Západočeská univerzita v Plzni

Fakulta aplikovaných věd

Katedra informatiky a výpočetní techniky

Bakalářská práce

Využívání .NET assembly  
z neřízeného C++

Plzeň, 2015 Vojtěch Kinkor

Zadání

1. Prozkoumejte stávající možnosti využívání funkcionality z .NET assembly v neřízeném C++ kódu na různých platformách (Windows, Linux, Mac, aj.), přičemž se zaměřte zejména na přístup prostřednictvím technologie COM a prostřednictvím hostování jádra platformy .NET v neřízeném kódu. Proveďte srovnání stávajících možností.
2. Po dohodě s vedoucím práce navrhněte a naimplementujte nástroj, který pro zadanou .NET třídu (resp. assembly), automaticky vytvoří C++ wrapper sloužící pro transparentní volání řízeného kódu zadané .NET třídy z neřízeného C++ kódu.
3. Funkčnost programového vybavení ověřte na několika jednoduchých interop (C++ -> .NET) příkladech, které si sám vytvoříte, a které budou ověřovat zejména korektní chování přenosu řetězců z neřízeného C++ kódu do .NET řízeného kódu a obráceně, korektní chování přenosu jedno i více rozměrných polí nativních datových typů, struktur nativních typů, apod. Proveďte zhodnocení časové režie volání.
4. Vyzkoušejte nasazení řešení na reálný problém a proveďte zhodnocení tohoto nasazení.

Prohlášení

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

V Plzni dne 5. 5. 2015

Vojtěch Kinkor

Abstrakt

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

Abstract

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

Obsah

[1 Úvod 1](#_Toc418614833)

[2 Základní informace 2](#_Toc418614834)

[2.1 Platforma .NET 2](#_Toc418614835)

[2.2 Neřízený kód 3](#_Toc418614836)

[2.3 Propojení neřízeného a řízeného kódu 3](#_Toc418614837)

[2.4 Platforma Mono 4](#_Toc418614838)

[3 Možnosti propojení .NET assembly a neřízeného C++ 6](#_Toc418614839)

[3.1 C++ Interop / IJW 7](#_Toc418614840)

[3.2 Platform Invoke 7](#_Toc418614841)

[3.2.1 Vytvoření C++ .NET assembly obsahující unmanaged API 7](#_Toc418614842)

[3.2.2 Vytvoření C++/CLI mostu k .NET assembly 9](#_Toc418614843)

[3.3 Přístup pomocí COM API 10](#_Toc418614844)

[3.4 Hostování CLR 12](#_Toc418614845)

[3.4.1 Hostování Microsoft .NET CLR 12](#_Toc418614846)

[3.4.2 Využití platformy Mono a „embedování kódu“ 13](#_Toc418614847)

[4 Srovnání možností 14](#_Toc418614848)

[4.1 Kritické oblasti 14](#_Toc418614849)

[4.1.1 Převod datových typů – marshalling 14](#_Toc418614850)

[4.1.2 Obsluha výjimek 14](#_Toc418614851)

[4.1.3 „Callback“ funkce 15](#_Toc418614852)

[4.2 Shrnutí výhod a nevýhod jednotlivých možností 15](#_Toc418614853)

[4.2.1 C++ Interop / IJW 15](#_Toc418614854)

[4.2.2 Platform Invoke 15](#_Toc418614855)

[4.2.3 Přístup pomocí COM API 16](#_Toc418614856)

[4.2.4 Hostování CLR 16](#_Toc418614857)

[5 Závěr 21](#_Toc418614858)

[6 Reference 22](#_Toc418614859)

# Úvod

Prozkoumejte stávající možnosti využívání funkcionality z .NET assembly v neřízeném C++ kódu na různých platformách (Windows, Linux, Mac, aj.), přičemž se zaměřte zejména na přístup prostřednictvím technologie COM a prostřednictvím hostování jádra platformy .NET v neřízeném kódu. Proveďte srovnání stávajících možností.

# Základní informace

## Platforma .NET

.NET je moderní softwarová platforma vyvíjená mnoha firmami[[1]](#footnote-1) a její součásti byly standardizovány společnostmi ECMA a později i ISO. Platforma slouží pro vývoj a běh aplikací. Jejím hlavním představitelem a první implementací je Microsoft .NET Framework určený pro osobní počítače se systémem Microsoft Windows. Vývojářům nabízí širokou knihovnu funkcí pokrývající mnoho oblastí – například grafické uživatelská rozhraní, řízení přístupu k datům, přístup k databázím, síťovou komunikace, aj. [[1]](#bib1)

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.  [[1]](#bib1)

Sestavu jednoho či více CIL kódu (spolu s tzv. *manifestem* – metadaty, která danou sestavu popisují) lze označit jako **.NET assembly**. [[2]](#bib2)



Obrázek 1 – Překlad aplikace ze zdrojového do nativního kódu. (zdroj:  [[3]](#bib3))

## 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.

## Propojení neřízeného a řízeného kódu

.NET Framework je v aktuální době velmi oblíbenou a prosazovanou možností pro tvorbu aplikací na platformě Microsoft Windows a díky svým možnostem zjednodušuje a zrychluje vývoj. 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 je často nerealizovatelné nebo může být velmi nákladné. Pokud je potřeba takový software rozšířit, je možné pokusit se jej propojit s komponentami vytvořenými jako .NET assembly.



Obrázek 2 – kontext řízených a neřízených aplikací. (zdroj:  [[4]](#bib4))

## Platforma Mono

Mono[[2]](#footnote-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 [[5]](#bib5).

Aktuální verze (3.10.0 k lednu 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,…) [[6]](#bib6).

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]](#footnote-3) (vývoj zejména mobilních aplikací) a Unity[[4]](#footnote-4) (herní engine). [[7]](#bib7)

Mono je binárně kompatibilní s .NET Frameworkem, z čehož plyne možnost spouštět programy zkompilované pomocí Microsoft i Mono C# kompilátoru na obou platformách. [[5]](#bib5)

# Možnosti propojení .NET assembly a neřízeného C++

Možnosti propojení lze rozdělit do 4 kategorií, lišících se přístupem a požadavky [8]:

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* [[9]](#bib9)*.* Jde o princip zveřejnění statických 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).

Všechny možnosti lze využít pro volání řízeného kódu z neřízené aplikace (tedy typicky využívání .NET knihovny z klientské C/C++ aplikace).

První 3 možnosti lze využít i opačně, pro volání neřízeného kódu z řízené aplikace (využívání nativních knihoven z klientské .NET aplikace).

Pro potřeby práce se zaměřím pouze na první variantu propojení. V příkladech budu vycházet z následující třídy 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 **}**

## 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*. [[9]](#bib9)

Toho lze využít, pokud je 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 může obsahovat jak řízený, tak neřízený kód. Poté je možné v takové aplikaci přímo volat komponenty z připojených .NET assemblies (viz ukázka 1). [[10]](#bib10)

Toto řešení je výhodné, pokud je možné klientskou aplikaci 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. [[9]](#bib9)

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.

## 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. [[9]](#bib9)

### Vytvoření C++ .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 postupy [[11]](#bib11). Častější způsob spočívá v přidání do původní řízené knihovny označení dllexport 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 vytvořit obalující třídy a funkce, které budou sloužit jako přemostění[[5]](#footnote-5). V klientské C++ aplikaci je pak nutné vytvořit deklarace exportovaných funkcí s označením dllimport.

1 #ifdef \_SERVER // při kompilaci knihovny

2 #define \_LNK \_\_declspec(dllexport)

3 #else // při kompilaci klientské aplikace

4 #define \_LNK \_\_declspec(dllimport)

5 #endif

6

7 class \_LNK YahooAPI **{**

8public**:**

9 double GetBid**(**const char**\*** symbol**) {** ... **}**

10 double GetAsk**(**const char**\*** symbol**) {** ... **}**

11 ...

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

1 #pragma comment(lib, "YahooAPI")

2 #include "YahooAPI.h"

3 ...

4 int main**()** **{**

5 YahooAPI **\***api **=** **new** YahooAPI**();**

6 cout **<<** YahooAPI**::**test**()** **<<** endl **<<** endl**;**

7 cout **<<** "Ask: " **<<** api**->**GetAsk**(**"MSFT"**)** **<<** endl**;**

8 cout **<<** "Bid: " **<<** api**->**GetBid**(**"MSFT"**)** **<<** endl**;**

9 ...

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 jednotnost 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 [[12]](#bib12), [[13]](#bib13), [[14]](#bib14). Nejedná se však o oficiálně popsaný a podporovaný postup.

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

Jedná se o speciální případ předchozí možnosti. Spočívá ve vytvoření knihovny, která bude sloužit pouze jako most či obalující mezivrstva (*wrapper*) mezi neřízenou C++ aplikací a řízenou .NET assembly. Musí být vytvořena v 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 **{**

10private**:**

11YahooAPIManagedWrapper\* \_private**;**

12public**:**

13 double GetBid**(**const char**\*** symbol**);**

14 double GetAsk**(**const char**\*** symbol**);**

15 ...

Ukázka 4 – Společný hlavičkový soubor YahooAPIWrapper.h pro most (wrapper) 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 **}**

9double YahooAPIWrapper**::**GetBid**(**const char**\*** symbol**)** **{**

10 **return** \_private**->**api**->**GetBid**(**gcnew System**::**String**(**symbol**));**

11 **}**

12 ...

Ukázka 5 – Soubor YahooAPIWrapper.cpp sloužící jako 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). Poměrně nevýznamnou nevýhodou je nutnost distribuovat s aplikací další knihovnu.

## 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. [[15]](#bib15)

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. [[9]](#bib9)

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).  [[16]](#bib16)

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 [[17]](#bib17).

1 **[**ComVisible**(true),** ClassInterface**(**ClassInterfaceType**.**AutoDual**)]**

2 **public** class YahooAPI **{**

3 ...

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

1 #import <mscorlib.tlb> raw\_interfaces\_only

2 #import "YahooAPI.tlb"

3 ...

4 int main**()** **{**

5 CoInitialize**(**0**);**

6

7 YahooAPI**::**\_YahooAPIPtr api**(**\_\_uuidof**(**YahooAPI**::**YahooAPI**));**

8 cout **<<** api**->**test**()** **<<** endl**;**

9 cout **<<** "Ask: " **<<** api**->**GetAsk**(**"MSFT"**)** **<<** endl**;**

10 cout **<<** "Bid: " **<<** api**->**GetBid**(**"MSFT"**)** **<<** endl**;**

11

12 CoUninitialize**();**

13 ...

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

## Hostování CLR

### 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*). [[18]](#bib18) 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. [[19]](#bib19)

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.

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, viz i ukázka 8. Vývoj komplikuje i nepříliš rozsáhlá dokumentace a omezené možnosti ladění kódu.

### 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“ [[20]](#bib20). 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.

# Srovnání možností

Zmíněné možnosti lze porovnávat z hlediska různých kritérií, ale je doporučováno vycházet z možností klientské aplikace a řízené knihovny. Zajímavým kritériem může být rychlost jednotlivých řešení. Pro základní výběr poskytuje Microsoft na stránkách MSDN následující rozhodovací strom:

**Write C++ wrapper and**

**expose as flat API**

**Write COM friendly managed**

**wrapper classes around it and**

**expose through COM interop**

**Recompile C++ client code with**

**new C++ compiler and access**

**managed API directly**

**Write managed API**

**using Design Guidelines and mark it as COM invisible**

**YES**

**NO**

**COM**

**FlatAPI**

**YES**

**Unmanaged**

**clients**

**important**

**Client is**

**C++ and**

**can be**

**recompiled**

**Type of**

**unmanaged**

**client**

**Done**

**NO**

Obrázek 3 – Rozhodovací strom pro výběr technologie. (zdroj: [[9]](#bib9))

## Kritické oblasti

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

### 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).

### 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 žádná taková možnost není, lze opět odchytit pouze obecnou výjimku bez jakýchkoliv podrobností.

### „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 .NET knihovny, nebo může .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.

## Shrnutí výhod a nevýhod jednotlivých možností

### C++ Interop / IJW

|  |  |
| --- | --- |
| + Jednoduché použití  + Snadné rozšíření o další funkcionalitu | – Nutné dodržet pravidla C++/CLI  – Nelze vždy použít – nutné překompilovat aplikaci jako C++/CLI |

### Platform Invoke

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

### Přístup pomocí COM API

|  |  |
| --- | --- |
| + Automatické generování obalujících struktur  + Univerzálnost | – Nutná registrace COM objektu v systému nebo použití Registration-Free technologie  – Vyšší režie |

### Hostování CLR

|  |  |
| --- | --- |
| + Přímá kontrola nad assembly  + Multiplatformní (Mono) | – Komplikované použití |

## 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. Přístup pomocí COM API se zdá být na první pohled jako nejjednodušší možnost, avšak tato možnost je vykoupena vyšší režií a omezenou použitelností. Ze zbývajících možností se jeví jako ideální kompromis možnost *Platform Invoke*, konkrétně ve variantě používající C++/CLI most.

# Realizace propojení pomocí C++/CLI mostu

Základní podoba tohoto propojení byla již naznačena v kapitole [3.2.2](#_Vytvoření_C++/CLI_mostu). Cílem je vytvořit mezivrstvu mezi neřízenou C++ aplikací a řízenou .NET assembly (typicky nějakou knihovnou funkcí). 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).

## Struktura .NET assembly

.NET assembly si lze představit jako seskupení různých částí, které jsou napsané pomocí nějakého 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 vlastnosti (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. *namespace* (pojmenovaný prostor tříd a dalších objektů), jedná se však především o kosmetické seskupování, který si lze představit jako prefix všech názvů (viz obrázek 4). Assembly v sobě nesou metadata umožňující snadné zobrazení informací o jednotlivých součástech (viz obrázek 5).

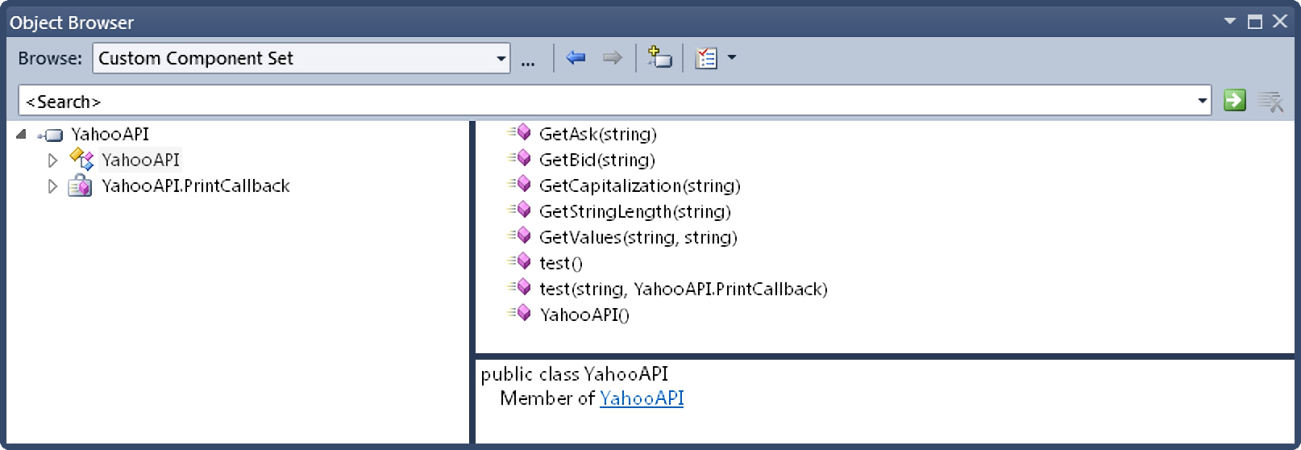
**System** namespace

**Windows** namespace

**Forms** namespace

**Button** class

Obrázek 4 – umístění třídy Button v rámci vnořených namespace System.Windows.Forms.  
Lze chápat jako třídu s názvem System.Windows.Forms.Button.



Obrázek 5 – Testovací třída YahooAPI zobrazená pomocí nástroje z Visual Studio 2010.

## Návrh struktury C++/CLI mostu

Cílem C++/CLI mostu je transparentně zpřístupnit jednotlivé součásti .NET 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 jeden objekt jako parametr metody jiného objektu), atd.

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

Výsledkem C++/CLI mostu po zkompilování je dynamická linkovaná knihovna (soubor s příponou .dll), která bude klientskou aplikací načítána. Jak již bylo zmíněno i v ukázkách propojení, všechny třídy a funkce, které jsou v tomto mostu zveřejněné pro „export“, musí být deklarovány pouze pomocí jazyka C++.

V následujících kapitolách je popsán návrh mostu z hlediska kódu.

### Rozdělení assembly

Každý prvek assembly (tj. například třída), ke kterému chceme vytvořit C++/CLI most, bude zastoupen vlastní sadou souborů. Jazyk C++/CLI, stejně jako C++, používá a hlavičkové (.h) zdrojové (.cpp) soubory. Vždy musí být vytvořen minimálně hlavičkový soubor, který bude obsahovat deklarace prvků exportovaných do DLL. Tento soubor lze připravit tak, aby byl použitelný jak pro kompilaci C++/CLI mostu, tak jako deklarace načítané knihovny pro klientskou aplikaci.

### Namespace

Jazyk C++ podporuje namespace ve stejné podobě, jakou používá i .NET. Lze tedy snadno vytvořit shodnou strukturu.

### Kolize názvů

Při tvorbě mostu mohou nastat kolize mezi názvy uvnitř původní .NET assembly a právě tvořeného mostu. Pro zamezení je možné použít několik postupů. Jedním z nich je přidání prefixů či postfixů k názvům všech prvků (řízená třída YahooAPI by byla v mostu zastoupena například pomocí třídy YahooAPIWrapper), přičemž není nutné měnit strukturu namespace.

Alternativou je právě změnění namespace. Je možné přejmenovat původní, případně přidat k názvu prefix/postfix. Jinou možností je vytvoření nové namespace, která bude celý most obalovat (řízená třída YahooAPI by tedy byla v mostu například pod názvem Wrapper::YahooAPI). 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.

### Třídy

Most pro třídu řízené .NET assembly se skládá z více částí a je rozložen do tří souborů.

Hlavní hlavičkový soubor obsahuje tzv. dopřednou deklaraci pomocné třídy, která uchovává odkaz na instanci řízeného objektu, a dále kompletní deklaraci exportované obalovací třídy (tj. deklaraci všech prvků, které budou použitelné v klientské aplikaci). Součástí obalovací třídy musí být odkaz na již zmíněnou pomocnou třídu. Jednou z metod obalovací třídy je konstruktor s parametrem této pomocné třídy – slouží pro vytvoření mostu v případě, že je řízený objekt návratovou hodnotou nebo parametrem nějaké další metody (ať už této, nebo i jiné třídy). Viz ukázka 10.

Dalším souborem je hlavičkový soubor obsahující právě definici pomocné třídy. Jejím jediným prvkem 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; // deklarace pomocné třídy

11

12 class \_LNK YahooAPI { // exportovaná třída

13 public:

14 YahooAPI\_IL\* \_\_IL;

15

16 YahooAPI();

17 YahooAPI(YahooAPI\_IL\* IL);

18 ~YahooAPI();

19

20 double GetAsk(std::wstring symbol);

21 ...

22 **};**

23

24  **}**

Ukázka 10 – hlavní hlavičkový soubor pro most třídy.

1 #pragma once

2 #pragma managed

3 #include <msclr\auto\_gcroot.h>

4

5 #using "YahooAPI.dll" // import řízené assembly

6

7 **namespace** Wrapper **{**

8

9 class YahooAPI\_IL **{**

10 public**:**

11 msclr**::**auto\_gcroot**<::**YahooAPI**^>** \_\_Impl**;** // odkaz na řízený objekt

12 **};**

13

14 **}**

Ukázka 11 – hlavičkový soubor s pomocnou třídou uchovávající odkaz na instanci řízené třídy.

Posledním souborem mostu třídy je zdrojový soubor .cpp, který obsahuje definice metod exportované třídy. Součástí konstruktorů je vytvoření instance pomocné třídy a tedy i vytvoření řízeného objektu. Zároveň je zde i destruktor, který naopak instanci pomocné třídy z paměti odstraní. O odstranění objektu se postará garbage collector navázaný na třídu auto\_gcroot.

1 #pragma once

2 #pragma managed

3 #include "marshaler\_ext.h"

4

5 #define \_LNK \_\_declspec(dllexport)

6 #include "Wrapper\_YahooAPI\_IL.h"

7 #include "Wrapper\_PrintCallback\_IL.h"

8

9 **namespace** Wrapper **{**

10

11 YahooAPI**::**YahooAPI**()** **{**

12 \_\_IL **=** **new** YahooAPI\_IL**;**

13 \_\_IL**->**\_\_Impl **=** gcnew **::**YahooAPI**();** // instance řízené knihovny

14 **}**

15

16 YahooAPI**::**YahooAPI**(**YahooAPI\_IL**\*** IL**)** **{**

17 \_\_IL **=** IL**;**

18 **}**

19

20 YahooAPI**::~**YahooAPI**()** **{**

21 **delete** \_\_IL**;**

22 **}**

23

24 double YahooAPI**::**GetAsk**(**std**::**wstring symbol**)** **{**

25 **::**System**::**String**^** \_\_Param\_symbol **=**

\_marshal\_as**<::**System**::**String**^>(**symbol**);**

26 **::**System**::**Double \_\_ReturnVal **=** \_\_IL**->**\_\_Impl**->**GetAsk**(**\_\_Param\_symbol**);**

27 **return** \_\_ReturnVal**;**

28 **}**

29 ...

30 **}**

Ukázka 12 – definice exportované třídy.

Uvnitř každé metody je typicky potřeba vykonat 2 až 3 kroky:

1. Převést parametry metody na správný datový typ.
2. Zavolat příslušnou metodu řízené třídy.
3. Převést návratovou hodnotu na správný datový typ a vrátit ji (pokud není metoda bez návratové hodnoty).

Převodu datových typů se věnuje kapitola 5.2.6. Pro zavolání řízené metody se přistupuje k řízenému objektu skrze pomocnou třídu.

<gettery/settery/fields ....>

### Struktury

Struktury se chovají jako hodnotové datové typy, tj. přiřazením do proměnné se obsah struktury zkopíruje. Zároveň však struktury v .NET jazycích mohou obsahovat určitou logiku a funkce. To je při vytváření mostu pomocí struktury v C++ poměrně obtížné zajistit. Dalším problémem může být správné uvolňování paměti řízených instancí. Jako možná alternativa se nabízí použití stejného postupu jako při vytváření mostů tříd, pouze s tím rozdílem, že v pomocné třídě bude tentokrát uložen odkaz na instanci struktury bez třídy auto\_gcroot. Bohužel se tímto postupem vytrácí možnost snadno kopírovat struktury mezi proměnnými.

### Výčtové typy

Jedná se o další 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. Hlavičkový soubor bude tedy obsahovat pouze výčtový typ zapouzdřený v odpovídající namespace.

1 **namespace** Utilities **{**

2 enum LineEnding **{**

3 None **=** 0**,**

4 DOS **=** 10**,**

5 Windows **=** 20**,**

6 Unix **=** 30

7 **};**

8 **}**

Ukázka 13 – „Most“ pro výčtový typ.

Popsat strukturu C++/CLI mostu z pohledu třídy – namespace, ... že je to +- stejné.

Popsat, co je potřeba za náležitosti (instance objektu v obalující třídě)

Vypsat rozdělení do souborů podle prvků (tříd/enumů/struktur), proč jsou struktury jako třídy

V samostatných nadpisech – popsat strukturu jednotlivých souborů, jako poslední .cpp soubor

Struktura jednotlivých metod podle druhů – volání metod/ctorů, property/fields

Marshalling

Rozšiřování marshalleru

Marshaling – stringy a generické kolekce

Generování projektů (sem nebo k další kapitole?)

# Nástroj pro generování C++/CLI mostů

Popsat obecně nástroj, v čem je napsán, jeho programovou strukturu

Další rozšiřitelnost

Možnost procházení assembly v C#

~~Uživatelská příručka~~ (v příloze)

Omezení

# Testování funkčnosti

Popsat/ukázat? funkčnost s YahooAPI

Popsat funkčnost s třídou Arrays

Nasazení na reálný problém, Ionic.Zip, problém s generickými třídami (nebo vyřešit?)

Zhodnocení funkčnosti řešení a nástroje

# Závěr

S čím jsem se seznámil, co jsem prozkoumal, jak jsem to použil při tvorbě nástroje, jestli je nástroj vůbec k něčemu, co je možné dále zlepšovat, zmínit multiplatformnost?

# Reference

​ [1] 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>

[2] 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>

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

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

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

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

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

[8] 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>

[9] 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>

[10] 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>

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

[12] 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>

[13] 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>

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

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

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

[17] 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>

[18] 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>

[19] 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>

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

1. Mj. Microsoft Corp., Hewlett-Packard, Intel Corp., IBM Corp., Novell Corp., Sun Microsystems [↑](#footnote-ref-1)
2. <http://www.mono-project.com/> [↑](#footnote-ref-2)
3. <http://xamarin.com/> [↑](#footnote-ref-3)
4. <http://unity3d.com/> [↑](#footnote-ref-4)
5. Je též možné vytvořit a exportovat zcela novou třídu pouze obalující původní funkčnost. [↑](#footnote-ref-5)