSDN Library. *Registration-Free COM Interop*. [Online] [Citace: 6. 1. 2015]. Dostupné z: http://msdn.microsoft.com/en-us/library/fh1h056h(v=vs.110).aspx

[21] MSDN Magazine. *CLR Inside Out: CLR Hosting APIs*. [Online] 8 2006 [Citace: 26. 12. 2014]. Dostupné z: http://msdn.microsoft.com/en-us/magazine/cc163567.aspx

[22] MSDN Developer Network – Samples. *C++ app hosts CLR 4 and invokes .NET assembly (CppHostCLR)*. [Online] 11. 6. 2012 [Citace: 26. 12. 2014]. Dostupné z: https://code.msdn.microsoft.com/CppHostCLR-e6581ee0

[23] Mono. *Embedding Mono*. [Online] [Citace: 26. 12. 2014]. Dostupné z: http://www.mono-project.com/docs/advanced/embedding/

[24] DotNetZip Library. *Home*. [Online] [Citace: 5. 6. 2015]. Dostupné z: http://dotnetzip.codeplex.com/

[25] DotNetZip Library. *DotNetZip can be used from COM Environments*. [Online] [Citace: 5. 6. 2015]. Dostupné z: http://dotnetzip.herobo.com/DNZHelp/Code%20Examples/COM.htm

[26] Math.NET. [Online] [Citace: 10. 6. 2015]. Dostupné z: http://www.mathdotnet.com/

Přílohy

A Uživatelská příručka

Spuštění a kompilace nástroje

Vytvořený nástroj (pojmenovaný *CppCliBridgeGenerator*) lze spustit pomocí souboru `CppCliBridgeGenerator.exe`. Aplikace ke svému běhu vyžaduje Microsoft .NET Framework 4.

Nástroj byl vytvořen v IDE Microsoft Visual Studio 2010 pomocí jazyka C#. Projekt se zdrojovými soubory se nachází ve složce `CppCliBridgeGenerator`.

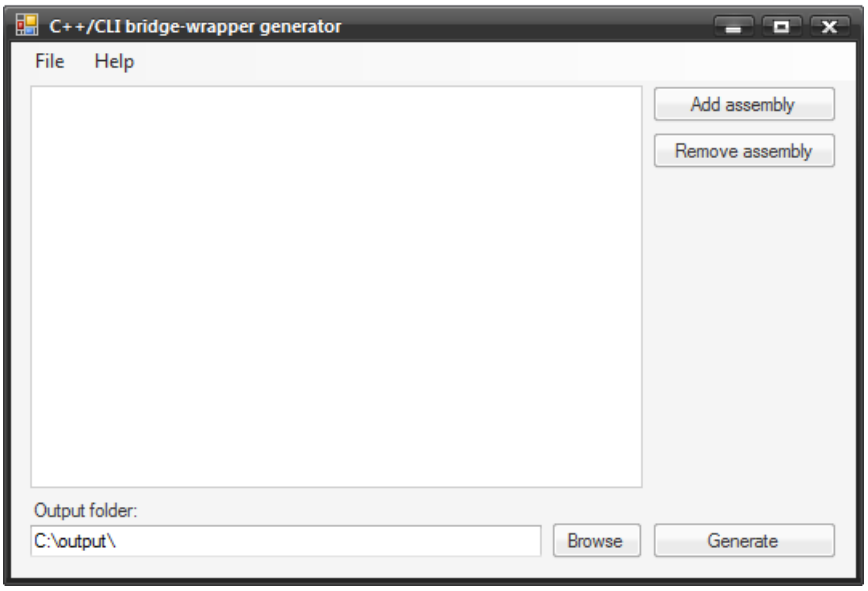
Obsluha nástroje

Po spuštění se uživateli zobrazí jednoduché okno (viz obrázek A1) obsahující panel s přidanými assemblies, tlačítka pro přidání resp. odebrání assembly, textové pole s cestou výstupní složky a tlačítko pro spuštění generování. Přidané assemblies se zobrazují ve stromovém zobrazení jako kořenové položky. Při rozbalení položky s assembly získáme obsažené datové typy. Obdobně při rozbalení datového typu získáme informace o jeho členech (obrázek A2). Jednotlivé assembly, datové typy a členy je možné vybírat pomocí zaškrtávacích polí a tím měnit rozsah generovaného wrapperu. Pokud při přidání assembly dojde k chybě (například když vložený soubor není .NET assembly), je uživatel informován chybovou hláškou. Rovněž na konci generování je informován o výsledku.

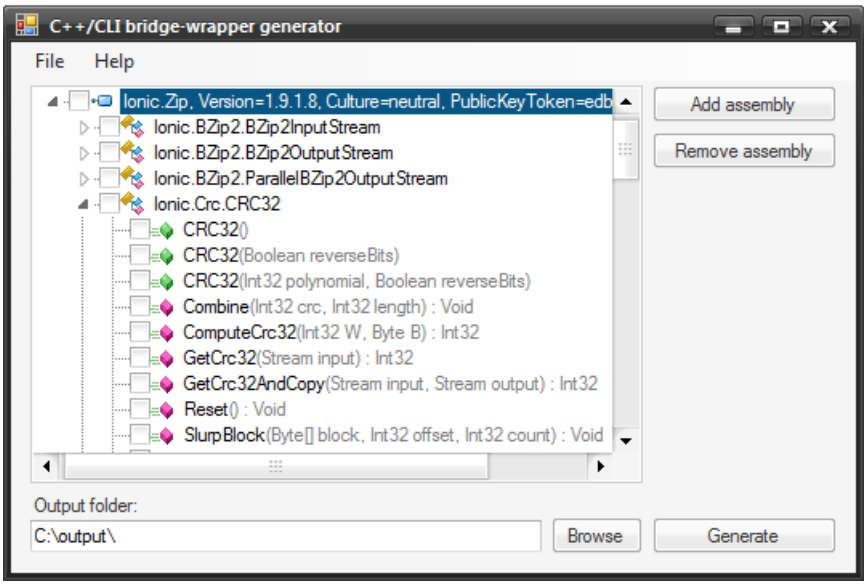
Součástí menu *Help* je položka *About and usage*, po jejímž rozkliknutí se uživateli objeví okno obsahující pár tipů k používání aplikace (obrázek A3). Dále obsahuje kontakt na autora a odkaz na git repozitář s celým projektem.

Typický postup práce s nástrojem je následující:

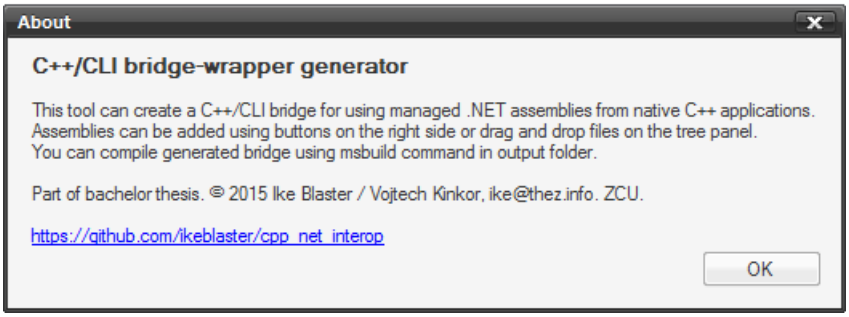
1) Spuštění nástroje.
2) Přidání .NET assembly pomocí tlačítka nebo přetažením na okno aplikace.
3) Výběr požadovaných součástí pomocí zaškrtávacích polí.
4) Stisknutí tlačítka „Generovat“.



Obrázek A1 – Okno aplikace po spuštění.



Obrázek A2 – Okno aplikace s přidanou assembly DotNetZip.



Obrázek A3 – Okno s informacemi o aplikaci.

Práce s vygenerovaným wrapperem

Vygenerovaný wrapper nalezneme ve zvolené výstupní složce. Uvnitř této složky se nacházejí veškeré soubory potřebné ke kompilaci. Pro zkompilování je nutné mít nainstalováno IDE Microsoft Visual Studio 2010 nebo novější (funkčnost ověřena s verzemi 2010 a 2013).

Wrapper je možné zkompilovat pomocí projektového souboru s příponou `.vcxproj` dvojím způsobem:

1. Pomocí IDE Visual Studio – importováním tohoto souboru získáme projekt, který lze jednoduše zkompilovat příslušným tlačítkem.
2. Pomocí nástroje `msbuild` – obvykle stačí spustit tento nástroj bez argumentů ve složce s projektovým souborem.

Po zkompilování vznikne složka `Debug`, případně `Release` (podle zvolené konfigurace), uvnitř které se nachází složka `bin` s vygenerovaným wrapperem.

Použití wrapperu v C++ aplikaci

Použití vygenerovaného wrapperu v C++ aplikaci je jednoduché a spočívá v následujících krocích (v příkladech použita knihovna DotNetZip):

1) Přidání wrapperu mezi linkované knihovny – lze v IDE, nebo např. touto direktivou přidanou na začátek kódu aplikace:

```
#pragma comment(lib, "Wrapper")
```

2) Přidání direktivy `include` pro hlavičkové soubory s definicemi wrapperu, například:

```
#include "Wrapper_Ionic_Zip_ZipFile.h"
```

3) Používání mostu ve funkcích C++ aplikace:

```
using namespace Wrapper::Ionic::Zip;
cout << ZipFile::IsZipFile("somefile") ? "yes" : "no";
```

B Obsah přiloženého média

Součástí práce je přiložené paměťové médium (DVD) obsahující tyto adresáře a soubory:

- `CppCliBridgeGenerator/` – adresář obsahující projekt nástroje pro IDE Microsoft Visual Studio 2010.
- `CppCliBridgeGenerator_Examples/` – adresář obsahující připravené příklady použitelné pro otestování funkčnosti nástroje.
- `CppCliBridgeGenerator_PerformanceTests/` – adresář obsahující nástroje v podobě použité pro testování funkčnosti v kapitole 7.
- `CppCliBridgeGenerator_Release/` – adresář obsahující spouštěcí soubor nástroje (tj. zkompilovaný projekt z první zmíněné složky).
- `InteropMethods/` – adresář obsahující příklady používané v kapitole 3.
- `Kinkor_A12B0082P_BP.pdf` – text této práce ve formátu PDF.
- `readme.txt` – textový soubor obsahující popis struktury DVD.

Obsah přiloženého média (včetně aktuální verze nástroje) je možné najít též v repozitáři projektu v rámci služby GitHub na adrese:
`https://github.com/ikeblaster/cpp_net_interop/`