JIHOČESKÁ UNIVERZITA V ČESKÝCH BUDĚJOVICÍCH

PEDAGOGICKÁ FAKULTA

SKRIPTOVACÍ ENGINE

BAKALÁŘSKÁ PRÁCE

Vedoucí bakalářské práce: Ing. Petr Vaněček, Ph.D.

Autor: Ladislav Karda

2007

České Budějovice

PODĚKOVÁNÍ

Děkuji vedoucímu bakalářské práce Ing. Petru Vaněčkovi, Ph.D. za metodické vedení a účinnou pomoc při zpracování bakalářské práce. Dále děkuji všem ostatním, kteří mi poskytli potřebné informace, bez nichž by tato práce nemohla vzniknout.

Všechny uvedené názvy programových produktů, firem apod. mohou být ochrannými známkami nebo registrovanými ochrannými známkami příslušných vlastníků.

PROHLÁŠENÍ

Prohlašuji, že jsem bakalářskou práci vypracoval samostatně na základě vlastních zjištění a materiálů a že jsem uvedl veškeré použité zdroje.

Prohlašuji, že v souladu s § 47b zákona č. 111/1998 Sb. v platném znění souhlasím se zveřejněním své bakalářské práce, a to v nezkrácené podobě elektronickou cestou ve veřejně přístupné části databáze STAG provozované Jihočeskou univerzitou v Českých Budějovicích na jejích internetových stránkách.

.....
Ladislav Karda

V Rudolfově dne 24. dubna 2007

ABSTRAKT

Cílem této práce je představit skriptování jako celek včetně jeho možnosti a vytvořit třídu, která umožní skriptování, tj. editaci zdrojového kódu skriptu a jeho spuštění v prostředí zabezpečeném proti softwarovým útokům. Hlavní důraz je kladen na prostředí .NET a na skriptovací jazyk Boo.

V druhé kapitole je představeno skriptování obecně; skripty jsou rozděleny do kategorií podle jejich použití. Toto rozdělení je však velice obecné, protože některé jazyky mohou patřit do více než jedné oblasti. U každého jazyka je stručný popis. Poslední podkapitola je věnována jazyku Boo, který byl zvolen pro použití v prostředí .NET.

Ve třetí kapitole je představena a popsána technologie Microsoft .NET. Jsou zde popsány její hlavní výhody a principy jejího fungování. Dotýkám se Common Language Infrastructure, Common Language Runtime i Common Intermediate Language. Dále je zde popsáno vývojové prostředí pro jazyky .NET.

Čtvrtá kapitola se zabývá praktickou realizací vytyčeného úkolu, tj. vytvořením třídy pro skriptování. První část této kapitoly ještě upřesňuje některé specifikace jazyka Boo, druhá se již věnuje popisu vytvoření třídy. Velký důraz je kladen na zabezpečení proti škodlivému kódu.

V přílohách je výpis zdrojového kódu tříd Booish, což je interaktivní interpreter jazyka Boo, a třídy ScriptHost, která zajišťuje správné a bezpečné provedení skriptu v jazycích C# a Boo. Třída ScriptHost může být použita i mimo rámec projektu GREG, je možno ji nasadit všude, kde je vyžadováno dynamické zpracování jazyka, ať už C# nebo Boo.

ABSTRACT

The target of this work is to introduce scripting as a whole including its possibilities and to create a class, which permits scripting; it means modification of source code of the script and its execution in malicious code-safe environment. .NET environment and Boo scripting language are accentuated here.

In the second chapter, there is the introduction itself. Scripts are divided into categories according to their usability. This division is just a general one because some languages may belong to more than one group. Every language is briefly described. The last subchapter is dedicated to Boo programming language which was chosen for application in .NET.

The third chapter familiarizes the reader with the Microsoft .NET technology. Advantages and principles of its behaviour are listed here. There are mentioned Common Language Infrastructure, Common Language Runtime and Common Intermediate Language. There is a characteristic of the development environment for .NET languages.

The fourth chapter deals with practical realization of the selected task; it means creation of the scripting class. The first part of this chapter defines some specifications of the Boo language and the other part pursues to explanation of the class creation. The main stress is on security against malicious code.

In the appendices, there is a listing of a source code of the Booish class – an interactive interpreter of the Boo language – and the ScriptHost class which ensures right and safe fulfilment of the script in C# and/or Boo languages. The ScriptHost class can be used even outside the GREG project; it is possible to use it everywhere where you need dynamical language processing – C# or Boo.

OBSAH

1. Úvod 6
2. Skriptování obecně 6
   a. Skripty operačního systému 8
   b. Makro jazyky 16
   c. Aplikačně specifické jazyky 17
   d. Webové skripty 18
   e. Skripty pro práci s textem 20
   f. Dynamické jazyky pro všeobecné použití 20
   g. Rozšiřující skriptovací jazyky 23
   h. Jazyk Boo 24
3. Speciality .NET 26
4. Praktická realizace 28
5. Shrnutí a závěr 35
6. Použité odkazy a literatura 37

Příloha č. 1 – Booish 38
Příloha č. 2 – Výpis zdrojového kódu třídy ScriptHost 39
Příloha č. 3 – Příklad použití třídy ScriptHost 55

1. ÚVOD

V této bakalářské práci si kladu za cíl představit několik skriptovacích jazyků, představit skriptování a prozkoumat jeho možnosti na jakékoliv platformě s důrazem na .NET. Ve druhé části se zaměřím na ukázky praktického použití skriptovacího jazyka *Boo*. Tato část by se měla, mimo jiné, zabývat také zabezpečením proti škodlivému kódu.

Hlavním přínosem této práce by měla být praktická komentovaná ukázka třídy, která umožní skriptování v prostředí jazyka C#. Tato třída by měla sloužit nejen zkušeným programátorům, ale i programátorům začátečníkům, měla by tedy být co nejvíce uživatelsky přívětivá a obsahovat co nejvíce užitečných metod pro práci se skriptem.

Toto téma jsem si vybral na základě mé účasti na projektu GREG – Graphics research group. „Skupina GREG vznikla koncem roku 2005. Hlavním cílem skupiny je vytvářet společné mezioborové projekty, které využívají možností moderní počítačové grafiky“[1]. Náš tým se věnuje projektu Stereoskopická projekce, který si klade za úkol vyvinout kompletní řešení pro učitele, kteří nejsou programátoři, pro použití stereoskopické projekce. Získané poznatky z této práce budu aplikovat do mnou vyvíjeného skriptovacího modulu tohoto projektu.

2. SKRIPTOVÁNÍ OBECNĚ

Skriptováním rozumíme psaní kódu v jazyce interpretovaném, na rozdíl od jazyků kompilovaných – např. C, C#, Java. Jazyky kompilované, jak už název napovídá, se píší ve zdrojovém kódu a ten se poté přeloží (překompiluje) do spustitelného tvaru pro danou architekturu procesoru a pro operační systém. Z toho vyplývá rychlost těchto jazyků, protože není potřeba překládat v reálném čase, ale i jejich omezení – program zkompilovaný na jedné platformě nejde spustit na platformě jiné. Pro spuštění na jiné platformě je potřeba na ní program znovu zkompilovat, aby fungoval.

[1] http://www.jcu.cz/groups/GREG/main/, cit. 16. 4. 2007

Na psaní kódu ve skriptovacím jazyce většinou postačí jednoduchý textový editor, neboť kontrolu syntaxe provádí tzv. interpreter při každém spuštění kódu.

Interpreter je program, který prochází zdrojový kód skriptu, kontroluje syntaktickou správnost jednotlivých řádků a poté je vykonává. Zajišťuje tudíž skriptu prostředí pro jeho průběh. Skript používá metody a struktury definované v interpreteru.

První skriptovací jazyky byly nazývány „dávkové programy“, například soubor autoexec.bat nebyl nic jiného, než soubor příkazů, což odpovídá definici skriptu. Velké využití mají skriptovací jazyky při zpracování webových stránek, kde se hlavně používá PHP, JavaScript, Perl, Python a další. Další široké uplatnění skriptů jsou konfigurační soubory, které se také dají považovat za skripty.

V této bakalářské práci se budu hlavně zabývat skriptovacím jazykem Boo, který je asi jediný, který je implementovatelný do prostředí .NET.

Hlavním cílem skriptování je izolace uživatele od spletitosti celého kódu a od používání vývojových nástrojů určených pro zcela odlišnou skupinu lidí. Velikým přínosem skriptování je také zpřístupnění programování pro jinak úzce specializované profese, např. techniky, účetní, fyziky atd. Obrovskou výhodou skriptování je rychlost psaní kódů a jejich úspornost, na druhé straně jako daň za efektivitu psaní je neefektivní průběh kódu. Tento problém se vyvažuje s nárůstem platů programátorů a snižováním cen hardwaru.

Skriptovací jazyky se neomezují pouze na pole působnosti, pro které byly napsány, ale je v nich možno realizovat i zcela odlišný typ programu.

Tyto jazyky se dají rozdělit podle prostředí, ve kterém se používají, na skripty operačního systému, makro jazyky, aplikačně specifické jazyky,

webové skripty, skripty pro práci s textem, dynamické jazyky pro všeobecné použití a rozšiřující skriptovací jazyky. Popis jazyků byl inspirován [2] a [3].

a. SKRIPTY OPERAČNÍHO SYSTÉMU

Na počátku bych rád předeslal, že pod slovem Unix jsou zahrnuty oba operační systémy, jak Unix, tak i Linux.

Tyto skripty vznikly pro umožnění kontroly spouštění systémových programů. Některá skriptovací prostředí fungují jako příkazové řádky a interpretery skriptů – *sh*, *bash*, *command.com*. Do této kategorie náleží zejména zmiňované unixové *bash* a *sh*, dále *command.com* z dob MS-DOSu, nynější *cmd.exe* a jeho nástupce *Windows PowerShell*.

Termínem shell se obecně myslí jakýkoliv program, do kterého se dají psát příkazy. Od dob, kdy si uživatelé Unixu mohou vybrat, jaký shell budou používat, se vžilo označení pouze shell, nikoliv celé jméno interpreteru. Název shell je z angličtiny, znamená skořápka, což značí obalení funkcí operačního systému právě tímto interpreterem. Uživatel tedy ani nemusí tušit detaily daného operačního systému, neboť se k nim nedostane přímo, ale prostřednictvím tohoto shellu. Může se též vyskytovat označení „grafický shell“, což znamená grafickou nadstavbu shellu – v Unixu např. KDE či GNOME.

Interpreter *sh* je původním shellem Unixu. Pokud se mluví o *sh*, myslí se Bourne shell. Tento shell se v počátcích Unixu stal standardem – každý unixový systém měl shell, který byl kompatibilní se *sh*. V každém systému je v /bin/sh buď přímo kód, nebo alespoň odkaz na podobný shell. *Sh* byl v počátcích nezvyklý, protože obsahuje jazyk k nastavování systému pomocí skriptů a také obsahuje příkazovou řádku.

Zkratka *bash* vznikla z Bourne-again shell. Jedná se o GNU variantu *sh*. *Bash* byl vytvořen roku 1978 Brianem Foxem. *Bash* je základním shellem pro většinu linuxových distribucí, stejně jako pro Mac OS X. Tento interpreter byl také portován do Windows jako CYGWIN a do MS-DOSu jako součást projektu DJGPP – vývojové prostředí pro C, C++, Objektové C, Ada a Fortran pro MS-DOS.

Jazyk shellu *bash* je rozsáhlejší než shellu *sh*, proto je možno drtivou většinu kódů pro *sh* spustit bez úprav i v *bash*. Jediné kódy, které se musí

upravit, jsou ty, které používají vestavěné funkce *sh* nebo speciální proměnné. *Bash* podporuje spoustu pokročilých funkcí z Korn shell a z C shell, např. editace příkazové řádky, historie příkazů a další. *Bash* má jednu ohromnou výhodu – po stisknutí klávesy TAB doplní názvy přijatelných souborů.

Základní schopnosti *bash* jsou veliké – je možno ho používat jako kalkulačku, nechat přesměrovat standardní vstupy a výstupy a běžně pracuje s regulárními výrazy. Z těchto oblastí je možno sestavit množství skriptů – takové startovací skripty jsou ukázkou. Například jeden skript na zjištění, zdali pracujeme na baterie, či nikoliv a pokud pracujeme, sníží frekvenci procesoru, aby notebook vydržel pracovat co nejdéle. Skript převzat z http://forum.ubuntu-fr.org/viewtopic.php?pid=746272 dne 23. 4. 2007:

```
#!/bin/bash

# Zjistíme stav baterie
PRESENT="`cat /proc/acpi/battery/BAT1/state | grep
present: | cut -f2 -d:`"

if [ $PRESENT = "yes" ]; then
  CHARGING_STATE="`cat /proc/acpi/battery/BAT1/state
  | grep "charging state:" | cut -f2 -d:`"
else
  CHARGING_STATE="empty"
fi

# Opět zjistíme stav baterie
if [ $PRESENT = "yes" -a $CHARGING_STATE =
"discharging" ]; then
# Případně snížíme frekvenci procesoru
  cpufreq-selector -g powersave
else
  cpufreq-selector -g ondemand
fi
```

Shell pro MS-DOS a Windows 95, 98, 98 SE a ME se nazývá *command.com*. Mimo své funkce interpreteru skriptovacího jazyka a příkazové řádky měl ještě jednu funkci – nastavil prostředí operačního systému zavedením dávky *autoexec.bat* (byl to první program zavedený do paměti počítače po bootu) a byl prvním předkem všech spuštěných programů.

Stejně jako *sh* nebo *bash,* měl *command.com* také dva režimy – interaktivní, ve kterém se psaly příkazy, které byly ihned vykonávány,

a skriptovací, tzv. dávkový, ve kterém se spouštěly sekvence příkazů uložené v textovém souboru s příponou .bat. Na rozdíl od *sh* a *bash* nerozlišuje malá a velká písmena v příkazech. Takže například rmDir je stejné jako RmDir.

Command.com je hlavně interpreter příkazů operačního systému, proto obsahuje několik příkazů pro práci se soubory. Jsou to především tyto:

- soubor.exe – spuštění programu, stačí pouze „soubor“
- c: – změna aktivního disku
- cd, chdir – zobrazí či změní aktuální adresář
- copy – kopíruje soubory (pokud cílový soubor existuje, systém se zeptá, zda ho má přepsat)
- del, erase – maže soubory či adresáře. Při použití na neprázdný adresář vymaže pouze soubory v něm a neprochází rekurzivně vnořené adresáře
- dir – vypíše seznam všech souborů v adresáři
- label – ukáže a umožní změnit jmenovku jednotky
- md, mkdir – vytvoří adresář
- ren, rename – přejmenuje soubor nebo adresář
- rd, rmdir – odstraní prázdný adresář
- type – vypíše obsah souboru na standardní výstup
- verify – zobrazí či změní stav kontroly zápisu souborů na disk (při chybách na médiích bylo občas nutno tuto kontrolu vypnout a potom se data většinou dala úspěšně zkopírovat)
- vol – zobrazí informace o jednotce

Dále obsahuje spoustu dalších příkazů pro práci s řetězci a obrazovkou:

- break – nastavuje ovládání přerušení Ctrl+C
- cls – vymaže obrazovku
- chcp – ukáže či změní aktuální kódovou stránku
- date – nastavuje systémové datum
- echo – nastavuje, zdali se budou zobrazovat příkazy na obrazovku (echo on/off), nebo posílá text na standardní výstup (echo text)
- lh, loadhigh – načte program do horní paměti
- lock – umožní externím programům nízkoúrovňový přístup k jednotkám (pouze Windows 95, 98 a ME)
- path – zobrazí nebo změní proměnnou prostředí PATH, která určuje, kde bude *command.com* hledat spustitelné programy
- pause – pozastaví průběh programu a vyčká na stisknutí klávesy

- prompt – změní proměnnou prostředí PROMPT a tím i vzhled začátku příkazové řádky (běžně se používá prompt pg, což značí, že příkazová řádka bude začínat `C:\>`)
- set – nastaví proměnnou prostředí, její jméno a hodnotu. Bez argumentů vypíše všechny proměnné prostředí
- time – nastaví systémový čas
- unlock – zakáže nízkoúrovňový přístup k disku (pouze Windows 95, 98 a ME)
- ver – zobrazí verzi operačního systému

Command.com obsahuje i kontrolní struktury pro běh programu:

- :návěští – definuje návěští pro skok příkazem goto
- for – opakování příkazu pro určitou množinu souborů
- goto – přesune výkon kódu na dané návěští
- rem – zakomentování řádky kódu
- if – podmínka, umožňuje větvení programu
- call – pozastaví výkon běžícího programu, zavolá jiný dávkový program a po ukončení pokračuje v původním programu
- exit – ukončení *command.com* a navrácení ke spouštěcímu programu
- shift – posouvá parametry příkazové řádky o jedničku – %1 za %2 atd.

Dále obsahuje ještě speciální proměnné:

- ERRORLEVEL – obsahuje návratovou hodnotu z intervalu 0 až 255 posledního ukončeného programu, který tuto hodnotu vrací. Existuje jistá konvence, že úspěšně ukončený program vrací nulu. Pokud program hodnotu ERRORLEVEL nevrací, zůstane v ní poslední vrácená hodnota.
- proměnné prostředí – mají tvar %jméno% a jsou spojeny s proměnnými nastavenými příkazem set
- parametry příkazové řádky – tyto proměnné mají tvar %0 až %9 a označují jednotlivá slova na příkazové řádce. Např. `muj.bat Petr a Pavel` bude obsahovat %0 = muj.bat, %1 = Petr, %2 = a, %3 = Pavel. Parametry s číslem větším než devět mohou být namapovány příkazem shift.
- proměnné příkazu for – použity ve smyčkách, mají tvar %%a a mají platnost pouze pro danou smyčku

Přesměrování standardního vstupu a výstupu:

- příkaz < soubor – přesměrování standardního vstupu ze souboru nebo zařízení
- příkaz > soubor – přesměrování standardního výstupu, soubor je přepsán, pokud existuje
- příkaz ≫ soubor – přesměrování standardního výstupu, k souboru je připsáno na konec, pokud existuje
- příkaz1 | příkaz2 – přesměruje standardní výstup příkazu1 do standardního vstupu příkazu2

Jako malou demonstraci bych rád předvedl dávkový soubor, který kopíruje neomezený počet souborů v parametru do jednoho adresáře:

```
@echo off
rem ALLCOPY.BAT kopíruje neomezené množství souborů
rem do jednoho adresáře.
rem Příkaz používá následující syntaxi:
rem mycopy adresář soubor1 soubor2 ...
set todir=%1
:getfile
shift
if "%1"=="" goto end
copy %1 %todir%
goto getfile
:end
set todir=
echo Hotovo.
```

Nástupcem *command.com* je *cmd.exe*, proto bych se rád věnoval i tomuto programu. Nalézá se ve Windows 2000, NT, XP a 2003 a OS/2. Je právoplatným nástupcem *command.com*, protože nabízí rozšířenou funkčnost. Některé příkazy, které byly původně externí (např. deltree.exe), jsou implementovány do *cmd.exe* prostřednictvím přepínačů (v našem případě RD /s). Některé starší MS-DOSové příkazy nejsou podporovány – např. lock, unlock a lh. Některé funkce, jako např. break, jsou sice *cmd.exe* podporovány, nicméně jsou přítomny pouze z důvodu zpětné kompatibility a nekonají nic, protože tuto činnost již nativně provádí operační systém.

Hlavní rozdíl mezi *cmd.exe* a *command.com* je, že *command.com* je MS-DOSový program, takže může používat pouze obecné funkce. *Cmd.exe* je

nativní v obou verzích, což mu dovoluje používat specifické vlastnosti platformy.

Cmd.exe přinesl některé nové příkazy:

- setlocal – začne pracovat v lokální oblasti, stejně jako např. příkaz for začíná svou vlastní oblast
- endlocal – ukončí práci v lokální oblasti a zruší všechny změny, které proběhly mezi setlocal a endlocal a nechává tak celé prostředí netknuté
- exit – ukončí interpreter

Obsahuje spoustu předdefinovaných proměnných prostředí, z nichž každá může být až 8192 bytů dlouhá [7]:

- %ALLUSERSPROFILE% – vrací cestu k profilu všech uživatelů
- %APPDATA% – vrací cestu k úložišti dat aplikací
- %CD% – vrací aktuální adresář
- %CMDCMDLINE% – vrací přesnou cestu k programu, který spouštěl daný shell
- %COMPUTERNAME% – vrací jméno počítače
- %COMSPEC% – vrací přesnou cestu k příkazovému interpreteru
- %DATE% – vrací aktuální datum
- %HOMEDRIVE% – vrací označení jednotky s uživatelovým domovským adresářem
- %HOMEPATH% – vrací cestu k uživatelskému profilu
- %HOMESHARE% – vrací cestu k uživatelskému síťovému sdílenému adresáři
- %LOGONSERVER% – vrací cestu k doménovému řadiči, který povolil tuto přihlašovací relaci.
- %NUMBER_OF_PROCESSORS% – vrací počet instalovaných procesorů v počítači
- %OS% – vrací název operačního systému, např. Windows_NT
- %PATH% – vrací adresáře, ve kterých se hledají spustitelné soubory
- %PATHEXT% – vrací přípony spustitelných souborů
- %PROCESSOR_ARCHITECTURE% – vrací architekturu procesoru, buď x86, nebo IA64
- %PROCESSOR_IDENTIFIER% – vrací popis procesoru, např. x86 Family 6 Model 8 Stepping 1, AuthenticAMD
- %PROCESSOR_LEVEL% – vrací modelové číslo procesoru – pro náš procesor vrací hodnotu 6

- %PROCESSOR_REVISION% – vrací číslo revize procesoru
- %PROMPT% – vrací tvar počátku příkazové řádky, standardně je PG – cesta následována znakem „>“
- %RANDOM% – vrací náhodné číslo generované *cmd.exe* z intervalu nula až 32767
- %SYSTEMDRIVE% – vrací označení jednotky, která obsahuje systémové soubory Windows
- %SYSTEMROOT% – vrací cestu ke kořenovému adresáři Windows
- %TEMP% a %TMP% – obě proměnné vrací cestu k adresáři pro dočasné soubory. Některé aplikace vyžadují TEMP, jiné zase TMP.
- %TIME% – vrací aktuální systémový čas
- %USERDOMAIN% – vrací název domény, do které patří aktuálně přihlášený uživatelský účet
- %USERNAME% – vrací uživatelské jméno aktuálně přihlášeného uživatele
- %USERPROFILE% – vrací cestu k profilu aktuálně přihlášeného uživatele
- %WINDIR% – vrací cestu k adresáři operačního systému

Nástupcem *cmd.exe* má být v letošním roce *Windows PowerShell*, dále jen „*WPS“*. Tento má být implementován ve Windows Vista, ovšem pro zpětnou kompatibilitu má být obsažen i *cmd.exe*.

WPS je navržen na základě modelu objektového programování a s podporou Microsoft .NET Framework. Pro instalaci *WPS* je nutná přítomnost .NET Framework ve verzi 2.0. *WPS* je podporován na Windows XP, Windows 2003 Server a Windows Vista.

Microsoft navrhnul *WPS*, aby mohl čelit skriptovatelné automatizaci z prostředí unixových systémů, kde byl od počátku základním stavebním kamenem příkazový řádek a veškeré grafické rozhraní jako by obalovalo pouze funkce příkazového řádku. Kritika Windows byla zaměřena na fakt, že Windows jsou plně konfigurovatelné pouze za použití grafického rozhraní, nikoliv však použitím příkazové řádky. Microsoft slíbil, že do budoucna bude stavět stejně jako Unix na skriptování a grafické rozhraní bude pouze používat příkazy *WPS*.

Velkou zajímavostí je původ kódového označení *WPS*, které zní Monad. Je inspirováno dílem Gottfrieda Wilhelma Leibnitze Monadologie. V tomto díle

autor tvrdí, že celý Vesmír je složen ze základních částeček – monád. Stejně tak *WPS* sestává z množství komponent.

WPS se zřejmě poučil z unixových shellů a již zavedl aliasy na příkazy, tj. zkrácená jména příkazů, např. pro vypsání všech proměnných prostředí je možno zavolat `Get-ChildItem –path env:` nebo za použití aliasu jenom zkráceně `gci –path env:`. *WPS* stále nerozlišuje malá a velká písmena ve jménech příkazů.

Největší předností *WPS* je jistě jeho rozšiřitelnost. Uživatel si může naprogramovat své vlastní programy, nazývané „cmdlets“, a poté je propojit s *WPS*. Tyto moduly jsou kompilované stejně jako jiné nástroje implementované do jakýchkoliv rozhraní. Tímto systémem můžeme přidat do *WPS* nové příkazy i prostředí. Cmdlets jsou vlastně třídy .NET navržené pro integraci do příkazové řádky. Hlavní rozdíl mezi unixovými shelly a *WPS* spočívá v předávání informací mezi cmdlets. Ve světě Unixu se toto děje pomocí předávání textových informací, *WPS* si předává informace ve formě objektů. Při volání samostatného cmdletu je jeho výstup formátován na text, ale při předávání informací dál se tyto informace konvertují na nejpříhodnější formát pro vstup do dalšího cmdletu. Tento přístup eliminuje potřebu utilit pro práci s textem, jako např. v Unixu `grep` nebo `awk`, a umožňuje zpracovávání informací za běhu bez nároků na složitost programování skriptů, například při výpisu procesů je možno poté zavolat na jakémkoliv objektu procesu jeho definované metody bez nutnosti předávat vlastnosti externím programům.

Mezi hlavní výhody *WPS* bych jistě zařadil následující:

- skriptování ve stylu C# s podporou mnoha pokročilých funkcí, jako např. slovníky (hash table), příkaz switch, regulární výrazy, anonymní metody a dělení polí. Standardní kontroly běhu programu (for/if) jsou samozřejmostí.
- všechny cmdlets dědí určité vlastnosti a tak dovolují uživateli rozhodnout, jak se budou chovat při chybách a jaká bude míra jejich interakce s okolím. Např. podporují možnosti –WhatIf, resp. –Confirm, které informují, co by provedly, ale nic nevykonají, resp. čekají na potvrzení každé akce.
- pozastavit chybující program (to je jedna z možností, jak se vypořádat s chybou) a v nové instanci *WPS* zjistit, kde se stala chyba a poté nechat program pokračovat

- *WPS* přináší politiky spouštění, kterými se snaží více zabezpečit spouštění skriptů; politiky mají čtyři varianty: Restricted (Omezené), AllSigned (Všechny podepsané), RemoteSigned (Vzdálené podepsané) a Unrestricted (Neomezené).
- parametry příkazové řádky jsou celá slova, ale mohou být zkráceny na minimální počet písmen, která jednoznačně odlišují jednotlivé možnosti od sebe
- doplňuje názvy příkazů a souborů po stisknutí klávesy TAB, což byla nejočekávanější funkce. *Command.com* neuměl žádné doplňování po stisknutí klávesy TAB, *cmd.exe* umí doplňovat pouze názvy souborů a *WPS* umí doplňovat i názvy příkazů. Velice to urychluje práci a minimalizuje možnost překlepů.
- možnost přiřadit výstup nějakého příkazu do proměnné, která poté bude objekt, nebo pole objektů a může být prověřena

V uvedeném příkladu na zjištění všech systémových proměnných je vidět použití prostředí, která máme k dispozici. Pokud bychom se ptali na jakoukoliv jinou oblast, použili bychom jedno z následujících prostředí: Alias, Environment, Filesystem, Function, Registry, Variable nebo Certificate.

Pro příklad bych uvedl pár příkazů, které mají veliký význam v operačním systému. Není nutno spouštět Správce úloh, ale je možno vše provést z příkazové řádky.

Pro nalezení a ukončení všech procesů od písmene p, které zabírají ve stránkovacím souboru více než 50 MB stačí příkaz `get-process p* | where {$_.VM -gt 50MB} | stop-process`. Pro čekání na ukončení určitého procesu postačí zavolat dvě řádky:

```
$process = get-process notepad
$process.WaitForExit()
```

b. MAKRO JAZYKY

Hlavním použitím makro jazyků je přiřazení funkcí grafického uživatelského rozhraní (GUI) klávesnici. Takto je možno obsluhovat všechna systémem generovaná okna, menu, tlačítka apod. Tento typ jazyků vzniknul hlavně pro urychlení a automatizaci opakovaných úkonů v GUI.

Mezi tento typ jazyků patří např. *AutoHotKey*, *AutoIt*, *Expect* a *Automator*.

c. APLIKAČNĚ SPECIFICKÉ JAZYKY

Tyto jazyky nejvíce vyhovují uživatelům daného aplikačního systému a respektují jejich zvyklosti. Stejně tak mnoho počítačových her se dá nastavit přímo z konzole ve hře nebo v konfiguračních souborech, které jsou vlastně také skripty. Skriptování se také používá k naprogramování chování nehráčských postav a herního prostředí. Jazyky tohoto typu jsou napsány přímo a jednoúčelově pro dané aplikace, a jakkoliv mohou připomínat běžné programovací jazyky, obsahují speciální vlastnosti, kterými se od nich odlišují. Mezi tyto jazyky by se daly zařadit následující:

- *Action Code Script* – jazyk používaný ve hře HeXen, je podobný C, ale méně flexibilní
- *ActionScript* – původně vyvinut pro tvorbu softwaru v prostředí Flash, používá se k „oživení“ animací či pro přidání interaktivity
- *AutoLISP* – odrůda jazyka Lisp, používaná v AutoCADu. Většina základních funkcí je pro práci s grafikou nebo manipulaci s objekty v AutoCADu
- *Emacs Lisp* – také odrůda jazyka Lisp, implementován v textových editorech GNU Emacs a XEmacs
- *Game Maker Language* – skriptovací jazyk vyvinutý pro použití s aplikací na tvorbu her nazvanou Game Maker. Tento jazyk je podobný C++, ale na rozdíl od C++ neočekává středníky na konci příkazu. Pokud tam má být, ale není, *Game Maker Language* si ho doplní.
- *HyperTalk* – vyvinut hlavně pro začínající programátory; programy napsané v *HyperTalku* vypadají jako psaná angličtina a logické struktury jsou podobné Pascalu.
- *IPTSCRAE* – skriptovací jazyk pro grafický chatovací systém The Palace
- *IRC script* – skriptovací jazyk používaný v IRC klientech ke zrychlení psaní příkazů. Např. „/j kanál“ místo „/join #kanál“. Vzniká tak bezpečnostní díra do systému.
- *JScript* – jazyk vytvořený Microsoftem, interpretuje ho stejně jako *VBScript* Windows Script Host
- *Lingo* – název *Lingo* používá několik na sobě nezávislých skriptovacích jazyků, nejznámější verze se používá v Macromedia Director. Zajímavostí je ochranná známka – po celém světě jí vlastní

Macromedia, ve Spojeném království Velké Británie a Severního Irska značku vlastní Linn micro.

- *LotusScript* – obměna BASICu používaná v Lotus Notes a v dalších produktech IBM Lotus. Tento skript je velice podobný s Visual Basicem, je možno dokonce pouze kopírovat kód mezi sebou.
- *Maya Embedded Language* – jazyk vestavěný do 3D grafického programu Maya. Jazyk je syntakticky podobný *Perlu*.
- *mIRC script* – skriptovací jazyk integrovaný do IRC klienta mIRC. Hlavní použití má pro psaní automatizovaných robotů, tzv. botů, her a dalších aplikací pro mIRC.
- *NWScript* – jazyk vyvinutý firmou BioWare pro hru Neverwinter Nights
- *QuakeC* – jazyk vyvinutý v roce 1996 Johnem Carmackem z id Software k programování částí hry Quake. S přispěním tohoto jazyka může programátor upravit celou hru, např. přidáním zbraní, jejich úpravou, úpravou prostředí a přidáním složitých objektů a terénů.
- *UnrealScript* – další z rodiny herních skriptů – používá se na ovládání Unreal Enginu. Jedná se o objektově orientovaný jazyk podobný Javě.
- *Visual Basic for Applications* – zkráceně *VBA*, jedná se o implementaci Visual Basicu vestavěnou do většiny aplikací z balíku Microsoft Office. Dále je obsažen i v balíku Office pro Mac OS. Na rozdíl od Visual Basicu je omezen spouštěním kódu pouze ze svého aplikačního prostředí, nemůžeme v něm tedy vytvořit samostatnou aplikaci.
- *VBScript* – zkratka pro *Visual Basic Scripting Edition*, česky Skriptovací vydání Visual Basicu, jazyk z rodiny Active Scripting, dříve ActiveX Scripting, interpretován Windows Script Host – součást Windows odpovědná za interpretování různých skriptů. Tento jazyk se stal nechvalně známým po útocích viru I love you, který byl napsán ve *VBScriptu*.
- *ZZT-oop* – jazyk vyvinul Tim Sweeney pro svou hru ZZT. Je to objektově orientovaný jazyk více zaměřený na jednoduchost než na flexibilitu. Jediný typ proměnných, které se ve hře mohou nalézat, je příznak typu boolean, což ztěžuje jakoukoliv aritmetiku.

d. WEBOVÉ SKRIPTY

Jazyky patřící do této skupiny jsou zaměřeny na generování dynamických webových stránek. Tyto jazyky jsou specializovány na webové aplikace, jako

např. internetové bankovnictví, zpracování formulářů na internetu, nástroje na posílání e-mailů z prostředí webu atd. Většina webových programovacích jazyků je dostatečně použitelná i na běžné programování. Na straně serveru běží tyto jazyky:

- *ColdFusion Markup Language* – dále *CFML*, skriptovací jazyk aplikačního serveru od firmy Macromedia, nyní Adobe. Tento server také podporuje výše zmiňovaný *ActionScript*. *CFML* se podobá HTML, také obsahuje tagy a nemusí být striktně dodržováno vnoření jednotlivých tagů navzájem a jejich ukončování.
- *Lasso* – spojuje interpretovaný propojující programovací jazyk se serverem pro vývoj internetových aplikací spojujících web a databáze. *Lasso* je velice dobře rozšiřitelný, může se do něj přidat podpora práce s obrázky, tvorba PDF, práce s e-maily a jejich posílání. Také obsahuje podporu pro integraci mnoha standardů, např. XML, SOAP, Java EE, Java Beans a WSDL. Vývojáři mohou dopsat své vlastní rozšiřující tagy v Javě nebo v jazyce C.
- *MIVA Script* – využívá se hlavně pro internetové aplikace jako je elektronické bankovnictví a obchodování. Ačkoliv je hojně podporován hostingovými firmami, je mnohem méně populární než jeho hlavní konkurent *PHP*.
- *PHP* – původně vyvinut pro tvorbu dynamických webových stránek, avšak postupem času přibyla další rozšíření – skriptování na příkazové řádce nebo použití v samostatných grafických aplikacích. Skriptovací engine *PHP* zpracovává *PHP* kód jako vstup a jako výstup tvoří webové stránky. *PHP* je velice rozšířený, na internetu je podle php.net přes 19 milionů domén, které běží na serverech, kde je *PHP* instalován.
- *SMX* – skriptovací jazyk původně dodávaný s Internet Factory's Commerce Builder – serverovým programem zaměřeným na tvorbu aplikací pro elektronické obchodování

Na straně klienta pracují tyto jazyky:

- *JavaScript* – objektově orientovaný skriptovací jazyk pocházející od Brendana Eicha ze společnosti Netscape. Ačkoliv má v názvu slovo Java, s Javou má pouze podobnou syntaxi a tím veškerá podobnost končí. Jazyk je používán většinou pro efekty na webových stránkách, různá informační okna, validace formulářů před odesláním, otevírání nových oken určité velikosti a změna textu ve stavovém řádku

prohlížeče. Spolu s XSLT běží na straně klienta, tj. až po načtení stránky ze serveru. Další použití *JavaScriptu* je nastavování vlastností v programech společnosti Mozilla – Thunderbird a Firefox mají veškeré nastavení psané v jazyce *JavaScript*.

- *XSLT* – zkratka pro *Extensible Stylesheet Language Transformation*, používá se pro transformace XML dokumentů do jiných XML dokumentů. Původní soubor zůstane nezměněn, vytvoří se nový výstupní soubor. *XSLT* vyvíjí konsorcium W3 od roku 1998.

e. Skripty pro práci s textem

Zpracování textových informací je jedno z nejstarších použití skriptovacích jazyků. Většina z nich byla navržena na pomoc systémovým administrátorům při správě textových konfiguračních souborů. Do této kategorie jazyků můžeme zařadit:

- *AWK* – název vznikl jako počáteční písmena příjmení autorů. *AWK* je příkladem jazyka, který velice využívá datového typu řetězec, asociativních polí (tj. polí s indexy typu string, neboli slovník) a regulární výrazy. Verze jazyka *AWK* je dnes dodávána s každou distribucí Unixu a je považován za povinnou součást Unixu.
- *Perl* – dynamický programovací jazyk vytvořený Larry Wallem a poprvé představený roku 1987. *Perl* spojuje vlastnosti jazyka C, *AWK*, *sed* a *Lisp* a shellu *sh*. *Perl* je textově zaměřený jazyk pro všeobecné použití. Jazyk je zaměřen především na funkčnost výměnou za krásu a velikost zdrojového kódu.
- *sed* – zkratka vznikla ze Stream Editor, jedná se o výkonný program používaný na aplikování textových transformací na spojitý tok dat, anglicky stream. Výhodou *sedu* je čtení po jednotlivých řádkách a tím je schopen editovat i opravdu veliké soubory, protože je nemusí celé nahrávat do paměti.

f. Dynamické jazyky pro všeobecné použití

Některé jazyky byly vyvinuty k jednomu účelu, ale později byly vylepšeny a byly jim přidány i další všeobecné funkce. Do této kategorie patří již zmiňovaný Perl. Některé z těchto jazyků byly označeny jako skriptovací jazyky, ale je možno v nich tvořit i samostatné aplikace, ovšem jejich uživatelé je „skriptovacími jazyky“ nenazývají. Do této skupiny můžeme zařadit:

- *APL* – jazyk založený na myšlence Kennetha E. Iversona z roku 1957. Nejvíce kritizovaná a zároveň vychvalovaná vlastnost *APL* je použití zvláštní znakové sady, aby i opticky vyjadřoval své kroky.
- *Boo* – tomuto jazyku bude věnován samostatný prostor a bude popsán dále
- *Dylan* – jazyk, který obsahuje jak funkční, tak i objektově orientované programování. Podporuje vícenásobnou dědičnost, polymorfismus, prohlížení objektů a další pokročilé funkce. Vznikl na počátku devadesátých let minulého století ve skupině vedené Apple Computer. Apple ukončil vývoj v roce 1995.
- *Ferite* – objektově orientovaný programovací jazyk, který byl ovlivněn některými jinými jazyky. Objekty převzal z C++, funkce z C, jmenné prostory z C++ a další. *Ferite* vytvořil v roce 2000 Chris Ross jako jazyk pro rychlý vývoj a vydání aplikací, které jsou „čisté“ a nenáročné na údržbu.
- *Groovy* – objektově orientovaný jazyk pro platformu Java jako alternativa k programovacímu jazyku Java, má vlastnosti podobné jako *Ruby, Perl, Python* a *Smalltalk*. Program napsaný v *Groove* je překompilován do Java Virtual Machine bytecode.
- *Io* – čistě objektově orientovaný jazyk, kde všechny proměnné jsou objekt. Jazyk používá dynamické typování. *Io* kód se spouští přes malý virtuální stroj.
- *Lisp* – původně specifikován v roce 1958, takže je druhý nejstarší, hojně používaný vyšší jazyk; pouze *Fortran* je starší. Brzy se stal oblíbeným jazykem pro vývoj na poli umělé inteligence. Celý zdrojový kód *Lispu* je seznam a spojový seznam je nejvýznamnější typ v celém jazyce. *Lisp* přinesl mnoho průkopnických myšlenek, jako např. dynamické typování, objektově orientované programování nebo stromové struktury.
- *Lua* – jednoduchý a procedurální jazyk, v používání od roku 1993, obsahuje definice pouze pro pár datových struktur – logická hodnota, číslo a řetězec – všechny ostatní struktury se musí vytvořit pomocí vestavěného typu „tabulka“. Jazyk je použit v mnoha komerčních aplikacích – např. hra Escape from Monkey Island od LucasArts, World of Warcraft, Baldur's Gate a další.
- *MUMPS* – zkratka *Massachusetts General Hospital Utility Multi-Programming Systém*, nebo jenom zkráceně *M*. Původně byl tento jazyk vyvinut koncem 60. let minulého století pro lékařský průmysl,

byl navržen pro snadné psaní databázových aplikací a pro efektivní využití výpočetního výkonu. Původní implementace byla interpretovaná, nynější je však již kompilovaná.

- *newLISP* – open source verze *Lispu*. Vývoj tohoto jazyka započal v roce 1991, jeho cílem je alokace minimálního množství zdrojů pro rychlou, výkonnou a meziplatformní verzi *Lispu*.
- *Python* – hybridní jazyk zveřejněný v roce 1991. Umožňuje tvorbu jak malých skriptů, tak i rozsáhlých aplikací včetně GUI. „Hybridní" zde znamená, že používá nejen objektově orientované, ale i procedurální a v omezené míře i funkcionální paradigma. *Python* se dá velice snadno vložit do jiných aplikací, kde poté slouží jako jejich skriptovací jazyk. Jedná se o jazyk dynamicky typovaný, který používá přístup tzv. „duck typing", tedy „Chodí-li to jako kachna a kváká-li to jako kachna, potom to musí být kachna." Velice slibně vypadá implementace jazyka *Python* pro .NET/Mono nazvaná *IronPython*. Je to tedy již druhý skriptovací jazyk napsaný pro .NET/Mono, tím prvním je *Boo*, o kterém se budu zmiňovat dále. Nevýhodou je poměrná nevyzrálost systému a fakt, že *IronPython* vyvíjí Microsoft a z toho vyplývající horší dostupnost zdrojového kódu, než je tomu u jiných variant tohoto jazyka.
- *Ruby* – vývoj tohoto jazyka započal v roce 1993. Autorem je Yukihiro Matsumoto, který hledal dostatečně rychlý a objektový jazyk. *Perl* se mu nezdál dostatečně výkonný a *Python* dostatečně objektový. Z tohoto vyplývá, že v *Ruby* je všechno objekt, i řetězec (tzn. můžeme nad ním zavolat nějakou metodu). *Ruby* je všeobecně použitelný skriptovací jazyk, jsou v něm napsány i některé aplikace. Podporou regulárních výrazů je předurčen ke zpracování textu. Mezi hlavní výhody jazyka *Ruby* patří zejména jednoduchá a snadno naučitelná syntaxe, plná podpora objektového programování, dynamické typování, regulární výrazy a další. Mezi nevýhody by se dala zřejmě zařadit nízká rychlost (jedná se o interpretovaný jazyk), nedostatečná česká dokumentace (donedávna i nedostatek anglické dokumentace) a závislost na interpreteru (jaký bude interpreter, takový bude běh skriptu). Nejvíce mě v tomto jazyce překvapilo vícenásobné přiřazení na jednom řádku: `a, b = b, a + b`.
- *Scheme* – jedná se o dialekt jazyka *Lisp*, ovšem oproti němu se snaží o minimalismus, proto má poslední reference jazyka pouhých 50 stran. Jazyk byl vyvinut v roce 1970. Podle wikipedia.org je jazyk

nepoužitelný pro praxi, používá se především pro výuku programování algoritmů.

- *Smalltalk* – silně objektově orientovaný, dynamicky typovaný jazyk, počátek vývoje je v roce 1969. *Smalltalk* je velice dynamický jazyk, který reflexí dokáže za běhu k sobě přidávat či měnit přidané metody a moduly. Jedná se o velice minimalistický jazyk, který zná pouze pět klíčových slov – `true`, `false`, `nil`, `self` a `super`. Komunita kolem *Smalltalku* vyvinula extrémní programování[2]. Silné orientování na objekt je ilustrováno faktem, že i blok kódu může být objekt.
- *SuperTalk* – jazyk velice kvalitního vývojového prostředí běžícího pod Mac OS a Mac OS X, poprvé vytvořen roku 1989. Vývoj verze pro Windows byl ukončen těsně před jejím vydáním.
- *Tcl* – skriptovací jazyk, který vznikl v roce 1988 jako jazyk, který bude jednoduchý se naučit, ale zároveň bude velice mocný v těch správných rukách. Jazyk je nezávislý na platformě, vše je možno dynamicky předefinovat a přetížit, plně podporuje Unicode, je možno ho použít v různých prostředích – jako textový skriptovací jazyk, jako jazyk podporující GUI pro různé aplikace a jako vnitřní jazyk různých webových aplikací a databází (PostgreSQL).
- *Revolution* – je jazyk prodávaného meziplatformního prostředí pro rychlé vyvíjení aplikací (Runtime Revolution). *Revolution* je dynamicky typovaný jazyk podobný angličtině. Jazyk podporuje pokročilé funkce, jako např. regulární výrazy, podpora QuickTime, přístup k databázím a podporu TCP/IP.

g. ROZŠIŘUJÍCÍ SKRIPTOVACÍ JAZYKY

Některé jazyky byly navrženy, pro použití v různých aplikacích jako jejich vnitřní skriptovací jazyk. Tyto jazyky fungují stejně jako aplikačně specifické jazyky, ale ještě umožňují přenos programátorových dovedností z jedné aplikace do jiné. Do této skupiny patří již zmíněný *JavaScript*, *Ferite*, *Python*, *Tcl*, *WPS* a další. Z dosud nepředstavených jazyků jsou to tyto:

[2] Jedná se o techniku založenou na synergickém efektu mnoha jednoduchých praktik – na jednom kódu pracují neustále dva programátoři, aby si kontrolovali kód najednou; neustálé testování programu; vždy se programuje jenom to, co je v dané chvíli nezbytné; neustálé testování funkčnosti, zda se během vývoje program nepoškodil. Základem extrémního programování je komunikace, jednoduchost, zpětná vazba a odvaha (smazat chybný kód a začít znovu psát správný kód). [4]

- *GameMonkeyScript* – meziplatformní jazyk navržený pro implementaci do her, podobný jazyku *Lua*, ale syntaxe je bližší jazyku C. Hlavní výhodou je nativní podpora vláken a existující a dodávaný debugger. Stejně jako v jazyce *Lua* je základní datovou strukturou tabulka, ze které se všechny složitější struktury musí vytvořit.
- *ICI* – univerzální interpretovaný programovací jazyk vytvořený v roce 1992 Timem Longem. Je to dynamicky typovaný jazyk, velice podobný *Perlu*, a stejně s ním velmi podporuje regulární výrazy. Primitivní typy zahrnují celá čísla, desetinná čísla, řetězce, soubory, bezpečné ukazatele a regulární výrazy. Ačkoliv se nejedná o objektově orientovaný jazyk, je možno spoustu takovýchto funkcí emulovat. *ICI* je volně k použití i se zdrojovými kódy, takže je asi nejlepší volbou i pro komerční aplikace.
- *RBScript* – skriptovací jazyk podobný REALbasicu (jazyk vyvinutý společností REAL Software Int., dialekt jazyka Basic) podporující moduly a objekty. Využívá se hlavně pro skriptování aplikací napsaných v REALbasicu. *RBScript* nemůže být použit mimo aplikace napsané v REALbasicu, stejně jako nemůže být mimo ně ani nainstalován.
- *Squirrel* – objektově orientovaný skriptovací jazyk navržený pro vývoj rychlých, nenáročných aplikací jako jsou videohry. *Squirrel* je dynamicky typovaný, používá delegáty, ošetřuje výjimky a podporuje více vláken.

h. JAZYK BOO

Jazyk *Boo* byl napsán v roce 2003 v jazyce C#, jeho autorem je Rodrigo Barreto di Oliveira. Jak sám popisuje ve svém Manifestu [8] k tomuto jazyku, napsal ho z důvodu své obliby CLI a „architektonické krásy celého prostředí .NET“. Byl frustrován, že nemohl použít svůj oblíbený jazyk v prostředí, ve kterém pracovala jeho firma. Mluvil o jazyce *Python*, který ovlivnil celý jazyk *Boo*. V *Pythonu* postrádal některé věci, které by byly v silněji staticky typovaném jazyce běžné – kontrola typů již při kompilaci, ale nejvíce postrádal promyšlenost .NET s jeho dobře vytvořenou globalizací a podporou Unicode.

Boo je objektově orientovaný, staticky typovaný (typy se určují již v čase kompilace kódu) jazyk a má *Pythonem* ovlivněnou syntaxi. Na určování typů proměnných volitelně používá výše zmíněnou metodu „duck typing“. Jazyk *Boo* je licencován jako open source.

Po delší době intenzivního programování vznikl jazyk *Boo*. Jazyk neměl zatěžovat programátora nutností psát všechna přetypování a měl být oproštěn od složených závorek, aby byl „přátelský k zápěstí“ (wrist-friendly). Autor chtěl jazyk, který by byl rozšiřitelný jeho vlastními konstrukcemi, a kompiler, který bude generovat CIL. [1]

Bloky kódu a příslušnost k jednotlivým blokům je vyjádřena vizuálně, odsazením. K odsazení se používá TAB, nikoliv jenom mezerník.

Jazyk měl podporovat nedeklarované proměnné, tedy bez jakékoliv deklarace vytvořit proměnnou jejím prvním použitím. Takto mohou ovšem vznikat různé chyby při překlepu atd. Dále by měl sám od sebe poznat, o jaký typ proměnné se jedná, tedy máme-li hypotetickou metodu, která očividně vrací integer, nemusí být proměnná, které je přiřazena hodnota metody, typována jako integer, ale postačí pouhé přiřazení. Pro ilustraci uvedu příklad:

```
def jedna():
     return 1
uno = jedna()
```

Jak je z uvedeného kódu patrné, řádky ani příkazy se neukončují středníkem či podobným znakem, jsou ukončeny pouze entrem. Tento uvedený příklad by mohl být přepsán pro programátora, který je zvyklý na uvádění typů, následovně:

```
def jedna() as int:
     return 1
uno as int = jedna()
```

V *Boo* nemusí být třídy, takže stačí napsat rovnou příkazy do souboru a program je vykoná i bez obligátního „`public static void Main`“. Např. slavný program Ahoj světe vypadá v *Boo* následovně:

```
print(„Ahoj světe!“).
```

Největší důraz je kladen na funkce, které mohou být jako parametry dalších funkcí nebo mohou být vraceny z jiných funkcí, dále mohou být ukládány do proměnných, tj. vytvářet jakési aliasy metod.

```
p = print
p(„Ahoj světe“)
```

Funkce mohou být i objekty, tj. dají se nad nimi volat metody.

```
print.Invoke(„Ahoj světe!")
```

Další důležitá součást jazyka *Boo* jsou generátory, tedy metody, které mohou produkovat více než jednu proměnnou, typickým příkladem je smyčka `for in`. Generátory jsou definovány podle vzoru:

```
<výraz> for <deklarace> in <iterátor> [if|unless
podmínka]
```

Generátory mohou být použity jako návratová hodnota funkce, jako parametr nebo mohou být uloženy do proměnných

```
lichaCisla = i for i in range(10) if i % 2
```

Generátorové metody jsou konstruovány pomocí klíčového slova „yield“**.** Například generátor Fibonacciho posloupnosti vypadá následovně:

```
def fibonacci():
     a, b = 0, 1
     while true
          yield b
          a, b = b, a + b
```

Ačkoliv je tento generátor neomezený, vždy se vygeneruje pouze potřebný počet čísel. Toto nám řekne vestavěný příkaz „zip“. Např. pro vypsání prvních pěti členů stačí napsat

```
for cislo as int, prvek in zip(range(5), fibonacci):
     print(„${cislo+1}: ${prvek}")
```

3. SPECIALITY .NET

.NET je název souboru technologií v softwarových produktech. „Umožňuje nebývalý stupeň integrace software prostřednictvím webových XML služeb: malých samostatných aplikací – stavebních bloků, které jsou pomocí internetu propojeny mezi sebou a také s většími aplikacemi.“ [6]

Pro softwarové vývojáře je .NET Framework velkou změnou – přináší mnoho nových vlastností, které byly původně záležitostí jednotlivých jazyků, do operačního systému. Spojením těchto vlastností s operačním systémem má přinést spoustu výhod, např.:

- zajištění dostupnosti všech vlastností celého rámce (.NET Framework) všem programům napsaným v jakémkoliv jazyce .NET [C# – vlajková loď Microsoftu, připomíná Javu a C++; Visual Basic .NET – kompletně předělaná verze jazyka pro .NET Framework; C++/CLI – „řízená“ verze C++ (Managed C++); J# – přechod mezi Javou a J++ pro prostředí .NET; JScript .NET – kompilovaná verze skriptovacího jazyka *JScript*; *Windows PowerShell* – interaktivní příkazová řádka a skriptovací prostředí, které zajišťuje plný přístup k prostředkům .NET; *IronPython* – implementace jazyka *Python* pro prostředí .NET; F# – funkcionální a objektově orientovaný jazyk pro prostředí .NET]
- zpřístupnění společných způsobů přístupu k rámci nezávisle na použitém programovacím jazyce
- zajištění stejného chování v prostředí rámce nezávisle na použitém programovacím jazyce
- snížení složitosti a zmenšení omezení komunikace mezi programy taktéž pro programy napsané v různých jazycích prostředí .NET
- dovoluje operačnímu systému poskytnout záruky chování programu, které by jinak nemohl nabídnout

Nejprve popíšu Common Language Infrastructure (CLI) – specifikace prostředí pro běh vysokoúrovňových jazyků v prostředí Microsoft .NET. Tato specifikace je vytvořena za účelem propagace a umožnění spolupráce ve více jazycích. Jedná se pouze o specifikaci, na rozdíl od implementace – Common Language Runtime (CLR). Specifikace CLS je otevřená, existují tedy i jiné implementace, než je Microsoft .NET. GNU obdoba .NET se nazývá DotGNU. Její část nazývaná DotGNU Portable.NET umožňuje spouštět všechny .NET aplikace na unixových platformách. V prostředí operačních systémů Linux, UNIX, Mac OS X je k dispozici i sada nástrojů kompatibilní přímo s Microsoft .NET pod názvem Mono (opensource projekt).

CLI specifikuje čtyři hlavní body – Common Type System (definuje základní datové typy a operace), Metadata (informace o struktuře programu zapsány jazykově nezávislým způsobem, ostatní programy si mohou vzít z metadat informace; takto můžeme pracovat i s kódem, který jsme nevytvořili a dokonce nebyl ani vytvořen ve stejném jazyce), Common Language Specification (sada základních pravidel – měl by je splňovat každý jazyk, který má vyhovovat CLI; cílem je dosažení vzájemné spolupráce mezi všemi jazyky implementujícími CLI) a Virtual Execution System (VES) (zavádí a provádí programy splňující specifikaci CLI, používá metadata k tomu, aby zajistil

spolupráci samostatně vytvořených kusů kódu za běhu programu). Více informací na toto téma je možno nalézt v [3] a [5].

Všechny jazyky splňující CLI jsou překládány do Common Intermediate Language (CIL) – společný mezijazyk, který se předává VES a ten zajistí kompilaci CIL do strojového kódu závislého na hardwaru. Z CIL je možno zpětně rekonstruovat zdrojový kód aplikace, která byla napsána v jazyce splňujícím CLI.

Pro vývoj aplikací pod .NET vydal Microsoft Visual Studio .NET – poslední verze 2005 podporuje a využívá .NET v2.0. Oproti verzi 2003 je rozšířeno o návrh webových XML služeb .NET. Prostředí .NET Framework pro PocketPC nebo PDA s operačním systémem Windows Mobile je ve variantě Compact Framework upravená, zmenšená verze .NET; oproti celé verzi neobsahuje velké množství metod a vlastností.

4. PRAKTICKÁ REALIZACE

V úvodu této kapitoly se budu detailněji věnovat popisu některých zajímavých a ve skriptování velice použitelných funkcí a vlastností jazyka *Boo*. První uvedenou vlastností je „duck typing“ – již výše zmiňovaná technika. Pro její zapnutí pro určitou proměnnou jí natypujeme jako „**duck**“. Tento přístup se hodí zejména při prozkoumávání nepopsaného API nebo při práci s nedokumentovanými COM objekty.

```
import System.Threading

def CreateInstance(progid):
     type = System.Type.GetTypeFromProgID(progid)
     return type()

ie as duck =
CreateInstance("InternetExplorer.Application")
ie.Visible = true
ie.Navigate2("http://www.go-mono.com/monologue/")

Thread.Sleep(50ms) while ie.Busy

document = ie.Document
print("${document.title} is ${document.fileSize}
bytes long.")
```

Tato ukázka kódu vytvoří instanci Internet Exploreru, zobrazí jeho okno a přejde na stránku http://www.go-mono.com/monologue/ a počká na kompletní načtení této stránky. Poté vypíše titulek dokumentu a jeho velikost. Výše uvedený příklad tím dokumentuje použití „duck typing“ na neznámém rozhraní Internet Exploreru.

Mnoho jazyků má nepříjemně vyřešené vlastnosti – jako jeden příklad za všechny bych uvedl rozdíl mezi *Boo* a C#. Kód v C# vypadá následovně:

```
class Osoba {

private string _jmeno;
private string _prijmeni;

public string Jmeno {
  get {
     return _jmeno;
  }
}

public string Prijmeni {
  get {
     return _prijmeni;
  }
}

public Osoba(string jmeno, string prijmeni) {
  _jmeno = jmeno;
  _prijmeni = prijmeni;
}
}
```

A stejně funkční kód v *Boo*:

```
class Osoba:

     [getter(Jmeno)]
     _jmeno

     [getter(Prijmeni)]
     _prijmeni

     def constructor([required] jmeno, [required]
prijmeni):
          _jmeno = jmeno
```

```
            _prijmeni = prijmeni
```

Atributy `getter` a `required` jsou pouze syntaktické atributy a stanovují kompileru, že má upravit příslušné části. Hlavní výhodou je, že kód přepisuje kompiler, nikoliv programátor. Po této změně bude kód vypadat následovně:

```
class Osoba:
     _jmeno as string
     _prijmeni as string

     def constructor(jmeno, prijmeni):
          raise ArgumentNullException("jmeno") if
     jmeno is null
          raise ArgumentNullException("prijmeni") if
     prijmeni is null
          _jmeno = jmeno
          _prijmeni = prijmeni

     Jmeno as string:
          get:
               return _jmeno

     Prijmeni as string:
          get:
               return _prijmeni
```

Syntaktická makra jsou podobná jako atributy v tom směru, že stanovují kompileru, že má změnit kód. Syntaktická makra ovšem rozšiřují jazyk. Vezmeme-li v úvahu tento příklad:

```
using reader = File.OpenText(jmeno):
     print(reader.ReadLine())
```

Na rozdíl od vestavěného příkazu „**using“** v C#, zde makro nechá kompiler upravit kód na:

```
try:
     reader = File.OpenFile(jmeno)
     print(reader.ReadLine())
ensure:
     if (__disposable__ = (reader as
System.IDisposable))
          __disposable__.Dispose()
```

```
        __disposable__ = null
    reader = null
```

Každý si může napsat své makro, pokud ho potřebuje. Stačí najít opakující se kus kódu a napsat ho do makra.

Rozšiřitelná syntaxe je pouze jedna část z autorových záměrů, další je rozšiřitelnost kompileru – programátoři by měli mít možnost určit, co se s kódem při kompilaci děje, např. generování dokumentace při kompilaci, vytvoření specifického prostředí pro odhalování chyb nebo úprava a kontrola programátorských konvencí. Tyto požadavky jsou realizovány v tzv. kompilačních pipelines definovaných sadou obecně propojených objektů – kroků – komunikujících spolu dobře navrženou datovou strukturou – kompilační kontext.

Mohou být definovány nové pipelines a kroky a tyto jsou omezeny pouze programátorovým pochopením vnitřní struktury kompileru a struktury jeho vnějších vlastností API. Toto byly pouze některé významné rysy tohoto jazyka.

V další části se zaměřím na použitelnost jazyka v prostředí aplikací napsaných pro architekturu .NET. Naskýtá se otázka, proč jsem si vybral právě jazyk *Boo*?

U zrodu tohoto tématu stála má účast ve zpracovatelském týmu na projektu GREG – Graphics research group – v pozici implementátora skriptování do modulární aplikace. Cílem bylo umožnit nastavení parametrů scény a v ideálním případě celou scénu skriptem. V úvahu připadalo více možností – použít programovací jazyk C# a za běhu jej překládat, použít existující skriptovací jazyk, nebo si vytvořit svůj vlastní skriptovací jazyk. Jakkoliv vypadala poslední varianta nelibě, nevyhnutelně se blížil čas rozhodnutí a vypadala jako jediná schůdná varianta. Na skriptování v jazyce C# se nám nelíbil styl programování, pro skriptování nebyl nejvhodnější, nehledě na problémy se zabezpečením proti škodlivému kódu (které se ukázaly nakonec větší s jazykem *Boo*). V té době jsme nevěděli o žádném vhodném skriptovacím jazyce, který by byl snadno zakomponovatelný do aplikace v prostředí .NET. Tvorba naprosto nového jazyka se nám zdála nevhodná především kvůli velké časové náročnosti a možným bezpečnostním chybám, které bychom dozajista udělali, protože nikdo z nás dosud nepsal svůj jazyk.

Největším problémem na tvorbě jazyka je vymyšlení celé techniky překladu, syntaxe, aby byla uživatelsky použitelná, a algoritmus rozpoznávání jednotlivých jazykových formulací.

Konečná volba padla na jazyk *Boo* a začala fáze implementace tohoto jazyka do naší aplikace. Po počátečním seznámení s jazykem jsem začal vytvářet zabezpečené prostředí pro běh, což se ukázalo jako největší problém. V dalším stádiu jsem dospěl do stavu, kdy jsem byl nucen upravit existující zdrojové kódy jazyka *Boo* a znovu je překompilovat. Poté nastal velký problém – při omezení přístupu na disk nebylo možné načíst ani externí knihovny potřebné k běhu programu. Nakonec se mi povedlo aplikovat omezení na zápis na všechny disky v počítači. Zdrojový kód metody v jazyce C#, která překládá a spouští kód *Boo,* je zde:

```
public CompilerContext RunBooCode()
{
    if (_interpreter != null && _booScript != null)
    {
        CompilerContext cc = null;
        try
        {
           string[] paths = GetDrives();
           FileIOPermission fip = new
FileIOPermission(FileIOPermissionAccess.Write |
FileIOPermissionAccess.Append, paths);
            fip.Deny();
            cc = _interpreter.Eval(_booScript);
            CodeAccessPermission.RevertDeny();
        }
        catch (TargetInvocationException e)
        {
            if (e.InnerException != null &&
e.InnerException is SecurityException)
               Console.WriteLine("Výjimka
zabezpečení, nelze dokončit operaci.");
            else
               Console.WriteLine("Nastala vnitřní
výjimka, nelze dokončit operaci.");
         }
         catch (Exception e)
         {
             Console.WriteLine(e.Message);
         }
```

```
            return cc;
        }
        else
            return null;
}
```

Zde bych rád osvětlil fungování této metody – návratový typ CompilerContext obsahuje veškeré nastavení kompileru, např. seznam používaných sestav (Assembly), konečné chyby při kompilaci, indexer a další věci.

_interpreter, resp. _booScript jsou instanční proměnné referencující objekt interpreteru, resp. zdrojový kód, který má být spuštěn. V bloku try … catch je metoda zjišťující všechny jednotky v počítači a poté blokace všech zápisů či připsání do souborů na těchto jednotkách, jejichž aplikaci provede *fip.Deny()*. Tato metoda je zděděna ze třídy CodeAccessPermission. Práva na cestu udanou parametrem *paths* konstruktoru FileIOPermission jsou aplikována rekurzivně, tedy každý z adresářů v proměnné *paths* je nejnižší adresář, na který se aplikují.

Hlavní výkonná část celé metody je `cc = _interpreter.Eval(_booScript)`, která spustí vlastní kód v interpreteru, pokud neobsahuje chyby. V případě chybného kódu je o tomto uživatel informován výpisem z interpreteru. Po provedení příkazů ze zdroje se vrátí všechna nastavená omezení zpět. Následuje ošetření výjimek a informace uživatele, z jakého důvodu případně skript neproběhl. Rozlišuje mezi třídou TargetInvocationException, která se vyhazuje při chybě při průběhu dynamicky kompilované metody, a ostatními výjimkami. V případě bezpečnostní výjimky je v poli InnerException vnitřní výjimka třídy SecurityException.

Tato metoda není schopna spustit interaktivní interpreter Booish, napsaný v *Boo* samotným autorem, kvůli dosud neodstraněné chybě v jazyce *Boo* – při pokusu o spuštění hlásí „BCE022: Cannot convert ‘Boo.Lang.Interpreter.InteractiveInterpreter’ to ‘Boo.Lang.Interpreter.-InteractiveInterpreter’ “. BCE znamená Boo Compiler Error, tedy chyba kompileru jazyka *Boo*, a chybová hláška by se dala přeložit jako „Nelze převést typ ‘Boo…’ na typ ‘Boo…’ “. Podobná chyba je popsána i na vývojovém portálu pod číslem BOO-795. Tato chyba ovšem je již vyřešena a spočívá v nemožnosti převést jakýkoliv objektový typ na null.

Jediná možnost jak vytvořit instanci interpreteru Booish v prostředí C# je přepsat si zdrojové kódy Booish do C# a poté ji vytvořit a pracovat s ní. Do přílohy č. 1 jsem vložil výpis zdrojového kódu třídy Booish, která reprezentuje jmenovaný interpreter. Konstruktor vytvoří instanci třídy, inicializuje si objekt interpreteru a nastaví zapamatování poslední hodnoty (tj. možnost využít interpreter také jako kalkulačku; na poslední hodnotu se odkazuje podtržítkem).

V příloze č. 2 je výpis zdrojového kódu celé třídy ScriptHost, která realizuje dynamické skriptování v jazyce C# a *Boo*. Třída by se měla používat jako celek, tj. vytvořit instanci, nastavit požadované kódy a ty poté spustit. Raději nepoužívat statické metody, neboť na ně je aplikováno menší zabezpečení z důvodu nemožnosti vytvořit instanci InteractiveInterpreter v prostředí s omezenými právy. Při použití metody CodeAccessPermission.Deny() je možno odepřít přístup pouze k jednomu vzácnému zdroji. Modelový příklad je v těle metody Main. Jediný rozdíl oproti originálu je v rozdílném jazyce komentářů a výpisů – v původním souboru jsou tyto psány anglicky, pro potřeby bakalářské práce jsem je přeložil do češtiny.

Jako demonstraci zabezpečení jsem vyrobil dávkový soubor, který maže všechny textové soubory v dočasném adresáři a nazval jej deltemp.bat:

```
@echo off
cd %tmp%
for %%1 in (*.txt) do del %%1
```

Poté jsem zavolal z prostředí Booish integrovaný příkaz shellp, který spustí proces *cmd.exe* s parametry „/c cesta_k_souboru“, což spustí daný dávkový soubor a poté ukončí interpreter:

```
process = shellp("""c:\windows\system32\cmd.exe""",
"""/c c:\deltemp.bat""")
```

Není vidět žádný textový výstup, ale hlavním zjištěním byla skutečnost, že z dočasného adresáře nezmizel jediný textový soubor. Při testování na příkazové řádce nezůstal v tomto adresáři žádný. Z toho tedy vyplývá zabezpečení proti úpravám souborového systému i spuštěnými procesy v rámci celého skriptovaného kódu.

Třídy, které nevyžadují prostředí s neomezenými právy, se většinou vytváří v aplikační doméně (tj. jakémsi kontejneru), na kterou jsou aplikována

omezení. Toto prostředí se nazývá AppDomain. „Aplikační domény, které jsou reprezentovány objektem AppDomain, pomáhají poskytnout izolaci, ukončení a bezpečnostní omezení pro spouštění řízeného kódu.“[3]

Celé zabezpečení modulu Booish je krkolomné, neboť není možné jednoduše vytvořit AppDomain s nastavenými právy a v ní vytvořit instanci interpreteru, protože tento odmítá instanciaci v jakkoliv omezeném prostředí. Je proto třeba řešit tyto problémy přímým odepřením přístupu ke zdrojům. Tento fakt neřešil možnost spuštěného kódu si v rámci svého prostoru přístup povolit. Po krátké, přesto však efektivní úpravě nemá skript oprávnění si přiřadit jakékoliv prostředky. Tento kód, který původně smazal soubor, vyhodí nyní výjimku:

```
using System;
using System.Collections.Generic;
using System.Security.Permissions;
using System.Security;
using System.IO;

namespace BadCode
{
    class Program
    {
        static void Main(string[] args)
        {
            FileIOPermission fip = new
FileIOPermission(PermissionState.Unrestricted);
            fip.Assert();
            File.Delete("d:\\ssh.old");
            CodeAccessPermission.RevertAssert();
        }
    }
}
```

5. SHRNUTÍ A ZÁVĚR

V této bakalářské práci jsem uvedl úvod do skriptování a zaměřil se hlavně na skriptování v prostředí .NET. Přiblížil jsem filozofii a posléze i použití jazyka Boo. Tento jazyk se jeví velice široce uplatnitelný a po krátké praxi s ním i velice použitelný. Přechod na psaní v jazyce *Boo* mi nečinil velké

[3] http://msdn2.microsoft.com/en-us/library/system.appdomain(vs.80).aspx, získáno dne 23. 4. 2007

problémy, neboť může používat i třídy ze systémových tříd a přístup k programování v tomto jazyce je velice příjemný – autor se snažil šetřit programátorovi psaní, v daný moment, méně podstatných pasáží (např. typy proměnných v cyklu „for“), nicméně mu tuto možnost ponechal. Je pouze na programátorovi, zdali bude explicitně typovat proměnné, či tuto činnost nechá na kompileru (jak již bylo zmíněno, *Boo* je staticky typovaný jazyk).

V části popisující obecně skriptování jsem se pokusil přinést výčet a popis skriptovacích jazyků rozdělených podle hlavního účelu používání. Toto rozdělení je ovšem mnohdy nejednoznačné a proto mohou být různé jazyky zařazovány do jiných skupin. V poslední sekci druhé kapitoly jsem se blíže věnoval jazyku *Boo*, neboť je použit v praktické části této práce.

V projektu GREG se snažíme vytvořit co nejúčinnější zabezpečení kódů psaných uživateli. Učinil jsem i několik pokusů o prolomení zabezpečení, z nichž se mi jeden povedl. Po zjištění možnosti útoku typu „přiřadit si práva metodou CodeAccessPermission.Assert()“, jsem se zabýval touto problematikou a po delší době se mi povedlo i tuto slabinu opravit. Program nemůže být spuštěn se sníženými právy, neboť s nimi není možné vytvořit instanci InteractiveInterpreter, která realizuje vlastní vykonávání *Boo* kódu. Bylo proto nutné vymyslet nějaký jiný způsob, který bude omezovat přístup skriptů k vybraným zdrojům, které chceme chránit.

Ukázka použití třídy ScriptHost je uvedena v příloze č. 3. Program píše stále dokola text písně „5 bottles of beer …“ a průběžně snižuje číslo. Celý skript je načten ze souboru script.boo a poté dynamicky proveden.

6. POUŽITÉ ODKAZY A LITERATURA

[1] *Domovská stránka jazyka Boo*. http://boo.codehaus.org, získáno 10. 4. 2007

[2] *Česká verze encyklopedie Wikipedia*. http://cs.wikipedia.org, získáno 10. – 16. 4. 2007

[3] *Anglická verze encyklopedie Wikipedia*. http://en.wikipedia.org, získáno 10. – 16. 4. 2007

[4] KOLESNIKOV, Pavel. *Extrémní programování v praxi*. http://interval.cz/clanky/extremni-programovani-v-praxi/, získáno 12. 4. 2007

[5] *.NET Framework Developer Center*. http://msdn2.microsoft.com/en-us/netframework/default.aspx, získáno 18. 4. 2007

[6] *Co je platforma .NET?*. http://www.microsoft.com/cze/net/basics/, získáno 18. 4. 2007

[7] *Microsoft Windows XP – Command shell overview.* http://www.microsoft.com/resources/documentation/windows/xp/all/proddocs/en-us/ntcmds_shelloverview.mspx?mfr=true, získáno 17. 4. 2007

[8] OLIVEIRA, R. B. (2004). *Boo manifesto.* Získáno duben 2007, z Boo language: http://boo.codehaus.org/BooManifesto.pdf

PŘÍLOHA Č. 1 – BOOISH

```
using System;
using Boo.Lang.Interpreter;
using System.Security.Permissions;
using System.Security;

namespace Scripting
{
    /// <summary>
    /// Třída reprezentující interaktivní interpreter
    /// jazyka Boo; omezuje jakékoliv operace na
    /// disku pouze na čtení
    /// </summary>
    [Serializable]
    class Booish
    {
        InteractiveInterpreter interpreter;

        // Konstruktor
        public Booish()
        {
            interpreter = new
InteractiveInterpreter();
            interpreter.RememberLastValue = true;
        }

        // Zamezí přístupu k souborům a k registrům,
        // vytvoří interpreter a opět nastaví práva
        // zpět
[RegistryPermission(SecurityAction.Deny,Unrestricted=
true)]
[SecurityPermission(SecurityAction.Deny, Flags =
SecurityPermissionFlag.Assertion |
SecurityPermissionFlag.UnmanagedCode)]
        public void Run()
        {
            FileIOPermission fip = new
FileIOPermission(FileIOPermissionAccess.Write |
FileIOPermissionAccess.Append,
ScriptHost.GetDrives());
            fip.Deny();
            interpreter.ConsoleLoopEval();
            CodeAccessPermission.RevertDeny();
        }    } }
```

PŘÍLOHA Č. 2 – VÝPIS ZDROJOVÉHO KÓDU TŘÍDY SCRIPTHOST

```
using System;
using System.Text;
using System.CodeDom.Compiler;
using System.Reflection;
using System.Security;
using System.Security.Policy;
using System.Security.Permissions;
using Microsoft.CSharp;
using System.IO;
using Boo.Lang.Compiler;
using Boo.Lang.Interpreter;


namespace Scripting
{
    /// <summary>
    /// Zapouzdřuje metody pro získávání informací
    /// o System.Assembly; dále nabízí metody pro
    /// kompilaci C# a Boo za běhu; tato třída nemůže
    /// být zděděna
    /// </summary>
    public sealed class ScriptHost
    {
        /// <summary>
        /// Obsahuje Assembly, které má být spuštěno
        /// </summary>
        private Assembly _scriptAssembly;
        /// <summary>
        /// Pole načtených typů z Assembly
        /// </summary>
        private Type[] _scriptType;
        /// <summary>
        /// Obsahuje cestu ke zkompilovanému souboru
        /// </summary>
        private string _savedFile;
        /// <summary>
        /// Získává cestu ke kompilovanému souboru
        /// (ve složce %TEMP%)
        /// </summary>
        public string SavedFile
        {
```

```
            get
            {
                return _savedFile;
            }
        }
        /// <summary>
        /// Reprezentuje instanci
        /// InteractiveInterpreter, která obsahuje
        /// všechny deklarované proměnné za běhu
        /// skriptu
        /// </summary>
        private InteractiveInterpreter _interpreter;
        /// <summary>
        /// Obsahuje Boo script, který se má spustit
        /// </summary>
        private string _booScript;
        /// <summary>
        /// Získává Boo kód ke spuštění
        /// </summary>
        public string BooScript
        {
            get
            {
                return _booScript;
            }
        }
        /// <summary>
        /// Obsahuje C# kód ke spuštění
        /// </summary>
        private string _cSharpScript;
        /// <summary>
        /// Získává C# kód, který má být spuštěn
        /// </summary>
        public string CSharpCode
        {
            get
            {
                return _cSharpScript;
            }
        }

        private const string appName = "GREG -
scripting module";

        /// <summary>
```

```
        /// Metoda získává všechny jednotky
        /// instalované  v počítači
        /// </summary>
        /// <returns>Pole jednotek (např. C:\)
        /// </returns>
        public static string[] GetDrives()
        {
            string[] arr = new string[24];
            int counter = 0;
            for (char c = 'C'; c <= 'Z'; c++)
            {
                if (Directory.Exists(c + ":\\"))
                    arr[counter++] = (c + ":\\");
            }
            counter = 0;
            while (arr[counter] != null)
                counter++;
            string[] result = new string[counter];
            for (int i = 0; i < counter; i++)
            {
                result[i] = arr[i];
            }
            return result;
        }

        /// <summary>
        /// Metoda vypíše všechna rozhraní, typy
        /// a metody obsažené v souboru
        /// </summary>
        /// <param name="filename">Cesta k souboru
        /// a jeho jméno</param>
        public static void
WriteMethodsFromFile(string filename)
        {
            try
            {
                Assembly asm =
Assembly.LoadFile(filename);
                Type[] typy = asm.GetTypes();
                if (typy == null)
                {
                    Console.WriteLine("Soubor
neobsahuje žádný typ");
                    return;
                }
```

```
                foreach (Type typ in typy)
                {
                    Console.WriteLine("Jméno typu:
{0}", typ.Name);
                    Type[] interfaces =
typ.GetInterfaces();
                    foreach (Type typ2 in typy)

Console.WriteLine("Implementuje rozhraní: {0}",
typ2.ToString());
                    MethodInfo[] mi =
typ.GetMethods();
                    foreach (MethodInfo mi2 in mi)
                    {
                        Console.WriteLine("Obsahuje
metodu: {0} {1}", mi2.IsPrivate ? "private" :
(mi2.IsPublic ? "public" : "???"), mi2.ToString());
                    }
                }

            }
            catch (Exception e)
            {
                Console.WriteLine("Někde byla chyba:
{0}", e.ToString());
            }
        }

        /// <summary>
        /// Načte a provede Boo skript předaný
        /// v parametru. Odepře jakékoliv zápisy na
        /// disk
        /// </summary>
        /// <param name="scriptSource">Zdroj Boo
        /// skriptu</param>
        /// <returns>Zdali uspěl</returns>
        public static bool LoadAndRunBooScript(string
scriptSource)
        {
            InteractiveInterpreter interpreter = new
InteractiveInterpreter();
            interpreter.RememberLastValue = true;
            try
            {
                string[] paths = GetDrives();
```

```
                // Odepřeme práva pro změnu
                // souborového systému
                FileIOPermission fip = new
FileIOPermission(FileIOPermissionAccess.Write |
FileIOPermissionAccess.Append, paths);
                fip.Deny();
                CompilerContext cc =
interpreter.Eval(scriptSource);
                if (cc.Errors.Count != 0)
                    foreach
(Boo.Lang.Compiler.CompilerError err in cc.Errors)
                    {
                        Console.WriteLine("Boo chyba:
"+err.ToString());
                    }
            }
            catch (Exception e)
            {
                Console.WriteLine("Někde byla chyba:
{0}", e.ToString());
                return false;
            }
            finally
            {
                CodeAccessPermission.RevertDeny();
            }
            return true;
        }

        /// <summary>
        /// Načte skript ze souboru a poté ho spustí.
        /// Odepře jakýkoliv zápis do souborového
        /// systému
        /// </summary>
        /// <param name="filename">Soubor se
        /// skriptem</param>
        /// <returns>Zdali uspěl</returns>
        public static bool
LoadAndRunBooScriptFromFile(string filename)
        {
            if (File.Exists(filename))
            {
                string sourceCode = "";
                try
                {
```

```
                    sourceCode =
File.ReadAllText(filename,Encoding.Default);
                }
                catch (Exception e)
                {
                    Console.WriteLine("Chyba při
čtení souboru {0}: {1}", filename, e.ToString());
                    return false;
                }
                return
LoadAndRunBooScript(sourceCode);
            }
            else
                return false;
        }

        /// <summary>
        /// Připojí Boo kód k vnitřnímu zásobníku
        /// </summary>
        /// <param name="code">Boo kód, který má být
        /// připojen</param>
        public void AppendBooScript(string code)
        {
            if (_booScript != null)
                _booScript += code;
            else
                _booScript = code;
        }

        /// <summary>
        /// Nastaví Boo proměnnou „jméno“ s hodnotou
        /// „hodnota“ do seznamu proměnných
        /// </summary>
        /// <param name="name">jméno proměnné</param>
        /// <param name="value">hodnota
        /// proměnné</param>
        /// <returns>Vrací false, pokud není
        /// interpreter inicializován</returns>
        public bool SetBooVariable(string name,
object value)
        {
            if (_interpreter != null)
            {
                _interpreter.SetValue(name, value);
                return true;
```

```
    }
    else
        return false;
}

/// <summary>
/// Vrací hodnotu proměnné „jméno“
/// </summary>
/// <param name="name">Jméno proměnné</param>
/// <returns>Vrací System.object hodnotu
/// proměnné</returns>
public object GetBooVariable(string name)
{
    if (_interpreter != null)
    {
        return _interpreter.GetValue(name);
    }
    else
        return null;
}

/// <summary>
/// Vymaže vnitřní Boo skript
/// </summary>
public void ClearBooCode()
{
    _booScript = null;
}

/// <summary>
/// Nastaví Boo skript k provedení
/// </summary>
/// <param name="code">Zdroj skriptu</param>
public void SetBooCode(string code)
{
    _booScript = code;
}

/// <summary>
/// Spustí vnitřní kód. Kód nemá práva na
/// zápis na disk a žádná práva na registry
/// </summary>
/// <returns>Vrací
/// Boo.Lang.Compiler.CompilerContext;
/// v případě chyby vrací null</returns>
```

```
        [RegistryPermission(SecurityAction.Deny,
Unrestricted=true)]
        [SecurityPermission(SecurityAction.Deny,
Flags = SecurityPermissionFlag.Assertion |
SecurityPermissionFlag.UnmanagedCode)]
        public CompilerContext RunBooCode()
        {
            if (_interpreter != null && _booScript !=
null)
            {
                CompilerContext cc = null;
                try
                {
                    // Odepření zápisu na disk
                    string[] paths = GetDrives();
                    FileIOPermission fip = new
FileIOPermission(FileIOPermissionAccess.Write |
FileIOPermissionAccess.Append, paths);
                    fip.Deny();
                    cc =
_interpreter.Eval(_booScript);
                }
                catch (TargetInvocationException e)
                {
                    if (e.InnerException != null &&
e.InnerException is SecurityException)

Console.WriteLine("Bezpečnostní výjimka, nemohu
dokončit požadovanou operaci.");
                    else
                        Console.WriteLine("Vnitřní
výjimka, nelze dokončit požadovanou operaci.");
                }
                catch (Exception e)
                {
                    Console.WriteLine(e.Message);
                }
                finally
                {
                    try
                    {

CodeAccessPermission.RevertDeny();
                    }
```

```
                    catch (ExecutionEngineException)
{ }
                }
                return cc;
            }
            else
                return null;
        }

        /// <summary>
        /// Vloží C# kód do vnitřního zásobníku
        /// </summary>
        /// <param name="code">C# zdroj
        /// skriptu</param>
        public void SetCSharpCode(string code)
        {
            _cSharpScript = code;
        }

        /// <summary>
        /// Spustí jakýkoliv kód z jakéhokoliv
        /// vnitřního zásobníku, v případě, že jsou
        /// nastaveny oba jazyky, první proběhne Boo
        /// a poté C#.
        /// </summary>
        /// <returns>Vrací true při úspěchu, jinak
        /// false</returns>
        public bool RunAnyCode()
        {
            bool vlajka = true;
            if (_booScript != null)
            {
                CompilerContext cc = RunBooCode();
                if (cc != null && cc.Errors.Count !=
0)
                {

_interpreter.DisplayErrors(cc.Errors);
                    vlajka = false;
                }
            }
            if (_cSharpScript != null)
            {
                vlajka &=
LoadAndRunCSharpScriptFromLine(_cSharpScript);
```

```
            }
            return vlajka;
        }

        /// <summary>
        /// Načte a spustí Boo skript ze souboru.
        /// Skript má omezení na zápis na disk
        /// a nemůže pracovat s registry
        /// </summary>
        /// <param name="filename">Jméno souboru se
        /// skriptem</param>
        /// <returns>Zdali uspěl</returns>
        public bool RunBooCodeFromFile(string
filename)
        {
            bool result = true;
            string code;
            try
            {
                code = File.ReadAllText(filename,
Encoding.Default);
            }
            catch (Exception)
            {
                Console.WriteLine("Nemohu otevřít
soubor {0}.",filename);
                return false;
            }
            SetBooCode(code);
            CompilerContext cc = RunBooCode();
            if (cc == null) {
                result = false;
            } else if (cc.Errors.Count != 0) {
                foreach
(Boo.Lang.Compiler.CompilerError ce in cc.Errors)
                    Console.WriteLine(ce.ToString());
                result = false;
            }
            ClearBooCode();
            return result;
        }

 /// <summary>
 /// Metoda načte C# skript ze parametru a spustí
 /// v něm metodu public void Run().
```

```
 /// Metoda má odepřen veškerý zápis na pevné disky.
 /// </summary>
 /// <param name="scriptSource">Zdroj skriptu
 /// </param>
 /// <returns>Zdali uspěl</returns>
public bool LoadAndRunCSharpScript(string scriptSource)
{
    bool vlajka = true;
    CompilerResults cr = null;
    CSharpCodeProvider provider = new
CSharpCodeProvider();
    System.CodeDom.Compiler.CompilerParameters
parameters = new
System.CodeDom.Compiler.CompilerParameters();
    // Abychom nekompilovali .exe, ale jenom assembly
    parameters.GenerateExecutable = false;
    // Vytvoření dočasného souboru
    try
    {
        _savedFile = Path.GetTempFileName();
        // generovat soubory na disku, nikoliv
        // v paměti
        parameters.GenerateInMemory = false;
        // nastavit jméno výstupního souboru na
        // hodnotu instanční proměnné
        parameters.OutputAssembly = _savedFile;

    }
    catch (IOException)
    {
        Console.WriteLine("Nastala chyba při
vytváření dočasného souboru.");
        return false;
    }
    // zkompilujeme zdrojový kód v závislosti na
    // parametrech kompilace
    try
    {
        cr =
provider.CompileAssemblyFromSource(parameters,
scriptSource);
    }
    catch (NotImplementedException e)
    {
        Console.WriteLine(e.ToString());
```

```
    }
    if (cr.Errors.Count > 0)
    {
        foreach
(System.CodeDom.Compiler.CompilerError ce in
cr.Errors)
            Console.WriteLine(ce.ToString());
        return false;
    }
    else
    {
        try
        {
            Console.WriteLine("Kompilace skončila
úspěšně.");
            // načteme assembly ze souboru, kam jsme
            // jej uložili
            _scriptAssembly =
Assembly.LoadFrom(cr.PathToAssembly);
            // získáme z něj typy
            _scriptType = _scriptAssembly.GetTypes();
        }
        catch (Exception e)
        {
            Console.WriteLine(e.StackTrace);
            return false;
        }
        if (_scriptType == null)
        {
            _scriptAssembly = null;
            Console.WriteLine("Chyba při zjišťování
typu.");
            return false;
        }
        // vypíšeme informace o assembly
        foreach (Type type in _scriptType)
        {
            try
            {
                // získáme informaci o metodě Run()
                // a tu poté zavoláme
                MethodInfo mi =
type.GetMethod("Run");
                if (mi != null)
                {
```

```
                    Console.WriteLine("Spouštím");
                // odepřeme zápis na všechny pevné
                // disky
                    FileIOPermission fip = new
FileIOPermission(FileIOPermissionAccess.Write |
FileIOPermissionAccess.Append,
ScriptHost.GetDrives());
                    fip.Deny();
                    try
                    {
                        // spustíme metodu Run()
mi.Invoke(Activator.CreateInstance(type), null);
                    }
                    catch (SecurityException e)
                    {
                        Console.WriteLine("Výjimka
zabezpečení: {0}", e.Message);
                        vlajka = false;
                    }
                    catch (TargetInvocationException
e)
                    {
                        if (e.InnerException is
SecurityException)
                            Console.WriteLine("Chyba
při spouštění metody, zabezpečení: {0}",
e.InnerException.Message);
                        else
                            Console.WriteLine("Chyba
při spouštění metody, ostatní: {0}",
e.InnerException.Message);
                        vlajka = false;
                    }
                    catch (Exception e)
                    {
                        Console.WriteLine("Obecná
výjimka: {0}", e.ToString());
                        vlajka = false;
                    }
                    finally
                    {
            // nakonec vrátíme zpět všechna omezení

CodeAccessPermission.RevertDeny();
                    }
```

```
                }
                else
                {
                    Console.WriteLine("Typ neobsahuje
metodu public void Run().");
                }
            }
            catch (Exception e)
            {
                Console.WriteLine("Někde byla chyba:
{0}", e.ToString());
                vlajka = false;
            }
        }
    }
    // vrátíme hodnotu true při úspěchu, jinak false
    return vlajka;
}

/// <summary>
/// Metoda načte C# skript ze souboru a spustí metodu
/// v něm obsaženou. Soubor musí obsahovat pouze
/// jedno tělo metody, tj. bez deklarace metody.
/// Importován je pouze namespace System.
/// </summary>
/// <param name="filename">Jméno souboru
/// s kódem</param>
/// <returns>Zdali uspěl</returns>

public bool LoadAndRunCSharpScriptFromFile(string
filename)
        {
            // Tato proměnná bude obsahovat tělo
souboru
            string fileBody = "";
            if (File.Exists(filename))
                fileBody =
File.ReadAllText(filename);
            else
                return false;
            // část kódu napsaná před tělo metody
            string prefix = "using System;\n public
class ScriptClass \n {  public void Run() { ";
            // kód vložený za tělo metody
            string suffix = "} \n }";
```

```
            fileBody = prefix + fileBody + suffix;
            return LoadAndRunCSharpScript(fileBody);
        }
        /// <summary>
        /// Metoda načte C# kód z parametru a provede
        /// ho. Kód musí obsahovat pouze tělo metody
        /// bez deklarace.
        /// </summary>
        /// <param name="code">Zdrojový kód</param>
        /// <returns>Zdali uspěl</returns>
        public bool
LoadAndRunCSharpScriptFromLine(string code)
        {
            // část kódu napsaná před tělo metody
            string prefix = "using System; public
class ScriptClass {  public void Run() { ";
            // kód vložený za tělo metody
            string suffix = " } }";
            code = prefix + code + suffix;
            // vrátíme výsledek akce
            return LoadAndRunCSharpScript(code);
        }

        /// <summary>
        /// Konstruktor. Vytvoří novou instanci třídy
        /// ScriptHost
        /// </summary>
        public ScriptHost()
        {
            _interpreter = new
InteractiveInterpreter();
            _interpreter.RememberLastValue = true;
        }

        /// <summary>
        /// Interaktivní skriptovací konzole jazyka
        /// Boo; pouze pro testování, poněkud pomalé
        /// a neefektivní. Nevrací žádnou textovou
        /// odezvu.
        /// Pro opravdové použití zvažte Booish.Run()
        /// </summary>
        [Obsolete("Pouze pro testování, neefektivní a
nevrací žádnou textovou odezvu. Raději použijte
Booish.Run()", true)]
        public void Run()
```

```
        {
            try
            {
                string command;
                while ((command = Console.ReadLine())
!= "")
                {

ScriptHost.LoadAndRunBooScript(command);
                }
            }
            catch (Exception e)
            {
                Console.WriteLine(e.ToString());
            }
        }

        [STAThread]
        static void Main(string[] args)
        {
            ScriptHost sh = new ScriptHost();
sh.SetBooCode("System.IO.File.Delete(\"\"\"d:\\ssh.ol
d\"\"\")");
            sh.RunAnyCode();
sh.LoadAndRunCSharpScriptFromFile("unsafe.cs");
        }
    }
}
```

PŘÍLOHA Č. 3 – PŘÍKLAD POUŽITÍ TŘÍDY SCRIPTHOST

```
using System;
using System.Collections.Generic;
using System.Text;
using Scripting;
using System.IO;

namespace test_ScriptHost
{
    class Program
    {
        static void Main(string[] args)
        {
            string script =
File.ReadAllText("script.boo");
            ScriptHost sh = new ScriptHost();
            sh.SetBooCode(script);
            sh.RunBooCode();
        }
    }
}
```

Výpis souboru script.boo (převzato z http://boo.codehaus.org/Ninety-Nine+Bottles+of+Beer dne 23. 4. 2007):

```
import System.Threading

class Bottle:
     [Property(Type)]
     static type = "beer"

     static start = 99
     static count = start
     static Count:
          get:
               return count
          set:
               count = value
               start = value

     id as int
     private def constructor(n as int):
          id = n
```

```
     static def take() as Bottle:
          if count > 0:
               print "take one down, pass it around,"
               return Bottle(--count)
          return null

     static def buy() as Bottle:
          count = start + 1
          print "go to the store, buy some more,"
          return Bottle(--count)

     static State as string:
          get:
               return Bottle.ToString()

     static def ToString():
          s = "${count} bottle"
          s += "s" if count != 1
          s += " of ${type}"
          return s


Bottle.Count = 5
while true:
     print "${Bottle.State} on the wall,
${Bottle.State},"
     b = Bottle.take() or Bottle.buy()
     print Bottle.State, "on the wall!"
     print
     Thread.Sleep(1000)
```