Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Andrej Bratož

Platforma .NET in interoperabilnost z Windows sistemom

diplomsko delo

univerzitetni študijski program prve stopnje
računalništvo in informatika

Mentor: doc. dr. Jurij Mihelič

Ljubljana, 2015

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Platforma .NET in interoperabilnost z Windows sistemom

Tematika naloge:

Platforma .NET je primarna platforma za razvijanje aplikacij na operacijskem sistemu Microsoft Windows. S svojo vsestranskostjo ter enostavnostjo uporabe je zasenčila predhodnike, kot so Component Object Model ter Microsoft Foundation Classes. Kljub zatonu predhodnikov pa platforma .NET še vedno omogoča komunikacijo z aplikacijami, ki so bile narejene s starejšimi tehnologijami.

V diplomski nalogi se osredotočite na opis delovanja platforme .NET ter njenih lastnosti. Opišite tudi načine, s katerimi platforma .NET komunicira s starejšimi tehnologijami v Windows sistemu. Nato implementirajte aplikacijo, ki bo na praktičnem primeru prikazala vsaj eno pomembno lastnost platforme .NET ter primer interoperabilnosti s starejšo tehnologijo.

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Bratož Andrej, z vpisno številko **63090047**, sem avtor diplomskega dela z naslovom:

Platforma .NET in interoperabilnost z Windows sistemom

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom doc. dr. Jurija Miheliča,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela
- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki "Dela FRI".

V Ljubljani, dne 23. junija 2015 Podpis avtorja:

Zahvaljujem se vsem, ki ste pripomogli k nastajanju tega diplomskega dela. Posebej pa se zahvaljujem mentorju doc. dr. Juriju Miheliču za ves vložen trud, za vse skrbne popravke ter za odlično strokovno usmerjanje in svetovanje.

Mojim staršem

Kazalo

Povzetek

Abstract

1 Uvod **1**

2 Platforma .NET **3**

2.1 Sklad .NET 3

2.2 Arhitektura platforme .NET 6

2.2.1 Common Intermediate Language 6

2.2.2 Common Language Runtime 12

2.2.3 Aplikacijske domene 15

2.3 Nadzorovani moduli in zbirke 15

2.3.1 Privatne zbirke 16

2.3.2 Deljene zbirke 16

3 Interoperabilnost z Win32 nenadzorovanimi knjižnicami **19**

3.1 Struktura Win32 DLL knjižnic 20

3.2 Pretvorba med nadzorovanimi in nenadzorovanimi tipi 22

3.3 Atribut DllImport 22

3.4 Klicanje funkcij, ki imajo strukture kot parametre 25

3.5 Dinamično alocirani tipi 26

4 Interoperabilnost med COM in .NET **29**

4.1 Opis tehnologije COM - Component Object Model 29

4.1.1 Komponente in njihove lastnosti 29
4.1.2 Komunikacija v tehnologiji COM 32
4.1.3 GUID 34
4.1.4 Vmesniki 34
4.2 Interoperabilnost med tehnologijama 40
4.2.1 RCW - Ovojnica za klicanje v času izvajanja 40

5 Uporaba opisanih tehnologij v praksi **45**
5.1 Opis problema 45
5.1.1 Pridobivanje metapodatkov 45
5.1.2 Izvoz podatkov v Microsoft Office Excel 51

6 Sklepne ugotovitve **57**
6.1 Teoretični del 57
6.2 Praktični del 57

A Dodatek - Programski vmesnik Excel Automation **59**

Povzetek

Platforma .NET predstavlja zelo pomemben element Windows sistema. Zaradi svoje vsestranskosti, bogatega sklada tehnologij ter velikega števila podprtih jezikov, je večinoma izrinila starejše tehnologije in tako postala primarna tehnologija za razvijanje aplikacij na operacijskem sistemu Microsoft Windows. V svojem diplomskem delu raziskujem arhitekturo platforme .NET, tehnologije, ki jih podpira, ter načine, s katerimi komunicira z aplikacijami, ki se nahajajo izven njenih meja, kot so Win32 aplikacije ter tehnologija Component Object Model.

Ključne besede: platforma .NET, .NET interoperabilnost, Component Object Model, Automation, Win32, Platform Invocation Services

Abstract

The .NET platform is a very important part of the Windows system. Due to its versatility, rich technology stack and a large number of supported languages, it has mostly overshadowed older technologies and became the primary technology for developing application on the Microsoft Windows operating system. In my thesis, I am researching the architecture of this platform, its technology stack and interoperability options with the applications that are outside its boundaries, such as Win32 applications and the Component Object Model.

Keywords: .NET platform, .NET interoperability, Component Object Model, Automation, Win32, Platform Invocation Services

Poglavje 1

Uvod

Operacijski sistem Microsoft Windows je trenutno najbolj razširjen operacijski sistem za namizne računalnike na svetu[1]. Zgodovina operacijskega sistema Windows se je začela že v osemdesetih letih prejšnjega stoletja, vendar pa je pravi zagon dobil šele po letu 1990. S prehodom na tehnologijo NT se je za Windows začelo novo obdobje, kjer je bila dominantna tehnologija Component Object Model (v nadaljevanju COM), skupaj z njim pa jezika Visual Basic ter C++. Konec prejšnjega tisočletja je Microsoft spoznal potrebo po nadzorovani platformi, ki bo združila vse tehnologije pod eno znamko. Razlog za to odločitev je bil tudi uspeh Jave. Tako se je leta 2002 pojavila platforma .NET, ki je močno spremenila razvijanje aplikacij v Windows sistemu ter zasenčila ostale tehnologije. Danes smo priča velikemu uspehu te platforme, ki se nenehno širi, izboljšuje ter v svoj sklad dobiva nove tehnologije. Kljub znatnim prednostim platforme .NET napram razvijanju s tehnologijo COM, se je podjetje Microsoft zavedalo, da so v desetih letih razvijanja pred platformo .NET nastale znatne količine kode, ki je podjetja niso bila pripravljena ponovno implementirati v novi tehnologiji. Razlogi za to so očitni - proces ponovne implementacije je zelo drag, težko je najti razvijalce programske opreme, ki bi to bili pripravljeni narediti, poleg tega pa se v kodo vnesejo novi hrošči. Microsoft je tako podjetjem ponudil rešitev v

[1] `https://en.wikipedia.org/wiki/Usage_share_of_operating_systems`

obliki relativno enostavne interoperabilnosti z nenadzorovanimi knjižnicami kot so Win32 dinamične knjižnice ter COM dinamične knjižnice (znane pod imenom COM strežniki). V svoji diplomski nalogi bom raziskal, kako deluje platforma .NET, kako komunicira z nenadzorovanimi knjižnicami ter kakšne so omejitve platforme same. Koda praktičnega dela diplomske naloge se nahaja na spletnem naslovu `https://github.com/andro47/Thesis`.

Poglavje 2

Platforma .NET

Platforma .NET je danes primarna platforma za razvijanje aplikacij na operacijskem sistemu Microsoft Windows. Podjetje Microsoft je začelo razvoj te platforme v poznih devetdesetih letih prejšnjega stoletja, prva stabilna verzija pa je bila na voljo leta 2002. V svoji osnovi je bila namenjena kot alternativa platformi Java na operacijskem sistemu Windows; slednji je podjetje Microsoft odreklo podporo v drugi polovici devetdesetih let prejšnjega stoletja. Razvoj platforme je prikazan v tabeli 2.1.

2.1 Sklad .NET

Sklad .NET so vse tehnologije, ki so del platforme .NET, vsaka nova različica pa je s seboj prinesla nekaj novega. Tehnologije znotraj platforme .NET opravljajo zelo različne naloge, od dela z podatkovnimi bazami do uporabniških vmesnikov, asinhronega programiranja itd. Spodaj so opisane najbolj pogosto uporabljene tehnologije iz sklada .NET:

Windows Forms (WinForms) WinForms je tehnologija za pisanje uporabniških vmesnikov. V svoji funkcionalnosti in načinu dela je zelo podobna svojemu predhodniku MFC (Microsoft Foundation Classes)[1],

[1]MFC je knjižnica, ki ovija dele Windows programskega vmesnika v C++ razrede.

Tabela 2.1: Pregled različic platforme .NET, izvajalnega okolja CLR ter verzij programskega paketa Visual Studio

Verzija platforme	Verzija CLR (glej podpoglavje 2.2.2)	Verzija okolja Visual Studio
1.0	1.0	Visual Studio .NET
1.1	1.1	2003
2.0	2.0	2005
3.0	2.0	/ (ni izšlo skupaj z VS)
3.5	2.0	2008
4	4	2010
4.5	4	2012
4.5.1	4	2013
4.5.2	4	/ (ni izšlo skupaj z VS)

vendar je za uporabo bolj enostavna. WinForms je .NET ovojnica okoli nenadzorovanih Windows komponent za uporabniške vmesnike, vendar za razliko od MFC (ter tudi originalnega Visual Basic-a) ponuja abstrakcijo na višjem nivoju ter je dogodkovno zasnovana [35].

ASP.NET ASP.NET je tehnologija za razvijanje spletnih aplikacij. V svoji osnovi je delovala na tehnologiji spletnih obrazcev (angl. "Web Forms"), ki so bili zelo podobni tehnologiji WinForms ter so omogočali zelo veliko stopnjo abstrakcije. Kljub relativni enostavnosti razvoja s spletnimi obrazci je imela velika stopnja abstrakcije tudi negativne učinke, saj je bilo zelo težko nadzorovati, kaj se dogaja na nižjih plasteh aplikacije, ki so bile skrite za to abstrakcijo. Kasneje je tehnologija ASP dobila še model MVC (Model - View - Controller), ki danes velja za primarni

Namenjena je predvsem delu z okni.

model za razvoj spletnih aplikacij v tehnologiji ASP.NET [39]. Model MVC omogoča jasno ločevanje med prednjim in zalednim delom spletne aplikacije.

ADO.NET ADO.NET je tehnologija, ki je namenjena dostopu do relacijskih podatkovnih baz, vendar pa lahko dostopa tudi do ne-relacijskih podatkovnih virov. Je močno predelan in izboljšan naslednik tehnologije ADO (ActiveX Data Objects), ki je temeljil na tehnologiji COM in je bil namenjen dostopu do OLE DB podatkovnih baz. Podatkovna struktura `DataSet` je jedro tehnologije ADO.NET [9, 34].

Windows Communication Foundation (WCF) Windows Communication Foundation je tehnologija za grajenje servisno orientiranih aplikacij. Namenjeno je asinhronemu pošiljanju sporočil iz enega servisa v drugega, sporočila pa so lahko v obliki enega znaka, XML ali pa toka binarnih podatkov. WCF se pojavlja kot naslednik tehnologije MTS oziroma Microsoft Transaction Server, ki je temeljila na tehnologiji COM [19].

Entity Framework (EF) Entity Framework je odprtokodni sistem za mapiranje med nekompatibilnimi podatkovnimi tipi. Je del tehnologije ADO.NET, ki omogoča kreiranje podatkovno orientiranih aplikacij. EF omogoča razvijalcem delo s podatki v obliki domensko specifičnih objektov in lastnosti. Omogoča, da izločimo večino kode za dostop do podatkov, kar je izboljšava glede na tehnologijo ADO.NET [6, 36].

Windows Presentation Foundation (WPF) Windows Presentation Foundation je najnovejši sistem za grajenje uporabniških vmesnikov na platformi Microsoft Windows. Za razliko od WinForms, WPF za vse komponente omogoča lastno oblikovanje ter učinke in sicer s pomočjo opisnega jezika XAML. WPF omogoča uporabo sofisticiranih 3D grafičnih elementov, saj za prikaz grafičnih elementov uporablja tehnologijo DirectX [20].

Windows Workflow Foundation (WF) Windows Workflow Foundation je knjižnica ter procesno izvajalno okolje (angl. "Workflow Engine"), ki omogoča uporabnikom načrtovanje aplikacije z dolgoročnim izvajanjem. Potek izvajanja se definira kot niz programskih korakov ali faz. WF je uporabljen tudi v Microsoftovem sistemu za grajenje in postavljanje aplikacij, imenovanem Team Foundation Server. Zadnja verzija WF je bila izdana skupaj z platformo .NET verzijo 3.0 [37].

(Parallel) Language Initegrated Query (P)LINQ je komponenta, ki jezikom na platformi .NET omogoča izvajanje poizvedb na podoben način kot jezik SQL. To pomeni zelo kompakten način iskanja podatkov po podatkovnih strukturah, kot so seznami oz. poljubne .NET kolekcije. Omogoča operacije, kot so `Select`, `Where`, `SelectMany`, `Aggregate`, `Join`... Parallel LINQ oz. PLINQ pa omogoča izvajanje teh operacij paralelno.

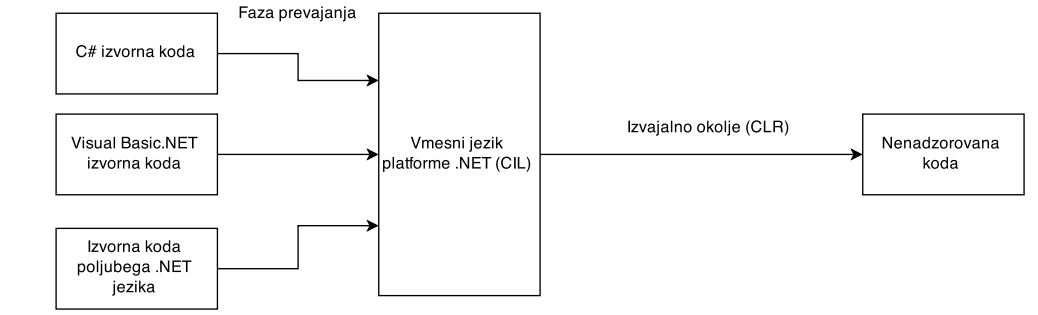
2.2 Arhitektura platforme .NET

Platforma .NET je kompleksno okolje, ki je sestavljeno iz več različnih delov. Filozofija te platforme je sinteza idej iz programskega jezika Java ter tehnologije COM. Osrednji komponenti platforme .NET sta Common Intermediate Language (CIL) ter Common Language Runtime (CLR). Program na platformi .NET potuje skozi več faz, ki so opisane na sliki 2.1.

2.2.1 Common Intermediate Language

Common Intermediate Langauge (v nadaljevanju CIL) je vmesni jezik platforme .NET. CIL temelji na skladu, enako kot javanski virtualni stroj. Nima nikakršnih ukazov za registre, prav tako pa se tudi izgubi sled za izvornim jezikom, iz katerega je vmesna koda nastala. To je razvidno iz izsekov 2.1, kjer imamo metodo implementirano v jeziku C#, ter 2.2, kjer imamo rezultat prevedbe v zbirnik .NET, ki je zadnja človeško berljiva stopnja pred

Slika 2.1: Faze od prevajanja do izvajanja na platformi .NET



prevedbo v vmesni jezik. Ena izmed najpomembnejših nalog vmesnega jezika je zagotavljanje interoperabilnosti med različnimi jeziki na platformi .NET. Lastnosti vmesnega jezika so:

- objektna usmerjenost
- razlikovanje med vrednostnimi in referenčnimi tipi
- močno tipiziranje
- uporaba izjem za upravljanje z napakami
- uporaba atributov

Izsek 2.1: Primer metode pred prevedbo v vmesno kodo

```
public class Foo
{
    public static int Add(int a, int b)
    {
            return a+b;
    }
}
```

Izsek 2.2: Primer vmesnega stanja kode na platformi .NET

```
.class public Foo
{
    .method public static int32 Add(int32,
        int32) cil managed
    {
      .maxstack 2
      ldarg.0 // nalozimo prvi argument
      ldarg.1 // nalozimo drugi argument
      add     // sestejemo;
      ret     // vrnemo rezultat;
    }
}
```

Interoperabilnost med jeziki

Ideja interoperabilnosti med jeziki se je v okolju Windows pojavila že s tehnologijo COM, na platformi .NET pa se je ta ideja še poglobila in poenostavila. V sistemu COM so komponente, napisane v različnih jezikih, med seboj komunicirale preko COM izvajalnega sistema, in ne direktno med seboj. Na platformi .NET je ta prepreka odpravljena, saj objekti, napisani v različnih jezikih, lahko direktno komunicirajo med seboj. To v praksi pomeni naslednje:

- razred, napisan v enem jeziku, lahko deduje po razredu, napisanem v drugem jeziku
- razred lahko vsebuje instanco nekega drugega razreda, ne glede na jezik, v katerem je razred napisan
- objekt lahko direktno kliče metode nekega drugega objekta, ne glede na jezik implementacije posameznega objekta

- objekti in njihove reference so lahko podane preko metod, ne glede na jezik
- programska okolja, kot je npr. Visual Studio, lahko z razhroščevalnikom koraka med klici metode, tudi če so metode definirane v različnih jezikih

Vrednostni in referenčni tipi

.NET vmesni jezik zelo natančno loči med vrednostnimi in referenčnimi tipi. Med vrednostne tipe spadajo tisti, kjer določena spremenljivka hrani dejansko vrednost, referenčni tipi pa hranijo le naslov, na katerem se nahajajo pripadajoči podatki.

Vrednosti tipi

Spremenljivke, ki direktno predstavljajo neko vrednost, spadajo v kategorijo vrednostih tipov (angl. "value types"). Vsak vrednostni tip na platformi .NET implicitno deduje po tipu `System.ValueType`. Vrednostni tipi se delijo v dve kategoriji, in sicer na enumeratorje ter strukture. Struktura je vrednostni tip, ki se uporablja za enkapsulacijo majhnih skupin sorodnih spremenljivk [14], enumeratorji pa so množice imenovanih konstant [15]. Vrednostni tipi se nadaljnje delijo, kot je prikazano spodaj. Vrednostni tipi ne morejo imeti vrednosti `null`, njihova primerjava pa je vedno glede na njihovo dejansko vrednost [10].

- numerični tipi
  - celoštevilski tipi, predznačeni in nepredznačeni [16] (glej tabelo 2.2)
  - tipi v plavajoči vejici [17] (glej tabelo 2.3)
  - decimalni tip [18] (glej tabelo 2.4)
- logični tip `bool` (vrednosti `true` ali `false`)
- poljubne uporabniško definirane strukture

Tabela 2.2: Celoštevilski tipi na platformi .NET

Tip	Razpon	Velikost
`sbyte`	-128 do 127	predznačeno 8-bitno število
`byte`	0 do 255	nepredznačeno 8-bitno število
`char`	U+0000 do U+FFFF	Unicode 16-bitni znak
`short`	-32768 do 32767	Predznačeno 16-bitno število
`ushort`	0 do 65535	Nepredznačeno 16-bitno število
`int`	-2147483648 do 2147483647	Predznačeno 32-bitno število
`uint`	0 do 4294967295	Nepredznačeno 32-bitno število
`long`	-9223372036854775808 do 9223372036854775807	Predznačeno 64-bitno število
`ulong`	0 do 18446744073709551615	Nepredznačeno 64-bitno število

Tabela 2.3: Tipi v plavajoči vejici na platformi .NET

Tip	Razpon	Natančnost
`float`	$\pm 1.5 * 10^{-45}$ do $\pm 3.4 * 10^{38}$	7 mest
`double`	$\pm 5.0 * 10^{-324}$ do $\pm 1.7 * 10^{308}$	15 mest

Tabela 2.4: Decimalni tip na platformi .NET

Tip	Razpon	Natančnost
`decimal`	$(-7.9 * 10^{28}$ do $7.9 * 10^{28})/(10^{0}$ do $10^{28})$	28 mest

Referenčni tipi

Referenčni tip vsebuje kazalec, ki kaže na lokacijo v pomnilniku, ki vsebuje dejanske podatke. Za njihovo inicializacijo, moramo uporabiti operator `new`. Tabele so vedno referenčni tipi, tudi če vsebujejo vrednostne elemente. Med referenčne tipe spada tudi .NET razred `System.String`, ter delegati[2].

[2]Delegat je podoben koncept, kot je kazalec na funkcijo v jezikih C ter C++, saj dinamično poveže klicatelja metode s ciljno metodo.

Močno tipiziranje

Ena izmed pomembnih lastnosti vmesnega jezika je tudi zahtevano močno tipiziranje. V praksi to pomeni, da nobena operacija ne more imeti rezultata, za katerega ne bi bilo mogoče nedvoumno določiti tipa [33, str. 8]. To je v nasprotju z jeziki kot so Visual Basic, kjer lahko operiramo s tipom `Variant`[3] ali pa C++, kjer lahko pretvarjamo med kazalci različnega tipa. Poleg jezikovne interoperabilnosti je močno tipiziranje tudi pogoj za delovanje mehanizmov, kot so:

- jezikovna interoperabilnost
- avtomatsko upravljanje s pomnilnikom
- varnost izvajanja
- aplikacijske domene

Common Type System

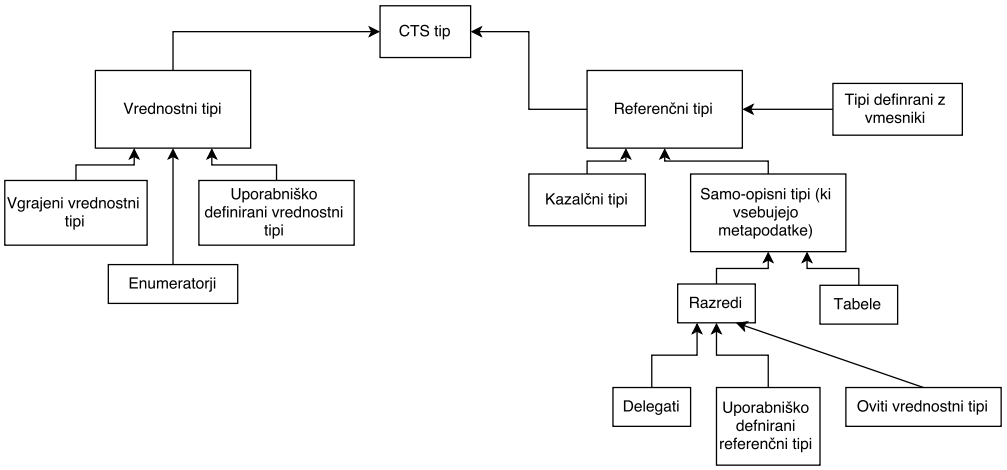
Common Type System definira vse tipe v .NET vmesnem jeziku, ter tako omogoča, da bodo vsi .NET jeziki na koncu imeli tipe, ki so skladni z tipi, definiranimi v izvajalnem okolju CLR. Kot primer, tipa `Integer` iz jezika Visual Basic ter `int` iz jezika C# bosta postala CTS tip `System.Int32` [33, str. 9]. Hierarhija tipov, ki jih določa CTS, je prikazan na sliki 2.2.

Common Language Specification

Common Language Specification (CLS), je dodaten mehanizem, ki zagotavlja interoperabilnost tipov na platformi .NET. Določa minimalno podmnožico standardov, ki jih morajo implementirati vsi prevajalniki, ki ciljajo na platformo .NET [26]. Večina prevajalnikov navadno ne implementira vseh zmožnosti vmesnega jezika CIL, saj jih jeziki načeloma ne potrebujejo, ampak zgolj njihovo podmnožico.

[3]Variant je poseben podatkovni tip, ki se je pojavil v tehnologiji COM. Lahko vsebuje poljubne podatke, razen nizov fiksne dolžine (glej 5.1.2)

Slika 2.2: Pregled CTS tipov



Atributi

Atributi so mehanizem, ki omogočajo vnos metapodatkov v nadzorovane zbirke (glej razdelek 2.3). Poleg že definiranih atributov na platformi .NET, lahko uporabnik definira tudi svoje [33, str. 13], in sicer na kateremkoli članu razreda, poljubni podmnožici članov razreda ali pa na razredu samem. Metapodatke, ki jih v nadzorovano zbirko vnesemo tekom prevajanja programa, lahko nato v času izvajanja pridobimo z mehanizmom, imenovanim odsevnost (angl, "reflection"). Primer uporabe atributov ter metapodatkov, ki jih ti vnesejo v nadzorovano zbirko, je pri testiranju enot programske opreme (angl. "unit tests"); vsaki metodi, ki jo želimo testirati, dodamo atribut `[TestMethod]`, platforma za testiranje pa v času izvajanja ta podatek prebere, in tako ve, katere metode so mišljene za testiranje [7].

2.2.2 Common Language Runtime

Jedro platforme .NET sestavlja komponenta imenovana CLR oziroma Common Language Runtime. Ta komponenta predstavlja najnižji nivo v njeni arhitekturi.

Vse lastnosti jezika se na poti do CLR-ja izgubijo in tako CLR nima nikakršne informacije o izvornem jeziku. V svoji osnovi platforma .NET podpira že kar nekaj jezikov, ki so vsi med seboj kompatibilni na nivoju CLR-ja. Nekaj bolj vidnih jezikov je C#, C++/CLI, Visual Basic .NET, F#, Iron Python ter Iron Ruby. Med poglavitne naloge CLR spadajo [38, 22]:

- upravljanje s pomnilnikom
- varno tipiziranje
- prevajanje vmesne kode v strojne ukaze
- upravljanje z izjemami
- uporaba delegatov namesto kazalcev na funkcije v namen varnega tipiziranja
- optimizacija izvajanja
- omogočanje konstruktov, kot so dedovanje, vmesniki ter nad-nalaganje (angl. "overloading")
- razširljivi tipi, ki jih poda knjižnica razredov
- omogočanje nitenja
- uporaba lastnih atributov

Upravljanje s pomnilnikom

Smetar (angl. "garbage collector") je na platformi .NET zadolžen za avtomatsko alokacijo ter sproščanje sredstev v aplikaciji. Vsakič, ko naredimo nov objekt, CLR zanj alocira pomnilnik na nadzorovani kopici. Ker prostora nimamo neomejeno, mora smetar odrabljene objekte pospraviti in sprostiti njihov prostor v pomnilniku [33, str. 10, 11]. Za čimbolj optimalno alociranje ter sproščanje objektov, si .NET smetar pomaga z metapodatki, ki jih prevajalniki dodajo v nadzorovane module. Kljub temu, da je upravljanje s

pomnilnikom na platformi .NET popolnoma avtomatski proces, ki ga nadzoruje CLR izvajalno okolje, je možno nadzor nad tem procesom prevzeti v aplikaciji ter ročno upravljati s smetarjem z pomočjo razreda `System.GC` [12].

Varno tipiziranje

Varno tipiziranje na platformi .NET zagotavlja, da koda dostopa le do pomnilniških lokacij, za katere ima dovoljenje. Med hipnim prevajanjem (angl. "Just in time compilation") JIT prevajalnik pregleda metapodatke ter vmesni jezik .NET z namenom verifikacije varnega tipiziranja [27]. Poleg pomnilniške definicije varnega tipiziranja, .NET upošteva tudi bolj splošno definicijo varnega tipiziranja, saj .NET prevajalniki preprečijo proces prevajanja, če pride do neskladja tipov. V primeru, da prevajalniki ne morejo zagotoviti varnega tipiziranja, kot je na primer uporaba tipov `dynamic`[4], potem varno tipiziranje zagotovi CLR.

Hipno ("Just in time") prevajanje

Hipno prevajanje je mehanizem, ki je pogosto uporabljen na platformah z nadzorovanimi jeziki (npr. Java). Omogoča, da se vmesna koda sproti prevaja v strojno kodo, ter tako dobi hitrost pravega stroja. Pomembna optimizacija tega procesa je le enkratno prevajanje, kar pomeni, da se ob klicu neke metode ta v strojno kodo prevede le prvič, nato pa se ta prevedba shrani in tako ob naslednjem klicu ni več potrebno te kode znova prevajati. S tem dosežemo hitrost klica, ki je enaka hitrosti pri nenadzorovanih aplikacijah [1, str. 11 – 13].

Upravljanje z izjemami

Platforma .NET ima zelo dobro razvit sistem za upravljanje z izjemami, ki je enostaven za uporabo. Izjeme se uporabljajo za kreiranje robustnih aplikacij,

[4] *dynamic* je posebni tip, ki lahko prevzame identiteto poljubnega tipa v času izvajanja (za razliko od tipa *object*, ki to omogoča v času prevajanja). Za bolj natančen opis glej 5.1.2

saj omogočajo konsistentno ter strukturirano ravnanje aplikacije ob morebitnih napakah. .NET ima zelo razvejano hierarhijo izjem, ki pa vse izhajajo iz tipa `System.Exception` [11]. Vsaka izjema vsebuje opis izvajalnega sklada, morebitno notranjo izjemo ter v primeru, če izjema izhaja iz komunikacije z COM sistemom, tudi ustrezne podatke o številki napake na COM strežniku.

2.2.3 Aplikacijske domene

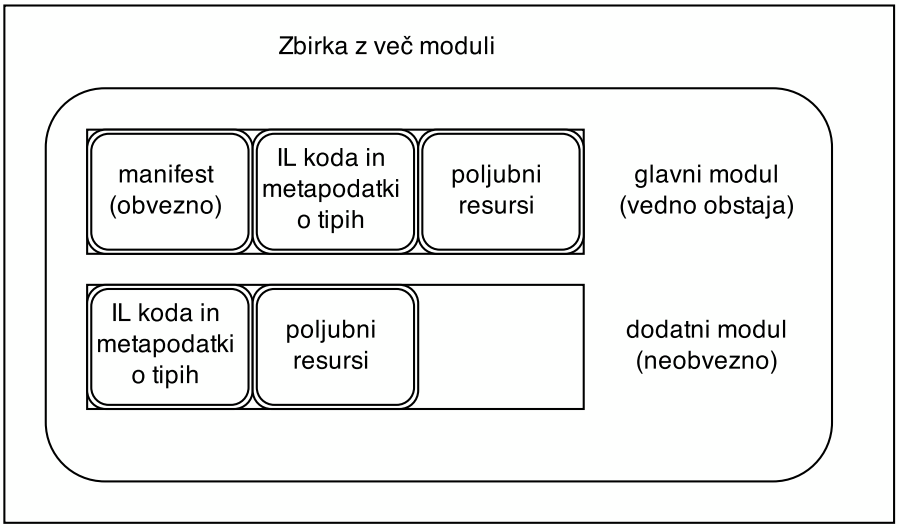
Aplikacijske domene so mogočen konstrukt na platformi .NET, ki je vidno zmanjšal kompleksnost ter povečal varnost za komunikacijo med aplikacijami. Tradicionalno so bile aplikacije ločene v različnih procesih, kar je pomenilo relativno veliko kompleksnost za medsebojno komuniciranje, ali pa so si proces delile. To je zmanjšalo varnost aplikacije in povečalo možnosti, da je napaka na eni aplikaciji zrušila cel sistem. Aplikacijske domene si še vedno delijo en skupen proces, kar omogoča majhno kompleksnost za medsebojno komuniciranje, vendar pa je proces razdeljen na različne aplikacijske domene. Vsaka domena pripada eni aplikaciji in vsaka nit izvajanja teče v ustrezni aplikacijski domeni [33, str. 12]. Kljub temu, da si aplikacije delijo proces, pa so zaprte v svoji aplikacijski domeni, kar pomeni, da ena aplikacija ne more onesnažiti ter poškodovati pomnilniške strukture druge aplikacije v istem procesu.

2.3 Nadzorovani moduli in zbirke

Zbirka (angl. "assembly") je samostojna enota, ki vsebuje prevedeno kodo, metapodatke ter resurse, če jih uporabnik definira. Vsaka zbirka je samozadostna v smislu, da vsebuje vse potrebne podatke [33, str. 14]; to pomeni, da ostale nadzorovane zbirke ali aplikacije lahko opravljajo klice v to zbirko, ne da bi bilo potrebno za informacije o njej pogledati v zunanje vire, kot na primer Windows register. Zbirka je sestavljena iz enega ali več modulov. Če ima zbirka več modulov, ima en modul vedno vlogo glavnega modula, ki vsebuje manifest – posebno XML datoteko. Ta je lahko samostojna ali pa

vgrajena v nadzorovano zbirko in vsebuje podatke o nadzorovani zbirki, ki so relevantni za operacijski sistem, kot na primer nivo delovanja aplikacije (navaden ali pa priviligiran) [4, str. 649]. Primer zbirke z več moduli vidimo na sliki 2.3.

Slika 2.3: Primer .NET zbirke z več kot enim modulom



2.3.1 Privatne zbirke

Privatne zbirke so zbirke, ki pridejo skupaj v kompletu z neko programsko opremo. Imajo garancijo in obenem omejitev, da jih bo lahko uporabljala le aplikacija, za katero je bila namenjena; da aplikacija zbirke lahko uporabi, se morajo nahajati v isti mapi ali podmapi kot program [33, str. 14].

2.3.2 Deljene zbirke

Deljene zbirke so namenjene uporabi kot skupne knjižnice in jih lahko uporabi poljubna aplikacija. V izogib trkom, ki jih lahko prinesejo različne deljene

zbirke, kot so imenski trki, ima platforma .NET poseben repozitorij, namenjen deljenim zbirkam, imenovan GAC oziroma Global Assembly Cache [33, str. 15]. .NET zbirke tja ne moremo enostavno prekopirati, ampak si moramo pomagati z posebnim orodjem, ki zbirko v GAC postavi in registrira.

Poglavje 3

Interoperabilnost z Win32 nenadzorovanimi knjižnicami

Platforma .NET bi bila zelo omejena, če ne bi imela možnosti posegati po knjižnicah izven svojega okolja. V ta namen so v platformo .NET vdelali mehanizem za klicanje funkcij iz nenadzorovanih knjižnic, znanimi pod imenom Win32 knjižnice.

Na platformi .NET je eden izmed načinov interakcije z nenadzorovano kodo preko tehnologije, imenovane Platform Invocation Services (na kratko P/Invoke), pri kateri iz neke nenadzorovane Win32 dinamične knjižnice (DLL - Dynamic Link Library) kličemo nenadzorovano funkcijo, ki jo nato v .NET uporabimo kot katerokoli drugo nadzorovano metodo.

Dinamične knjižnice so eden izmed temeljev operacijskega sistema Windows, kot, na primer, *kernel32.dll* in *user32.dll*. Operacijski sistem Windows je v svoji preteklosti imel veliko težav z dinamičnimi knjižnicami, oziroma z njihovim verzioniranjem[1] [41]. Ker operacijski sistem dinamično knjižnico naloži le enkrat, potem pa ostane v pomnilniku, dokler jo uporablja vsaj še en program, lahko prihrani nekaj pomnilnika. Vendar pa v starejših verzijah operacijski sistem Windows ni znal prepoznati različnih verzij ene knjižnice. Tako je verzija dinamične knjižnice A, ki je bila 1.5, v očeh OS bila enaka ver-

[1]Problem je znan kot DLL pekel (angl. "DLL hell")

ziji 1.6. Zaradi tega so se pojavljale napake in nedelovanje programov. Danes tega problema nimamo več, saj je Windows sposoben imeti v pomnilniku več različic iste knjižnice.

3.1 Struktura Win32 DLL knjižnic

Tako kot izvajalni programi, ki vsebujejo vstopno točko `WinMain`, dinamične knjižnice vsebujejo vstopno točko `DllMain`, vendar pa za razliko od vstopne točke `WinMain`, njena uporaba ni obvezna. Funkcija `DllMain` [24] je definirana v spodnjem izseku:

```
BOOL APIENTRY DllMain(HANDLE hModule,
DWORD ul_reason_for_call,
LPVOID lpReserved )
{
    switch ( ul_reason_for_call )
    {
            case DLL_PROCESS_ATTACH:
                    // ...
                    break;
            case DLL_THREAD_ATTACH:
                    // ...
                    break;
            case DLL_THREAD_DETACH:
                    // ...
                    break;
            case DLL_PROCESS_DETACH:
                    // ...
                    break;
    }
    return TRUE;
}
```

hModule Identifikacija DLL modula. Vrednost je začetni naslov DLLja v pomnilniku.

ul_reason_for_call Razlog, zakaj je bila vhodna točka poklicana.

lpReserved Če je `ul_reason_for_call` enak `DLL_PROCESS_ATTACH`, potem je `lpReserved` enako `NULL`, če je bila knjižnica naložena dinamično; v nasprotnem primeru karkoli razen `NULL`.
Če je `ul_reason_for_call` enak `DLL_PROCESS_DETACH`, je vrednost `lpReserved` enako `NULL`, če je bila klicana funkcija `FreeLibrary` ali pa je nalaganje DLLja spodletelo; karkoli razen `NULL`, če je proces v fazi terminacije.

DLL_PROCESS_ATTACH Eden izmed razlogov, zakaj je bila klicana funkcija `DllMain`. DLL je bil naložen v virtualni naslovni prostor trenutnega procesa zaradi začetka delovanja procesa, ali pa klica funkcije `LoadLibrary`.

DLL_THREAD_ATTACH Trenutni proces je ustvaril novo nit. Ko se to zgodi, sistem pokliče vstopne točke vseh DLLjev, ki so trenutno priključene na proces.

DLL_THREAD_DETACH Nit se je zaključila brez napak. Sistem pokliče vstopne točke vseh priklopljenih DLLjev procesa.

DLL_PROCESS_DETACH DLL se je v fazi odhoda iz virtualnega naslovnega prostora klicočega procesa, zaradi neuspešnega nalaganja, ali pa je števec referenc na knjižnico dosegel vrednost 0.

DLL datoteke omogočajo izvoz elementov, kot so funkcije, razredi ali enumeratorji. Elemente izvozimo z rezerviranim izrazom Visual C++ prevajalnika `__declspec(dllexport)`.

3.2 Pretvorba med nadzorovanimi in nenadzorovanimi tipi

Pri prehodu iz platforme .NET v nenadzorovano okolje je potrebno poskrbeti, da se tipi iz nenadzorovanega okolja ujemajo s tistimi iz platforme .NET. Težava je v tem, da nenadzorovani tipi nimajo vedno ekvivalenta pri nadzorovanih tipih, zato so potrebna pravila, ki te tipe pravilno mapirajo. Zaglavna datoteka `wtypes.h` definira tipe, ki se uporabljajo v okolju Win32.

Ko smo v interakciji z neko funkcijo iz nenadzorovanega okolja, je pomembno, da so .NET tipi pravilno definirani, sicer bo pretvorba (angl. "marshalling") v nenadzorovane tipe napačna in bo program najverjetneje terminiran zaradi dostopa do nedovoljenega dela pomnilnika. V tabeli 3.1 so prikazana pravila mapiranja med nenadzorovanimi ter nadzorovanimi tipi [5, str. 19].

3.3 Atribut DllImport

Atribut `DllImport` združuje delovanje funkcij `LoadLibrary` (nalaganje dinamične knjižnice v pomnilnik) ter `GetProcAddress` (lociranje naslova posamezne funkcije v pomnilniku) Win32 sistema [5, str. 25]. Definiran je v imenskem prostoru `System.Runtime.InteropServices`. Definicija tega atributa je prikazan v izseku 3.1.

Izsek 3.1: Definicija atributa DllImport

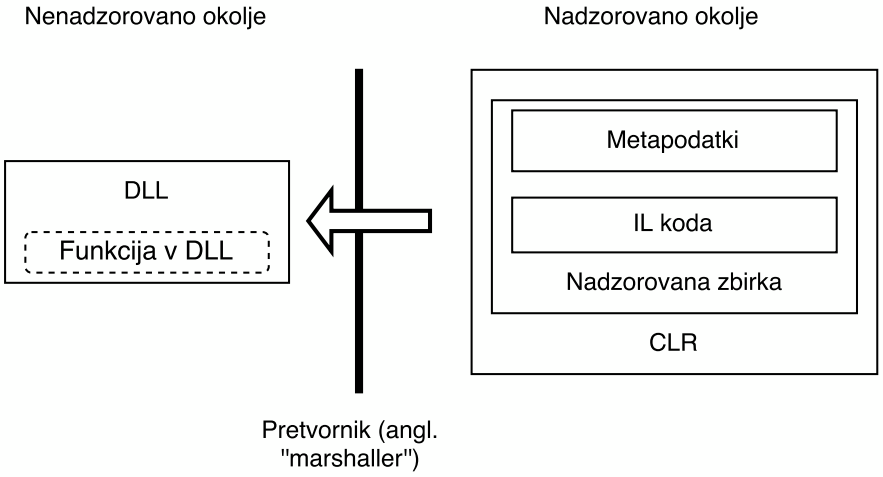
```
[AttributeUsage(AttributeTargets.Method)]
public sealed class DllImportAttribute :
   Attribute
{...}
```

`[AttributeUsage(AttributeTargets.Method)]` pomeni, da ta atribut lahko uporabljamo nad metodami. Namen tega atributa je posredovati ustrezne podatke. Ti so potrebni za klic metode, ki se nahaja v nenadzorovani

DLL knjižnici. Dejanski klic opravi platforma, in sicer v štirih korakih, ki so opisani spodaj [23], ter prikazani na sliki 3.1:

1. Lociranje DLL knjižnice, ki vsebuje željeno funkcijo
2. Nalaganje DLL knjižnice v pomnilnik (platforma implicitno kliče funkcijo `LoadLibrary`)
3. Iskanje naslova funkcije znotraj pomnilnika, nalaganje pomnilnika na sklad ter pretvarjanje med tipi. Prva dva koraka se opravita samo prvič, ko zahtevamo klic nenadzorovane funkcije, za vsak naslednji klic iste funkcije ali druge funkcije v istem DLLju to ni več potrebno.
4. Prenos nadzora na nadzorovano platformo

Slika 3.1: Delovanje P/Invoke



Atribut `DllImport` prav tako podpira več različnih parametrov, ki nam dajejo večji nadzor nad klicem nenadzorovane funkcije. Parametri so opisani spodaj [5, str. 28 – 30]:

CallingConvention S tem določimo način nenadzorovanega klica. Operacijski sistem Winwodws pozna več različnih načinov klicev, kot so

`cdecl` (klicatelj počisti sklad), `stdcall` (klicana funkcija počisti sklad), `thiscall` (privi parameter `this` je shranjen v registru ECX, ostali parametri so na skladu) ter `winapi` (privzeta konvencija na platofrmi).

CharSet S tem prametrom povemo, kateri nabor znakov uporabljamo, ter posledično, na kakšen način naj se argumenti pretvarjajo. Privzeta vrednost tega atributa je `CharSet.Ansi` pogosta alternativa pa je Unicode.

EntryPoint Ime ali ordinalno število funkcije, ki jo želimo klicati. Če želimo funkcijo izvoziti z ordinalnim številom, ne moremo uporabiti izraza `__declspec(dllexport)`, temveč moramo napisati `.def` datoteko, kjer vsaki izvoženi funkciji dodamo ordinalno število.

ExactSpelling S tem povemo, ali želimo, da se ime funkcije natanko ujema s tistim imenom, s katerim je funkcija definirana v DLL datoteki. Primer: Funkcija `MessageBox` je v resnici definirana kot dve različni funkciji `MessageBoxA` ter `MessageBoxW`. Ustrezna metoda je klicana glede na tip nizov, kjer `A` pomeni `Ansi`, `W` pa `Wide` oziroma Unicode.

PerserveSig Ta atribut je namenjen klicanju funkcij, ki vračajo kode za stanje tipa `HRESULT`, dejanska vrednost, ki jo vrne funkcija, pa je podana kot kazalec na enega izmed parametrov. Platforma lahko to pretvori v nenadzorovano definicijo funkcije, kjer se koda stanja ignorira, vrne pa se dejanska vrednost funkcije. Atribut nastavljamo glede na to, ali želimo, da se morebitne izjeme rokujejo na COM (preverjanje kode ter ukrepanje glede na vrednost) ali .NET način (z `try/catch` blokom).

SetLastError Z uporabo tega atributa se lahko dokopljemo do nenadzorovanih napak v Win32 sistemu preko `Marshal.GetLastWin32Error()`.

3.4 Klicanje funkcij, ki imajo strukture kot parametre

Za razliko od primitivnih tipov, kjer je pretvarjanje iz nenadzorovanega tipa v nadzorovanega in obratno zelo dobro definirano, pa je za podajanje struktur potrebno nekoliko več dela. Strukturo moramo znova definirati na nadzorovani platformi, in sicer v enakem vrstnem redu, kot je definirana v nenadzorovanem okolju. V arhitekturi x86 je običajno struktura definirana v pomnilniku zaporedno, ter ima n-bajtno poravnavo. V nadzorovanem okolju teh garancij nimamo, in je tako potrebno posebej povedati platformi, da zahtevamo specifično poravnavo. To naredimo tako, da na strukturi definiramo atribut `StructLayout` [5, str. 35]. Primer imamo podan v izseku 3.2.

Izsek 3.2: Primer uporabe atributa StructLayout na strukturi, kjer zahtevamo sekvenčno pomnilniško postavitev elementov

```
[StructLayout(LayoutKind.Sequential)]
public Point3D
{
    int x;
    int y;
    int z;
}
```

V primeru, da želimo sami nadzorovati postavitev strukture, lahko uporabimo `LayoutKind.Explicit`, kjer moramo za vsak element podati tudi odmik. Primer je podan v izseku 3.3.

Izsek 3.3: Primer strukture, kjer eksplicitno navajamo odmike

```
[StructLayout(LayoutKind.Explicit)]
public Point3D
{
    [FieldOffset(0)] short x;
    [FieldOffset(2)] int y;
```

```
        [FieldOffset(6)] int z;
    }
```

3.5 Dinamično alocirani tipi

`DllImport` nam omogoča, da iz nenadzorovanega okolja v nadzorovano okolje prenesemo tudi alocirane referenčne tipe. Platforma .NET ima definiran poseben razred za upravljanje z alociranimi nenadzorovanimi objekti, in sicer tip `IntPtr` [5, str. 39]. Ta razred služi kot nadzorovani nadomestek za kazalec v nenadzorovanem okolju. S tem objektom tako dobimo nadzor nad delom pomnilnika, kjer se element nahaja. Zaradi lastnosti nadzorovanega okolja pa tipa ne moremo dereferencirati ter ob koncu izpustiti iz pomnilnika, kot bi to storili v nenadzorovanem okolju. Zaradi tega je razred `IntPtr` opremljen z metodami, kot so `PtrToStructure`, `SizeOf`, `DestroyStructure` ter `FreeToCastMem`.

Tabela 3.1: Mapiranje med nenadzorovani in nadzorovani tipi

Win32 tip	ANSI C tip	.NET tip
BOOL	long	System.Int32
BYTE	unsigned char	System.Byte
CHAR	char	System.Char
DOUBLE	double	System.Double
DWORD	unsigned long	System.UInt32
FLOAT	float	System.Single
HANDLE	void *	System.IntPtr
INT	int	System.Int32
LONG	long	System.Int32
LPCSTR	const char *	System.String System.StringBuilder
LPCWSTR	const wchar_t *	System.String System.StringBuilder
LPSTR	char *	System.String System.StringBuilder
LPWSTR	wchar_t *	System.String System.StringBuilder
SHORT	short	System.Int16
UINT	unsigned int	System.UInt32
ULONG	unsigned long	System.UInt32
WORD	unsigned short	System.UInt16

Poglavje 4

Interoperabilnost med COM in .NET

4.1 Opis tehnologije COM - Component Object Model

Component Object Model (COM) je tehnologija, ki jo je v zgodnjih devetdesetih letih prejšnjega stoletja razvilo podjetje Microsoft. Namen te tehnologije je popolna odtujitev implementacije komponent od uporabnika in s tem zmožnost enostavne nadgradnje/prirejanja programa. Komponente lahko priklapljamo ali odklapljamo kar med samim izvajanje programa, kar omogoča zelo veliko fleksibilnost. COM je jezikovno in sistemsko neodvisen. Posledica tega je, da ni vezan na specifičen operacijski sistem, ampak ga lahko implementiramo na poljubni platformi.

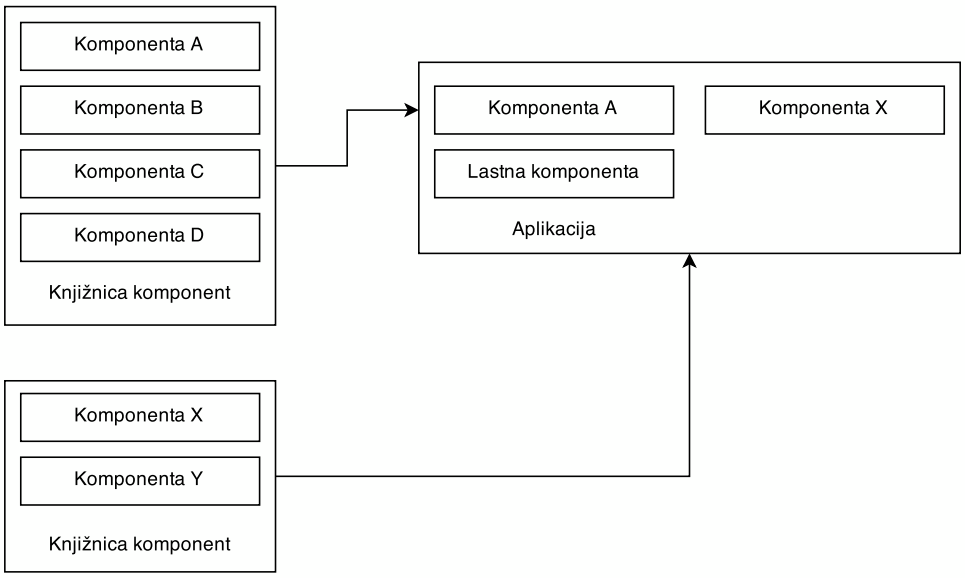
4.1.1 Komponente in njihove lastnosti

Komponente so osnovni gradniki tehnologije COM. Aplikacije so velikokrat monolitske - sestavljene iz ene same binarne datoteke ali več statično povezanih binarnih datotek. To pomeni, da je ob morebitni novi verziji potrebno aplikacijo znova prevesti in povezati, nato sledi odpošiljanje; komponente ta

problem rešijo. Pomembna prednost komponentne arhitekture je zmožnost enostavnega nadgrajevanja aplikacije skozi čas.

Knjižnice komponent nam omogočajo zelo hitro razvijanje novih aplikacij. Razlog za to je možnost, da komponente iz knjižnice oziroma njihovo poljubno podmnožico zložimo skupaj v novo aplikacijo ter razvijemo samo tisti del aplikacije, ki ga komponente, ki so na voljo, ne pokrivajo [3, str. 4]. Primer take zasnove je prikazan na sliki 4.1.

Slika 4.1: Primer aplikacije s komponentno zasnovo



Prirejanje aplikacije

Zmožnost prirejanja aplikacij je pomembna lastnost, saj omogoča vtičniško zasnovo aplikacije in posledično omogoča dodajanje in spreminjanje funkcionalnosti, ko je aplikacija že nameščena. Primeri aplikacij, kjer s pomočjo COM komponent prirejamo delovanje programa, se nahajajo v programskem

paketu Microsoft Office, kjer lahko COM komponente dodajamo kot vtičnike.

Standardiziran binarni vmesnik

COM definira binarni standard, ki omogoča veliko prenosljivost komponent. Potrebno je upoštevati dve zahtevi:

- standardna postavitev pomnilnika mora biti enaka virtualni tabeli C++ prevajalnika
- vsi vmesniki morajo dedovati po vmesniku `IUnknown`

Dinamično povezovanje

Brez dinamičnega povezovanja je potrebno aplikacijo ob vsaki spremembi znova prevesti in povezati. To je v nasprotju s filozofijo komponentne arhitekture, poleg tega pa od uporabnika ne moremo pričakovati, da ima na svojem računalniku ustrezen prevajalnik in povezovalnik [3, str. 6]. Uporabnik mora imeti možnost zamenjevanja komponent v času izvajanja.

Jezikovna neodvisnost

Jezikovna neodvisnost je pomemben aspekt komponentne arhitekture. Vsak proizvajalec programske opreme si želi, da bi za njegovo programsko opremo obstajalo veliko komponent. Vendar brez jezikovne neodvisnosti prisilimo pisce komponent v uporabo določenega programskega jezika, ki je morda proizvajalcu všeč, uporabniku komponent pa ne. COM zahteva kompatibilnost le na binarnem, ne pa tudi na jezikovnem nivoju.

Enkapsulacija

Klient in komponenta sta povezana preko vmesnika (angl "interface") [3, str. 6]. Komponente se lahko spreminjajo, vmesnik pa mora po tem, ko je enkrat narejen, vedno ostati enak. Ravno zaradi tega je potrebna enkapsulacija – tako klient kot implementacija komponente morata biti drug pred drugim

skrita, saj ne smeta vplivati drug na drugega. To pa lahko zagotovimo le tako, da druga o drugi ne vesta nič. V nasprotnem primeru seveda ne bi mogli komponent dinamično povezovati, saj bi spremembe zahtevale ponovno prevajanje in povezovanje.

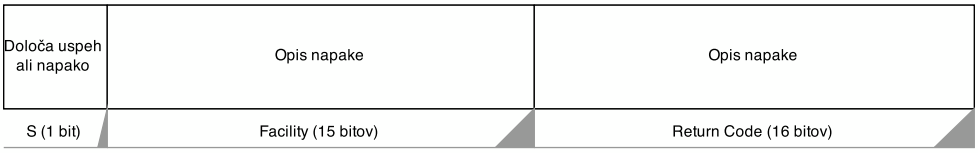
4.1.2 Komunikacija v tehnologiji COM

Tehnologija COM ima poseben način, kako komponente med seboj komunicirajo. Imamo več tipov, kot so `HRESULT` ter `GUID`, za komunikacijo pa uporablja tudi Widnows register (angl. "registry").

HRESULT

`HRESULT` je 32 bitno število, ki nakazuje uspeh ali neuspeh posamezne operacije [40]. Uporaba takega načina komuniciranja je posledica tega, da COM ne zna upravljati z izjemami, zato se mora zateči k manj invazivnim načinom poročanja o uspehu oziroma neuspehu posamezne operacije. Sestavo števila `HRESULT` lahko vidimo na sliki 4.2. Standardne kode za COM komponente so prikazane v tabeli 4.1.

Slika 4.2: Sestava števila HRESULT



Koda rezultata (angl. "Return Code"): biti [0 - 15]

Spodnjih 16 bitov nakazuje kodo, ki jo je vrnila neka funkcija. Kljub temu, da približno vemo, kakšne kode nam lahko vrne klic funkcije, pa dobljene vrednosti ni priporočljivo primerjati direktno z neko standardno vrednostjo. Namesto tega uporabljamo makra `SUCCEEDED()` in `FAILED()`. Definicija je prikazana v izseku 4.1.

Tabela 4.1: Najpogostejše kode, ki jih vračajo COM komponente [8]

S_OK	funkcija je uspela in vrnila TRUE
NOERROR	enako kot prejšnja
S_FALSE	funkcija je uspela in vrnila FALSE
E_FAIL	splošna napaka
E_NOTIMPL	funkcija ni implementirana
E_NOINTERFACE	komponenta ne podpira zahtevanega vmesnika
E_OUTOFMEMORY	komponenta ni uspela pridobiti pomnilnika
E_UNEXPECTED	neznana napaka
E_POINTER	kazalec ni veljaven
E_INVALIDARG	eden ali več parametrov ni veljavnih
E_HANDLE	neveljavna instanca
E_ACCESSDENIED	splošna napaka dostopa

Izsek 4.1: Definicija makrov za preverjanje uspešnosti operacije

```
#defune SUCCEEDED(hr) (((HRESULT)(hr)) >= 0)
#define FAILED(hr) (((HRESULT)(hr)) < 0)
```

Iz tega primera je zopet očitno, da je bit 31 tisti, ki določa uspešnost. Ta bit v 32-bitnem predznačenem številu določa predznak.

Lokacijska koda: bit [16 - 30]

Lokacijska koda (angl "facility code") prepozna del operacijskega sistema, od koder vrnitvena koda izvira. Pisec operacijskega sistema si vzame pravico za definicijo teh kod. Kode so definirane v zaglavni datoteki `winerror.h`.

Resnost (angl. "Severity"): bit [31]

Bit 31 označuje uspešnost ali napako. Če je bit 31 postavljen na 1, potem imamo napako, v nasprotnem primeru pa ne.

4.1.3 GUID

GUID (angl. "Global Unique Identifier") je 128-bitno število. Razlog za tako veliko število je v tem, da želimo, da je vsak GUID enoličen. Glede na to, da imamo možnih 2^{128} kombinacij, je verjetnost za enoličnost števila GUID zelo velika.

Uporaba števil GUID

Število GUID se uporablja za identifikacijo vmesnikov. Vsak vmesnik ima unikaten GUID, s pomočjo katerega ga pokličemo. Če imamo hipotetičen vmesnik `IMyComponent`, bomo morali v času pisanja odjemalca za COM strežnik do komponente `MyComponent` dostopati preko `IMyComponent` vmesnika. Da bomo ta vmesnik lahko poklicali, potrebujemo njegovo identifikacijo oziroma GUID. Prevajalnik za jezik IDL (glej 4.1.4) v generirane datoteke za vmesnik `IMyComponent` zapiše spremenljivko `IID_IMyComponent`, ki bo služila kot sredstvo za identifikacijo tega vmesnika.

4.1.4 Vmesniki

Osnovna zgradba vmesnikov

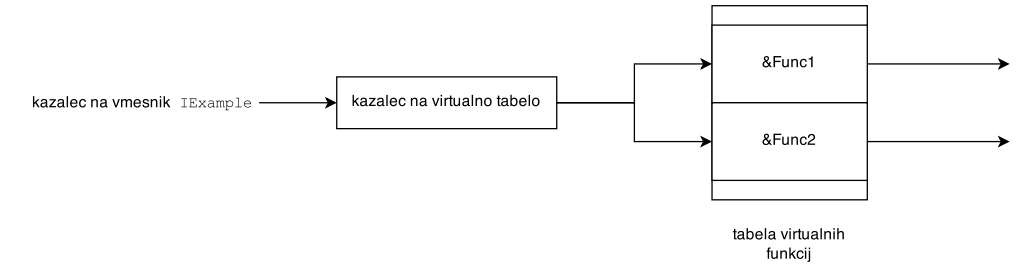
Grajenje vmesnikov temelji na sledeči filozofiji: Imamo objekt A in objekt B. Objekt A želimo priklopiti na objekt B, pri čemer sta lahko A in B popolnoma različna. To je možno, če imamo nek standardiziran vmesnik oziroma komunikacijski protokol. Tehnologija COM sledi tej filozofiji, saj so vsi vmesniki vedno zgrajeni enako ter neodvisni od implementacije komponente, ki jo enkapsulirajo. Osnovni jezik za definiranje COM vmesnikov je jezik C++.

Vmesniki COM in virtualne tabele

Vsak razred, ki ima vsaj eno čisto abstraktno funkcijo, je abstraktni razred, kar pomeni, da ne moremo imeti njegove instance. Zahteva po tem, da so vmesniki čisti abstraktni razredi, ni naključna. To s strani prevajalnika

sproži posebno postavitev virtualne tabele (angl. "vtable") [3, str. 29], kar omogoča, da imajo vsi vmesniki enak binarni standard. Primer grafične predstavitve virtualne tabele vidimo na sliki 4.3.

Slika 4.3: Vizualna predstavitev virtualne tabele, ki je nujna za programiranje COM komponent



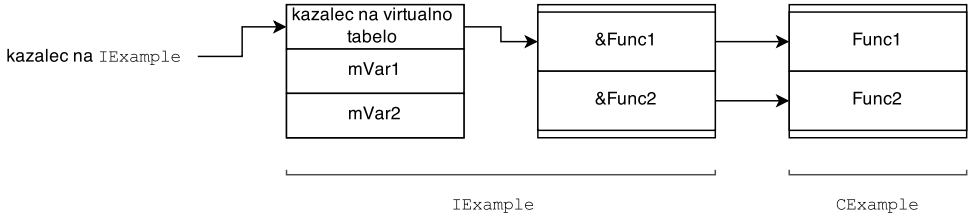
Noben standard C++ prevajalnikov ne obvezuje, da upoštevajo omenjeno postavitev virtualne tabele (C++ prevajalnik podjetja Microsoft to postavitev upošteva). Kazalec virtualne tabele doda novo stopnjo abstrakcije. Če implementiramo še razred, ki deduje po vmesniku (s tem postane komponenta), ki jo bo vmesnik enkapsuliral, se v virtualno tabelo doda še en nivo, kot je prikazano v izseku 4.2.

Izsek 4.2: Primer implementacije komponente, s katero bomo komunicirali preko vmesnika IExample

```
class CExample : public IExample
{
    public:
        CExample(int num): mVar1(num + num),
           mVar2(num - num){}
        virtual void __stdcall Func1()
        {
        //implementacija
        }
        virtual void __stdcall Func2()
```

```
        {
        //implementacija
        }
    private:
        int mVar1;
        int mVar2;
};
```

Slika 4.4: Vmesnik skupaj z enkapsulirano implementacijo, kar tvori komponento



Posledica te postavitve je, da lahko kličemo komponente le preko vmesnikov in tako popolnoma skrijemo njihovo implementacijo pred klientom. Vizualna predstavitev vmesnika in zaledne implementacije je vidna na sliki 4.4.

Jezik IDL (Interface Description Language)

Jezik IDL je namenjen opisovanju vmesnikov in njihovih članov. Microsoft ima svoj prevajalnik za jezik IDL, imenovan MIDL. V jeziku IDL vmesnik opremimo z različnimi metapodatki, kot so verzija in GUID ter smer (`in`/`out`) posameznih argumentov funkcij v vmesniku. Primer definicije vmesnika v jeziku IDL je prikazan spodaj:

```
    [
    uuid(34df5c9b-01f7-474b-b439-613c7a74942a),
    version(1.3)
```

```
]
interface MyInterface
{
    const unsigned short ARRAY_LEN = 200;
    void MyProcedure([in] int param1, [out] int
       outArray[INT_ARRAY_LEN]);
}
```

Prevajalnik za jezik IDL nato generira ustrezne datoteke v jeziku C++. Uporaba jezika IDL ni obvezna za grajenje vmesnikov ter COM komponent, je pa priporočljiva, saj odstrani nekaj kompleksnosti.

Grajenje Windows kompatibilnih vmesnikov

Implementacija tehnologije COM na operacijskem sistemu Windows je že v svoji osnovi opremljena z določenimi komponentami in vmesniki. Za grajenje lastnih vmesnikov, ki se bodo lahko povezovani z njihovimi, je potrebno upoštevati določena pravila.

Vsaka komponenta mora biti vedno dostopna vsaj preko osnovnega vmesnika `IUnknown`. To v praksi pomeni, da mora vsak razred, ki predstavlja implementacijo komponente, dedovati po vmesniku `IUnknown` [29]. Vmesnik je definiran kot:

```
interface IUnknown
{
    virtual HRESULT __stdcall QueryInterface(
       const IID&, void**) = 0;
    virtual ULONG __stdcall AddRef() = 0;
    virtual ULONG __stdcall Release() = 0;
};
```

IDL implementacija vmesnika `IUnknown` je prikazana spodaj:

```
[
    local,
    object,
    uuid(00000000-0000-0000-C000-000000000046)
]
interface IUnknown
{
    HRESULT __stdcall QueryInterface([in] REFIID
        riid,[out] void** ppv) = 0;
    ULONG __stdcall AddRef(void) = 0;
    ULONG __stdcall Release(void) = 0;
};
```

Funkcija QueryInterface

Funkcija `QueryInterface` vrne kazalec na željen vmesnik pod pogojem, da ta obstaja za določeno komponento. Povpraševanje po vmesniku `IUnknown` je edino, ki mora vedno uspeti [30]. Ker je vsak vmesnik enolično definiran z 128 bitnim številom GUID, po njem vprašujemo z njegovim unikatnim identifikatorjem. Primer implementacije funkcije `QueryInterface`:

```
HRESULT __stdcall CExample::QueryInterface(const
    IID& i,void** ppv)
{
    if (i == IID_IUnknown)
    {
            *ppv = static_cast<IUnknown*>(this);
    }
            else if (i == IID_IExample)
    {
            *ppv = static_cast<IExample*>(this);
```

```
        }
        else
        {
                *ppv = NULL;
                return E_NOINTERFACE
        }
        return S_OK;
    }
```

Funkciji AddRef() in Release()

Ti dve funkciji rešujeta problematiko, kdaj naj se določeni komponenti dovoli zapustiti pomnilnik. Določanje tega ročno je praktično nemogoča naloga, saj ni mogoče vedeti, ali v danem trenutku še kdo uporablja neki vmesnik (še posebej, če aplikacija uporablja več niti). Tu nastopi mehanizem štetja referenc. Vsakič, ko pokličemo neki vmesnik, je potrebno klicati funkcijo `AddRef` [31]. Ta namreč poveča števec, ki določa, koliko instanc komponente je trenutno živih. Podobno je potrebno po končani uporabi klicati funkcijo `Release`. Primer implementacije funkcij `AddRef` ter `Release` je prikazan spodaj:

```
    ULONG __stdcall AddRef()
    {
        return InterlockedIncrement(&m_cRef);
    }
    ULONG __stdcall AddRef()
    {
        if (InterlockedDecrement(&m_cRef) == 0)
        {
                delete this;
                return 0;
        }
```

```
        return m_cRef;
    }
```

Ko števec uporabe doseže nič, se komponenta pobriše iz pomnilnika [32]. Funkciji `InterlockedIncrement` in `InterlockedDecrement` skrbita za to, da se vrednosti prištevajo oz. odštevajo sinhronizirano, v primeru, da imamo več niti. Če pri štetju referenc nismo dosledni, dobimo programske hrošče, ki jih je zelo težko odkriti. Pomagamo si lahko s knjižnico ATL (Active Template Library), ki vsebuje veliko število razredov za pomoč pri pisanju in upravljanju s komponentami, kot sta pametna kazalca `CComPtr` ter `CComQIPtr`, ki omogočata avtomatsko in varno štetje referenc.

4.2 Interoperabilnost med tehnologijama

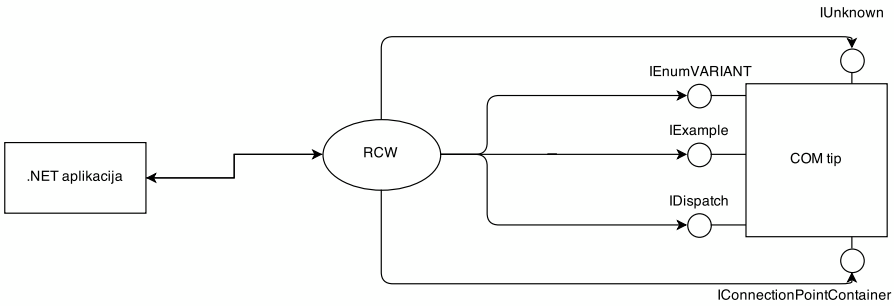
Za razliko od tehnologije P/Invoke, kjer neko funkcijo iz nenadzorovane knjižnice DLL pokličemo v okolje .NET, je interoperabilnost s COM bolj kompleksna, saj ima COM bolj zapletene protokole ter nasploh bolj zapleteno strukturo nenadzorovanih knjižnic DLL.

4.2.1 RCW - Ovojnica za klicanje v času izvajanja

COM ter .NET sta si po zgradbi in delovanju precej različna. Zaradi tega so v platformo .NET vgradili mehanizem, ki te razlike premosti, ter tako omogoča, da so razlike v delovanju med tehnologijama navzven nevidne. Ovojnica za klicanje v času izvajanja (angl. "Runtime Callable Wrappper - RCW") je .NET komponenta, ki je zadolžena za komuniciranje med nadzorovanimi in nenadzorovanimi enotami kode. Nadzorovana koda kliče nenadzorovano, RCW pa vsak posamezni klic prestreže, ga spremeni v klic, primeren za klicanje nenadzorovane COM kode, ter prenese klic naprej, do implementacije komponente [5, str. 340, 341]. RCW deluje tudi, ko pride do klicev v povratni smeri, in sicer, če komponenta vrača kakršnokoli informacijo klicatelju. Naloga RCW pa ni zgolj komunikacija med nenadzorovano ter nadzorovano

kodo, temveč tudi poenostavitev; COM za svoje delovanje uporablja mehanizme kot so štetje referenc in tovarne objektov, .NET pa uporablja prave objektno usmerjene mehanizme, kot je uporaba rezervirane besede `new` za kreacijo novih objektov ter avtomatsko čiščenje odpadnih objektov. RCW poskrbi, da platformi .NET ni potrebno odstopati od svojih pravil, kadar komunicira s COM komponentami, ampak jih lahko obravnava kot poljuben .NET objekt. RCW pretvarja med COM ter .NET tipi glede na tipe, ki jih uporablja IDL. Vsak tip, podprt s strani jezika IDL ima ustrezen ekvivalent na platformi .NET. Prikazani so v tabeli 4.2. Vsaka instanca COM komponente, s katero platforma komunicira, ima svoj RCW, saj tako .NET lažje avtomatsko upravlja s pomnilnikom, in odstranjuje komponente, ki niso več v uporabi. Povezava med .NET aplikacijo ter COM komponento je prikazan na sliki 4.5.

Slika 4.5: Vizualna predstavitev povezave med .NET aplikacijo ter COM komponento



Poleg mapiranja primitivnih IDL tipov, je RCW zadolžen tudi za upravljanje z različnimi COM vmesniki, ki omogočajo, da .NET klient ne opazi razlike med COM ter .NET objektom. To je potrebno zaradi bistveno večje moči platforme .NET, saj je le ta zmožna stvari, ki jih COM ne ne zna. Mednje spadajo odsevnost objektov v času izvajanja, ter dogodki na osnovi delegatov. Vmesniki, ki jih RCW priklopi nase v času izvajanja, ter njihov namen so opisani spodaj [5, str. 351, 352]:

Tabela 4.2: Mapiranje med IDL ter .NET tipi

`bool, bool *`	`System.Int32`
`char, char *`	`System.SByte`
`small , small *short, short *`	`System.SByte`
`long, long *`	`System.Int16`
`int , int *`	`System.Int32`
`hyper, hyper *`	`System.Int64`
`unsigned char, unsigned char *`	`System.Byte`
`byte, byte *`	`System.Byte`
`wchar_t, wchar_t *`	`System.UInt16`
`unsigned short, unsigned short *`	`System.UInt16`
`unsigned short unsigned short *`	`System.UInt16`
`unsigned long unsigned long *`	`System.Int32`
`unsigned int unsigned int *`	`System.Int32`
`unsigned hyper`	`System.UInt64`
`unsigned hyper *`	`System.UInt64`
`float, float *`	`System.Single`
`double, double *`	`System.Double`

IUnknown Vmesnik `IUnknown` je skupen vsem komponentam v tehnologiji COM. Skrbi za življensko dobo objekta in njegovo pretvarjanje v druge tipe. Ker platforma .NET nikoli direktno ne kliče funkcij, ki jih `IUnknwon` implementira, RCW ne omogoča direktnega dostopa do teh funkcionalnosti na platformi .NET (ampak jih uporablja sam, kar pa ni vidno s strani .NET klienta).

IDispatch Vmesnik `IDispatch` je namenjen konceptu poznega priklopa (angl. "late binding"), ki se uporablja predvsem v podsistemu sistema COM, imenovanem Automation, ki je namenjen dostopu do COM komponent iz skriptnih jezikov.

VARIANT_BOOL	System.Boolean
VARIANT_BOOL *	System.Boolean
void *, void **	System.IntPtr
HRESULT, HRESULT *	System.Int16,System.IntPtr
SCODE, SCODE *	System.Int32
BSTR, BSTR *	System.String
LPSTR, char *	System.String
LPSTR *	System.String
LPWSTR, wchar_t *	System.String
LPWSTR *	System.String
VARIANT, VARIANT *	System.Object
DECIMAL, DECIMAL *	System.Decimal
CURRENCY, CURRENCY *	System.Decimal
DATE, DATE *	System.Decimal
GUID, GUID *	System.Guid
IUnknown *, IUnknown **	System.Object
IDispatch * IDispatch **	System.Object
SAFEARRAY(type)	type[] (System.Array)

IProvideClassInfo RCW uporabi ta vmesnik za grajenje močnejše identitete tipov.

IDispatchEx Vmesnik `IDispatchEx` je razširitev vmesnika `IDispatch`, ter omogoča enumeracijo, dodajanje, brisanje ter klicanje elementov komponente, ki jo implementira. Če RCW implementira `IDispatchEx` (in ne samo `IDispatch`), potem to implementira preko vmesnika imenovanega `IExpando`, ki pa deduje po vmesniku `IReflect`. Ta je namenjen interoperabilnosti z vmesnikom `IDispatch` na platformi .NET.

IErrorInfo Vmesnik služi delu z napakami na platformi COM. Za razliko od platforme .NET, kjer imamo strukturirane izjeme, COM to počne s pošiljanjem posebnih števil tipa `HRESULT`. RCW prevaja ta števila v

strukturirani sistem izjem platforme .NET.

IConnectionPoint Vmesnik omogoča pravilno mapiranje COM dogodkov v .NET dogodke.

IEnumVARIANT Vmesnik omogoča, da se COM enumeracijski tipi preslikajo v .NET kolekcije, ter tako omogoča delo z operatorjem `foreach`.

Poglavje 5

Uporaba opisanih tehnologij v praksi

V tem poglavju sem bom osredotočil na implementacijo tehnologij, predstavljenih v prejšnjih poglavjih. V praktičnem delu svojega diplomskega dela se bom osredotočil na pridobivanje metapodatkov iz nadzorovanih zbirk ter interoperabilnost s tehnologijo COM, bolj natančno z tehnologijo Automation, ki je del tehnologije COM. V svojem praktičnem delu bom uporabil programski jezik C#, trenutno najbolj razširjen ter popularen jezik na platformi .NET.

5.1 Opis problema

Svoj programski problem sem razdelil na dva dela: Pridobivanje metapodatkov iz nadzorovanih zbirk ter interoperabilnost s tehnologijo Automation v obliki izvoza podatkov v Microsoft Office Excel.

5.1.1 Pridobivanje metapodatkov

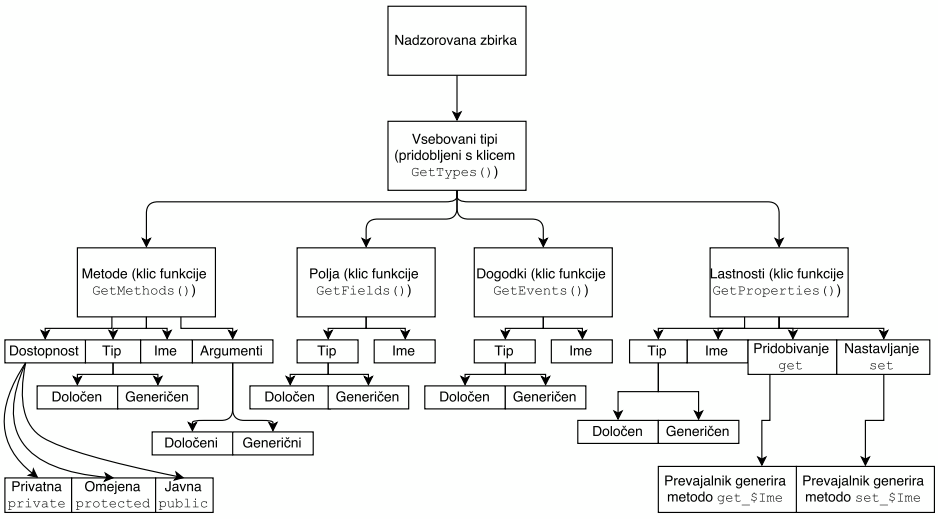
Pridobivanje metapodatkov iz nadzorovanih zbirk je na platformi .NET relativno enostavna naloga, saj platforma .NET že vsebuje knjižnice, ki to omogočajo. Mehanizem, s pomočjo katerega sem se dokopal do metapo-

datkov se imenuje odsevnost (angl. "reflection"). Koraki pri pridobivanju metapodatkov so naslednji:

- dinamično nalaganje zbirke v aplikacijsko domeno aplikacije
- iskanje tipov ter njihovih imenskih prostorov v nadzorovani zbirki
- iskanje vsebovanih elementov tipov v nadzorovani zbirki
  - polja (angl. "fields")
  - lastnosti (angl. "properties")
  - metode (angl. "methods")
  - dogodki (angl. "events")

Na sliki 5.1 je razviden postopek pridobivanja metapodatkov iz nadzorovane zbirke. Knjižnica za njihovo pridobivanje se nahaja v imenskem prostoru `System.Reflection`.

Slika 5.1: Postopek pridobivanja podatkov iz nadzorovane zbirke



Vsak tip v nadzorovani zbirki, ki sovpada z .NET tipom imenovanim `System.Type`, lahko predstavlja:

- tipe razredov (angl. "class types")
- tipe vmesnikov (angl. "interface types")
- tipe tabel (angl. "array types")
- vrednostne tipe (angl. "value types")
- enumeracijske tipe (angl. "enumeration types")
- parametre tipov (angl. "type parameters")
- generične definicije tipov (angl. "generic type definitions")
- opdrte ali zaprte konstruirane generične tipe (angl. "open or closed constructed generic types")

Iz vsakega posameznega tipa v nadzorovani zbirki nato z metodami, ki jih tip `System.Type` vsebuje, pridobivam informacije o članih posameznega tipa. Metode so:

- `GetFields` za pridobivanje podatkov o poljih
- `GetProperties` za pridobivanje podatkov o lastnostih
- `GetEvents` za pridobivanje podatkov o dogodkih
- `GetMethods` za pridobivanje podatkov o metodah

Poleg podatkov, ki so dostopni direktno s pomočjo metod tipa `System .Type`, sem v svoji aplikaciji uporabil tudi zunanjo knjižnico, ki omogoča prikaz implementacije posamezne metode v IL jeziku. Knjižnica se imenuje ILDisassembler [42]. Prikaz implementacije posamezne metode v jeziku IL je možen zaradi dejstva, da informacija o metodi, ki jo dobimo preko `System.Type`, omogoča tudi dostop do IL ukazov, in sicer preko klica funkcije `GetMethodBody`. Vendar pa so ti ukazi podani kot množica binarnih podatkov, tipa `MethodBody`. Knjižnica ILDisassembler nam omogoča prevedbo teh podatkov v ljudem berljivo obliko.

Primer pridobivanja metapodatkov v jeziku C#

Pridobivanje metapodatkov v jeziku C# je mogoče na zelo strnjen način, z uporabo vgrajene tehnologije LINQ. Najprej dinamično naložimo nadzorovano zbirko, nato pa za vsak vsebovan tip pridobimo potrebne podatke. Primer:

```
var assembly = Assembly.LoadFrom(path);
assembly.GetTypes().ToList().ForEach(type =>
{
    GetFieldInfo(type);
    GetPropertyInfo(type);
    GetMethodInfo(type);
    GetEventInfo(type);
});
```

V spodnji metodi je opisan način pridobivanja podatkov o vsebovanih metodah v posameznem tipu:

```
public List<MethodInfoDto> GetMethodInfo(System.
   Type classType)
{
    var methodInfos = classType.GetMethods();
    var list = new List<MethodInfoDto>();
    methodInfos.ToList().ForEach(info =>
    {
        var item = new MethodInfoDto();
        item.MethodName =
            string.Format("{0} {1} ({2})",
            GetMethodAccessModifiers(info),
            info.Name, GetMethodParameters(info)
               );
        item.MethodBody = GetMethodILBody(info);
        list.Add(item);
```

```
        });
        return list;
    }
```

Metoda `GetMethods` nam vrne seznam opisov vsebovanih metod posameznega tipa. Iz posameznega elementa seznama nato lahko pridobimo vse ustrezne informacije o metodi, ko so stopnja dostopnosti, statičnost, parametri ter tudi bajtna koda metode. Stopnjo dostopnosti pridobimo na sledeči način:

```
public string GetMethodAccessModifiers(
    MethodInfo info)
{
    var mod = "";
    if (info.IsPrivate) mod += "private";
    if (info.IsPublic) mod += "public";
    if (info.IsStatic) mod += " static";
    return mod;
}
```

Parametre ter informacije o njih lahko pridobimo z metodo, imenovano `GetParameters`, ki je vsebovana v razredu `MethodInfo`. Vrne nam seznam informacij o parametru, ki so tipa `ParameterInfo`. Iz posameznega elementa seznama nato lahko pridobimo informacije, kot so generičnost ter tip parametra. Primer pridobivanja informacij o parametrih je opisan v spodnjem izseku:

```
    public string GetMethodParameters(MethodInfo
       info)
    {
        var parameters = info.GetParameters();
        var list = new List<string>();
        parameters.ToList().ForEach(x =>
        {
```

```
            if (!x.ParameterType.IsGenericType)
            {
                list.Add(String.Format("{0} {1}", x.
                    ParameterType.Name, x.Name));
            }
            else
            {
                list.Add(String.Format("{0} {1}",
                    GetGenericArguments(x.
                    ParameterType), x.Name));
            }
            });
        return string.Join(", ", list.ToArray());
    }
```

Vsi generični parametri posameznega tipa so pridobljeni z metodo `GetGenericArguments`. Ta operira na tipu `System.Type`, implementacija pa je prikazana v spodnjem izseku:

```
    public string GetGenericArguments(Type t)
    {
        var list = new List<string>();
        t.GetGenericArguments().ToList().ForEach(x
            => list.Add(x.Name));
        return string.Join(",", list.ToArray());
    }
```

Zadnja pridobljena informacija o metodi je njena implementacija v IL zbirniku. Za vsako metodo lahko, preko razreda `MethodInfo`, pridobimo IL implementacijo in sicer preko klica funkcije `GetMethodBody`, vendar pa dobljeni rezultat še ni človeško berljiv. V ta namen sem uporabil dodatno knjižnico, ki rezultat pretvori v človeško berljivo obliko. Primer pridobivanja človeško berljive kode je prikazan v spodnjem izseku:

```
public string GetMethodILBody(MethodInfo info)
{
    Disassembler dis = new Disassembler();
    return dis.DisassembleMethod(info);
}
```

Iz zgornjih izsekov je vidno, da je pridobivanje metapodatkov na platformi .NET relativno enostavno, vendar pa je za bolj kompleten pregled nad njimi vseeno potrebno implementirati dodatne funkcionalnosti.

5.1.2 Izvoz podatkov v Microsoft Office Excel

Izvoz podatkov v Microsoft Office Excel je možen na več različnih načinov. Sam sem izbral izvoz s pomočjo tehnologije Automation, ki je del tehnologije COM, kar omogoča dostop do COM komponente brez posredovanja RCW. Odkar je Microsoft posodobil svoj format, je možno ustvarjati Excel (in tudi Word) datoteke s pomočjo dialekta jezika XML, imenovanega OpenXML.

Automation v kombinaciji z prevajanimi jeziki

Tehnologija Automation je bila zasnovana z namenom uporabljanja aplikacij iz skriptnih jezikov – predvsem Visual Basic in dialekti, danes pa ta mehanizem podpira kar nekaj jezikov, kot npr. Python (izsek 5.1), Ruby, Powershell in še mnogi drugi. V skriptnih jezikih je uporaba mehanizma Automation zelo enostavna [2], kot je prikazano v spodnjem izseku, v statično tipiziranih jezikih, kot je C#, pa je bila uporaba do vpeljave mehanizma DLR nekoliko bolj zapletena.

Izsek 5.1: Primer dostopa do Excel Automation programskega vmesnika in njegove uporabe v jeziku Python

```
import win32com.client
app = win32com.client.Dispatch("Excel.
   Application")
```

```
workBook = app.Wokrbooks.Add("C:\\wb.xls")
workSheet = app.Worksheets("Reflection Data")
for row in ws.UsedRange.Rows:
    print row.Cells(1,2).Value
```

S četrto verzijo platforme .NET (.NET 4) se je interakcija z mehanizmom Automation poenostavila na nivo skriptnih jezikov, saj je CLR izvajalno okolje dobilo dodatno tehnologijo, imenovano Dinamično izvajalno okolje (angl. "Dynamic Language Runtime - DLR"). DLR je bil v svoji osnovi namenjen lažjemu prenosu dinamično tipiziranih jezikov na platformo .NET [28]. Njegove naloge obsegajo:

- enostavnejše selitve dinamičnih jezikov na platformo .NET
- uporaba lastnosti dinamičnih jezikov v statično tipiziranih jezikih
- hitro dinamično razpošiljanje in klicanje

Omenjene lastnosti mehanizma DLR so za uporabo mehanizma Automation pomembne, saj tipi Automation operacij niso znani v času prevajanja, zato jim ni mogoče dodeliti nobenega statičnega tipa. S pomočjo mehanizma DLR ter rezervirane besede `dynamic` prevajalniku povemo, da bo tip posamezne spremenljivke ugotavljalo izvajalno okolje CLR, oziroma njegov dodatek DLR.

Automation uporablja poseben tip `VARIANT`, ki lahko prevzame vlogo poljubnega IDL tipa. Služijo podobnemu namenu, kot v .NET tip `System.Object`, vendar tip `VARIANT` ni tako močan in vsestranski. Tipi, podprti znotraj tipa `VARIANT` so prikazani v tabeli 5.2.

Uporaba Excel Automation programskega vmesnika v jeziku C#

V praktičnem delu svoje diplomske naloge uporabljam Automation za izvoz podatkov o nadzorovanih zbirkah v Excel dokument. Kljub temu, da je programski vmesnik Excel Automation zelo velik, kot je razvidno iz tabele v dodatku A.1, bom pri funkcionalnosti uporabil le majhen košček vsega, kar

nam ponuja. Proces uporabe tehnologije Automation v jeziku C# se prične z izgradnjo pravilnega Automation tipa. Glede na to, da Excel svoj Automation programski vmesnik registrira pod imenom "Excel Automation", je potrebno najprej ustvariti ustrezen tip. To storimo z:

```
Type excelType = Type.GetTypeFromProgID("Excel.
   Application")
```

Pridobljeno spremenljivko `excelType` moramo nato aktivirati in s tem se naša komunikacija z Excel Automation programskim vmesnikom začne. Tip aktiviramo s posebnim razredom `Activator`, bolj natančno, s klicem metode `CreateInstance` razreda `Activator`. Ker tip, ki ga dobimo s klicem funkcije `CreateInstance` v času prevajanja ni znan, se moramo zanesti na tip `dynamic` ter mehanizem DLR, da ga bodo pravilno konstruirali v času izvajanja. Ko imamo pod nadzorom instanco aplikacije "Excel.Application", lahko uporabimo karkoli iz programskega vmesnika. V svoji aplikaciji Automation uporabljam na spodaj prikazan način:

```
    Type excelType = Type.GetTypeFromProgID("Excel.
       Application");
    dynamic excelApp = Activator.CreateInstance(
       excelType);
    dynamic workBook = excelApp.Workbooks.Add();
    dynamic activeSheet = workBook.ActiveSheet();
    activeSheet.Cells[count, indent] = data;
    workBook.SaveAs(savePath);
    workBook.Close();
    excelApp.Quit();
```

Kot je razvidno iz Excel Automation programskega vmesnika, so nekatere lastnosti v Excel Automation programskem vmesniku tipa `System.__ComObject`. To je posebni interni tip platforme .NET, ki nam pove, da se za tem objektom skriva neka COM komponenta, vendar pa ni znano katera. Tako `Workbook` kot `ActiveSheet` sta tipa `System.__ComObject`, klici

na teh objektih, kot je `Workbooks.Add()`, pa se bodo zgodili in razrešili v času izvajanja, in sicer s klici platforme v vmesnik `IDispatch`.

Slika 5.2: Tip VARIANT

Referenca na `null` objekt	`VT_EMPTY`
`System.DBNull`	`VT_NULL`
`ErrorWrapper`	`VT_ERROR`
`System.Reflection.Missing`	`VT_ERROR`
`DispatcherWrapper`	`VT_DISPATCH`
`UnknownWrapprer`	`VT_UNKNOWN`
`CurrencyWrapper`	`VT_CY`
`System.Boolean`	`VT_BOOL`
`System.SByte`	`VT_I1`
`System.Byte`	`VT_UI1`
`System.UInt16`	`VT_UI2`
`System.Int32`	`VT_I4`
`System.UInt32`	`VT_I4`
`System.Int64`	`VT_I8`
`System.UInt64`	`VT_UI8`
`System.Single`	`VT_R4`
`System.Double`	`VT_R8`
`System.Decimal`	`VT_DECIMAL`
`System.DateTime`	`VT_DATE`
`System.String`	`VT_BSTR`
`System.IntPtr`	`VT_INT`
`System.UIntPtr`	`VT_UINT`
`System.Array`	`VT_ARRAY`

Poglavje 6

Sklepne ugotovitve

6.1 Teoretični del

V teoretičnem delu svoje diplomske naloge sem ugotovil, da je .NET zelo vsestranska platforma. Omogoča, da z relativno enostavnostjo preidemo iz starejših tehnologij, kot je COM, na .NET, ter s tem pridobimo vse ugodnosti nadzorovanega okolja. Interoperabilnost z nenadzorovanimi aplikacijami je enostavna in zelo mogočna. Kot potencialna težava platforme .NET se mi zdi njena zelo velika odvisnost od programskega okolja Visual Studio, brez katerega bi bila uporaba vseh tehnologij v skladu .NET zelo zahtevna naloga.

6.2 Praktični del

Iz praktičnega dela svoje diplomske naloge sem ugotovil, da je odsevnost dobro podprta na platformi .NET. Vsa logika, potrebna za pridobivanje strukture nadzorovanih zbirk, je že vdelana na platformo .NET ter se nahaja v imenskem prostoru `System.Reflection`. Tip `System.Type` je prav tako zelo vsestranski element platforme .NET. Z vgradnjo DLR tehnologije v platformo .NET je tudi uporaba konstruktov, namenjenim skriptnim jezikom, relativno enostavna. Statično tipiziran in prevajan jezik se tako poenostavi na nivo jezika Python ali podobnega skriptnega jezika. Kljub enostavnosti dobimo

tudi veliko stopnjo abstrakcije, saj lahko upravljamo s COM komponentami kot s poljubnim .NET objektom. Pri interoperabilnosti z nenadzorovanimi tehnologijami, kot je COM, pa je kljub vsem mehanizmom in abstrakcijam, ki jih vsebuje platforma .NET, potrebno biti pazljiv, saj vgrajeni mehanizmi ob nepravilni uporabi ne morejo pravilno počistiti težav, kot jih to storijo v čistih .NET aplikacijah. Primer sta zadnja dva klica iz praktičnega primera, in sicer `workBook.Close()` ter `excelApp.Quit()`. Brez teh dveh klicev proces Excel.Application ne bi sprostil ključavnice nad datoteko kamor zapiše podatke ter bi ostal živ, kljub temu, da ne bi imel nobene reference. Ko bi se aplikacija končala, bi imeli zombi proces; s tem so lepo vidne omejitve interoperabilnosti platforme .NET.

Dodatek A

Dodatek - Programski vmesnik Excel Automation

Vse Microsoft Office aplikacije imajo vdelan tudi programski vmesnik za interoperabilnost z tehnologijo Automation. Ker v praktičnem delu svoje diplomske naloge uporabljam Microsoft Office Excel, so tu (tabela A.1) predstavljene vse lastnosti in metode, ki jih lahko uporabimo, skupaj z njihovimi privzetimi vrednostmi. Vrednosti so pridobljene preko skriptnega jezika Powershell, in sicer s sledečim ukazom:

```
$automation = New-Object -ComObject "Excel.Application"
$automation
```

Tabela A.1: Funkcije in lastnosti, dosegljive preko programskega vmesnika Excel Automation ter njihove privzete vrednosti na avtorjevem računalniku

Application	Microsoft.Office.Interop.Excel.ApplicationClass
Creator	xlCreatorCode
Parent	Microsoft.Office.Interop.Excel.ApplicationClass
ActiveCell	(ni privzete vrendnosti)
ActiveChart	(ni privzete vrendnosti)
ActiveDialog	(ni privzete vrendnosti)
ActiveMenuBar	System.__ComObject

ActivePrinter	Xerox WorkCentre 3045NI on Ne03:
ActiveSheet	(ni privzete vrendnosti)
ActiveWindow	(ni privzete vrendnosti)
ActiveWorkbook	(ni privzete vrendnosti)
AddIns	System.__ComObject
Assistant	System.__ComObject
Cells	(ni privzete vrendnosti)
Charts	(ni privzete vrendnosti)
Columns	(ni privzete vrendnosti)
CommandBars	{System.__ComObject, System.__ComObject, System .__ComObject,...}
DDEAppReturnCode	0
DialogSheets	(ni privzete vrendnosti)
MenuBars	System.__ComObject
Modules	(ni privzete vrendnosti)
Names	(ni privzete vrendnosti)
Rows	(ni privzete vrendnosti)
Selection	(ni privzete vrendnosti)
Sheets	(ni privzete vrendnosti)
ThisWorkbook	(ni privzete vrendnosti)
Toolbars	System.__ComObject
Windows	System.__ComObject
Workbooks	System.__ComObject
WorksheetFunction	System.__ComObject
Worksheets	(ni privzete vrendnosti)
Excel4IntlMacroSheets	(ni privzete vrendnosti)
Excel4MacroSheets	(ni privzete vrendnosti)
AlertBeforeOverwriting	True
AltStartupPath	(ni privzete vrendnosti)
AskToUpdateLinks	True

EnableAnimations	True
AutoCorrect	System.__ComObject
Build	4719
CalculateBeforeSave	(ni privzete vrendnosti)
Calculation	(ni privzete vrendnosti)
CanPlaySounds	True
CanRecordSounds	True
Caption	Excel
CellDragAndDrop	True
DisplayClipboardWindow	False
ColorButtons	True
CommandUnderlines	xlCommandUnderlinesAutomatic
ConstrainNumeric	False
CopyObjectsWithCells	True
Cursor	xlDefault
CustomListCount	4
CutCopyMode	0
DataEntryMode	-4146
_Default	Microsoft Excel
DefaultFilePath	C:\Users\Andrej\Documents
Dialogs	System.__ComObject
DisplayAlerts	True
DisplayFormulaBar	True
DisplayFullScreen	False

DisplayNoteIndicator	True
DisplayCommentIndicator	xlCommentIndicatorOnly
DisplayExcel4Menus	False
DisplayRecentFiles	True
DisplayScrollBars	True
DisplayStatusBar	True
EditDirectlyInCell	True
EnableAutoComplete	True
EnableCancelKey	xlInterrupt
EnableSound	False
EnableTipWizard	False
FileSearch	(ni privzete vrendnosti)
FileFind	(ni privzete vrendnosti)
FixedDecimal	False
FixedDecimalPlaces	2
Height	569,25
IgnoreRemoteRequests	False
Interactive	True
Iteration	(ni privzete vrendnosti)
LargeButtons	False
Left	176,5
LibraryPath	C:\Program Files\Microsoft Office\Office15\ LIBRARY
MailSession	(ni privzete vrendnosti)
MailSystem	xlMAPI
MathCoprocessorAvailable	True

MaxChange	(ni privzete vrendnosti)
MaxIterations	(ni privzete vrendnosti)
MemoryFree	(ni privzete vrendnosti)
MemoryTotal	(ni privzete vrendnosti)
MemoryUsed	(ni privzete vrendnosti)
MouseAvailable	True
MoveAfterReturn	True
MoveAfterReturnDirection	xlDown
RecentFiles	System.__ComObject
Name	Microsoft Excel
NetworkTemplatesPath	(ni privzete vrendnosti)
ODBCErrors	System.__ComObject
ODBCTimeout	45
OnCalculate	(ni privzete vrendnosti)
OnData	(ni privzete vrendnosti)
OnDoubleClick	(ni privzete vrendnosti)
OnEntry	(ni privzete vrendnosti)
OnSheetActivate	(ni privzete vrendnosti)
OnSheetDeactivate	(ni privzete vrendnosti)
OnWindow	(ni privzete vrendnosti)
OperatingSystem	Windows (64-bit)NT 6.02
OrganizationName	(ni privzete vrendnosti)
Path	C:\Program Files\Microsoft Office\Office15
PathSeparator	\
PivotTableSelection	False
PromptForSummaryInfo	False
RecordRelative	False
ReferenceStyle	xlA1
RollZoom	False
ScreenUpdating	True

SheetsInNewWorkbook	1
ShowChartTipNames	True
ShowChartTipValues	True
StandardFont	Calibri
StandardFontSize	11
StartupPath	C:\Users\Andrej\AppData\Roaming\Microsoft\ ExceXLSTART
StatusBar	False
TemplatesPath	C:\Users\Andrej\AppData\Roaming\Microsoft\ Templates\
ShowToolTips	True
Top	210,75
DefaultSaveFormat	xlOpenXMLWorkbook
TransitionMenuKey	/
TransitionMenuKeyAction	1
TransitionNavigKeys	False
UsableHeight	510,75
UsableWidth	1078,5
UserControl	False
UserName	Andrej Bratoz
Value	Microsoft Excel
VBE	(ni privzete vrendnosti)
Version	15.0
Visible	False
Width	1080
WindowsForPens	False
WindowState	xlNormal
UILanguage	0
DefaultSheetDirection	-5003
CursorMovement	1
ControlCharacters	False
EnableEvents	True

DisplayInfoWindow	False
ExtendList	True
OLEDBErrors	System.__ComObject
COMAddIns	System.__ComObject
DefaultWebOptions	System.__ComObject
ProductCode	...
UserLibraryPath	C:\Users\Andrej\AppData\Roaming\Microsoft\ AddIns\
AutoPercentEntry	True
LanguageSettings	System.__ComObject
Dummy101	(ni privzete vrendnosti)
AnswerWizard	(ni privzete vrendnosti)
CalculationVersion	152511
ShowWindowsInTaskbar	True
FeatureInstall	msoFeatureInstallNone
Ready	True
FindFormat	System.__ComObject
ReplaceFormat	System.__ComObject
UsedObjects	System.__ComObject
CalculationState	xlDone
CalculationInterruptKey	xlAnyKey
Watches	System.__ComObject
DisplayFunctionToolTips	True
AutomationSecurity	msoAutomationSecurityLow
DisplayPasteOptions	True

DisplayInsertOptions	True
GenerateGetPivotData	False
AutoRecover	System.__ComObject
Hwnd	4067532
Hinstance	(ni privzete vrendnosti)
ErrorCheckingOptions	System.__ComObject
AutoFormatAsYouTypeReplaceHyperlinks	True
SmartTagRecognizers	System.__ComObject
NewWorkbook	System.__ComObject
SpellingOptions	System.__ComObject
Speech	System.__ComObject
MapPaperSize	True
ShowStartupDialog	False
DecimalSeparator	,
ThousandsSeparator	.
UseSystemSeparators	True
ThisCell	(ni privzete vrendnosti)
RTD	System.__ComObject
DisplayDocumentActionTaskPane	False
ArbitraryXMLSupportAvailable	True
MeasurementUnit	1
ShowSelectionFloaties	True
ShowMenuFloaties	True
ShowDevTools	False
EnableLivePreview	True

DisplayDocumentInformationPanel	False
AlwaysUseClearType	False
WarnOnFunctionNameConflict	True
FormulaBarHeight	1
DisplayFormulaAutoComplete	True
GenerateTableRefs	xlGenerateTable-RefStruct
Assistance	System.__ComObject
EnableLargeOperationAlert	True
LargeOperationCellThousandCount	33554
DeferAsyncQueries	False
MultiThreadedCalculation	System.__ComObject
ActiveEncryptionSession	-1
HighQualityModeForGraphics	False
FileExportConverters	System.__ComObject
SmartArtLayouts	System.__ComObject
SmartArtQuickStyles	System.__ComObject
SmartArtColors	System.__ComObject
AddIns2	System.__ComObject
PrintCommunication	True
UseClusterConnector	False
ClusterConnector	(ni privzete vrendnosti)
Quitting	False
Dummy22	(ni privzete vrendnosti)
Dummy23	(ni privzete vrendnosti)
ProtectedViewWindows	System.__ComObject
ActiveProtectedViewWindow	(ni privzete vrendnosti)
IsSandboxed	False
SaveISO8601Dates	True

HinstancePtr	140701804331008
FileValidation	msoFileValidationDefault
FileValidationPivot	xlFileValidationPivot-Default
ShowQuickAnalysis	True
QuickAnalysis	System.__ComObject
FlashFill	True
EnableMacroAnimations	False
ChartDataPointTrack	True
FlashFillMode	False
MergeInstances	True
EnableCheckFileExtensions	True

Literatura

[1] J. Richter, "CLR via C#, Fourth Edition", *Microsoft Press, A Division of Microsoft Corporation* str. 5–35, 2012.

[2] J. Demšar, "Python za programerje", *Fakulteta za računalnisšstvo in informatiko* str. 165–167, 2009.

[3] D. Rogerson, "Inside COM" *Microsoft Press, A Division of Microsoft Corporation* str. 3 – 13, 1997

[4] J. and B. Albahari, "C# 4.0 in a Nutshell", *O'Reilly Media, Inc* str. 115, 647 – 652, 2010.

[5] A. Troelsen, "COM and .NET Interoperability" *Apress* str. 25 – 30, 323 –334, 340 – 342, 347 – 352, 2002

[6] Data Developer Center "Entity Framework" `https://msdn.microsoft.com/en-us/data/ef.aspx` Obisk 21.05.2015

[7] J. Newkirk, A. Vorontsov, "How. NET's custom attributes affect design", *Software, IEEE*, št. 19, zv. 5, str. 18–20, 2002.

[8] MSDN Dev Center "Common HRESULT Values" `https://msdn.microsoft.com/en-us/library/windows/desktop/aa378137(v=vs.85).aspx` Obisk: 17.06.2015

[9] MSDN Develper Network "ADO.NET Architecture" `https://msdn.microsoft.com/en-us/library/27y4ybxw(v=vs.110).aspx` Obisk: 21.05.2015

[10] MSDN Develper Network "Value Types (C# Reference)" https://msdn.microsoft.com/en-us/library/s1ax56ch.aspx Obisk: 10.04.2015

[11] MSDN Developer Network "Exception Class" https://msdn.microsoft.com/en-us/library/system.exception(v=vs.110).aspx#inheritanceContinued Obisk: 17.06.2015

[12] MSDN Developer Network "GC Class" https://msdn.microsoft.com/en-us/library/system.gc(v=vs.110).aspx Obisk: 17.06.2015

[13] MSDN Developer Network "Type Safety and Security" https://msdn.microsoft.com/en-us/library/hbzz1a9a.aspx Obisk: 21.05.2015

[14] MSDN Developer Network "struct (C# Reference)" https://msdn.microsoft.com/en-us/library/ah19swz4.aspx Obisk: 2105.2015

[15] MSDN Developer Network "enum (C# Reference)" https://msdn.microsoft.com/en-us/library/sbbt4032.aspx Obisk: 2105.2015

[16] MSDN Developer Network "Integral Types Table (C# Reference)" https://msdn.microsoft.com/en-us/library/exx3b86w.aspx Obisk: 21.05.2015

[17] MSDN Developer Network "Floating-Point Types Table (C# Reference))" https://msdn.microsoft.com/en-us/library/9ahet949.aspx Obisk: 21.05.2015

[18] MSDN Developer Network "decimal (C# Reference)" https://msdn.microsoft.com/en-us/library/364x0z75.aspx Obisk: 21.05.2015

[19] MSDN Develper Network "What Is Windows Communication Foundation" https://msdn.microsoft.com/en-us/library/ms731082(v=vs.110).aspx Obisk: 12.04.2015

[20] MSDN Develper Network "Introduction to WPF" https://msdn.microsoft.com/en-us/library/aa970268(v=vs.110).aspx Obisk: 21.05.2015

[21] MSDN Developer Network "Variant Data Type" https://msdn.microsoft.com/en-us/library/office/gg251448.aspx Obisk: 27.05.2015

[22] MSDN Develper Network "Common Language Runtime (CLR)" https://msdn.microsoft.com/en-us/library/8bs2ecf4 Obisk: 10.04.2015

[23] MSDN Develper Network "A Closer Look at Platform Invoke" https://msdn.microsoft.com/en-us/library/0h9e9t7d(v=vs.71).aspx Obisk: 13.04.2015

[24] MSDN Dev Center - Desktop "DllMain entry point" https://msdn.microsoft.com/en-us/library/windows/desktop/ms682583%28v=vs.85%29.aspx Obisk: 17.04.2015

[25] MSDN Develper Network "Dynamic Language Runtime Overview" https://msdn.microsoft.com/en-us/library/dd233052%28v=vs.110%29.aspx Obisk: 13.04.2015

[26] Visual Studio Documentation .NET Frameowrk 4 "Common Language Specification" https://msdn.microsoft.com/en-us/library/vstudio/12a7a7h3(v=vs.100).aspx Obisk: 16.06.2015

[27] MSDN Develper Network "Type Safety and Security" https://msdn.microsoft.com/library/hbzz1a9a.aspx Obisk: 16.06.2015

[28] MSDN Develper Network "Dynamic Language Runtime Overview" https://msdn.microsoft.com/en-us/library/dd233052(v=vs.110).aspx Obisk: 18.06.2015

[29] MSDN Windows Dev Center "IUnknown interface" https://msdn.microsoft.com/en-us/library/windows/desktop/ms680509(v=vs.85).aspx Obisk: 17.06.2015

[30] MSDN Windows Dev Center "IUnknown::QueryInterface method" https://msdn.microsoft.com/en-us/library/windows/desktop/ms682521(v=vs.85).aspx Obisk: 17.06.2015

[31] MSDN Windows Dev Center "IUnknown::AddRef method" https://msdn.microsoft.com/en-us/library/windows/desktop/ms691379(v=vs.85).aspx Obisk: 17.06.2015

[32] MSDN Windows Dev Center "IUnknown::Release method" https://msdn.microsoft.com/en-us/library/windows/desktop/ms682317(v=vs.85).aspx Obisk: 17.06.2015

[33] C. Nagel, B. Evjan, J. Glynn, K. Watson, M. Skinner, "Professional C#4 and .NET 4", *Wiley Publishing, Inc* str. 3-16, 2010.

[34] Wikipedia "ADO.NET" http://en.wikipedia.org/wiki/ADO.NET Obisk 21.05.2015

[35] Wikipedia "Windows Forms" http://en.wikipedia.org/wiki/Windows_Forms Dostopano 21.05.2015

[36] Wikipedia "Entity Framework" http://en.wikipedia.org/wiki/Entity_Framework Obisk 21.05.2015

[37] Wikipedia "Windows Workflow Foundation" http://en.wikipedia.org/wiki/Windows_Workflow_Foundation Obisk 21.05.2015

[38] Wikipedia "Common Language Runtime" http://en.wikipedia.org/wiki/Common_Language_Runtime Obisk: 23.05.2015

[39] Wikipedia "ASP.NET" https://en.wikipedia.org/wiki/ASP.NET Obisk: 16.06.2015

[40] Wikipedia "HRESULT" `https://en.wikipedia.org/wiki/HRESULT` Obisk: 17.06.2015

[41] WIkipedia "DLL Hell' `https://en.wikipedia.org/wiki/DLL_Hell` Obisk: 17.06.2015

[42] NuGet "ILDisassembler 1.0.1-beta2" `https://www.nuget.org/packages/ILDisassembler/1.0.1-beta2` Obisk: 22.06.2015