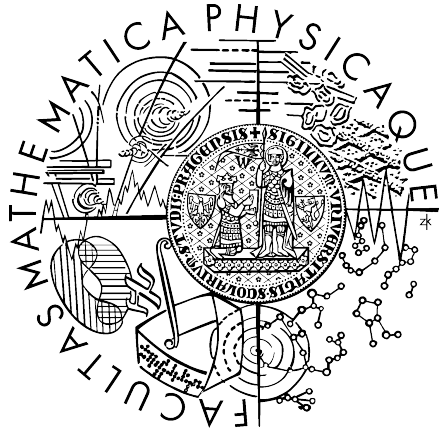
Univerzita Karlova v Prahe

Matematicko-fyzikálna fakulta




# BAKALÁRSKA PRÁCA

Martin Molnár

## Objektovo orientovaný príkazový interpret

Katedra softwarového inžinierstva

Vedúci bakalárskej práce: Mgr. Pavel Ježek

Študijný program Informatika, obor Obecná informatika

2007

Na tomto mieste by som rád vyslovil vďaku svojmu vedúcemu ročníkového projektu a tejto bakalárskej práce, pánovi Pavlovi Ježkovi, za jeho ochotu, vedenie v projekte a tiež za mnohé podnetné návrhy k obsahu i forme tejto práce.

Vyhlasujem, že som svoju bakalársku prácu napísal samostatne a výhradne s použitím citovaných prameňov. Súhlasím so zapožičiavaním práce a jej zverejňovaním

V Prahe dňa 9. 8. 2007 Martin Molnár

# Obsah

Názov práce: Objektovo orientovaný príkazový interpret
Autor: Martin Molnár

Katedra: Katedra softwarového inžinierstva

Vedúci bakalárskej práce: Mgr. Pavel Ježek

e-mail vedúceho: pavel.jezek@mff.cuni.cz

Abstrakt: Predložená práca sa zaoberá vnášaním objektovo orientovaného prístupu do programovania v príkazovom riadku. Študuje možnosti zľadenia konceptov OOP a dávkového spracovania a navrhuje vhodný kompromis medzi týmito prístupmi. Hlavným cieľom tejto práce je implementácia zvoleného riešenia v rodinách operačných systémov Unix a Microsoft Windows. Práca teda popisuje rozhodnutia súvisiace s implementáciou, ako aj používanie výslednej aplikácie.

Kľúčové slová: príkazový interpret, objektovo orientované programovanie, filozofia programovania

Title: Object-oriented Command Interpreter

Author: Martin Molnár

Department: Department of Software Engineering

Supervisor: Mgr. Pavel Ježek

Supervisor's e-mail: pavel.jezek@mff.cuni.cz

Abstract: The goal of the proposed work is to explore possibilities of enriching command line interpreter programming by an object oriented approach. Author studies options of harmonizing concepts of the OOP and the batch processing and designs appropriate compromise between these two concepts. The main goal of the work is to create an implementation of the chosen design in the Unix and Microsoft Windows operating system families. This work describes decisions during the implementation and also include user manual of the resulting application.

Keywords: command interpreter, object-oriented programming, programming philosophy

# Kapitola 1
# Úvod

V dobe vzniku operačných systémov boli vytvorené príkazové interprety, ktorých základnou funkciou bolo spúšťanie úloh. Medzi časom, vznikom grafického rozhrania, sa vyvynuli iné spôsoby spúšťania programov, preto bola táto funkcia u príkazových interpretov oslabená. Tým sa však dostalo do popredia ich, oproti tlačítku štart a rôznym formulárom, silná stránka, a tou je možnosť programovaním skriptov automatizovať často alebo na viacerých počítačoch vykonávané postupy.

Ďalšou významnou zmenou v oblasti IT bol vznik objektovo orientovaného programovania, ktoré prináša programátorovi určitý komfort hlave v podobe intuitívnejšieho programovania.

Cieľom tejto práce a ročníkového projektu, na ktorý táto práca nadväzuje je sprístupniť objektovo orientované programovanie pre účely, na ktoré sa používa príkazový interpret. Výsledný shell má byť určený pre rodiny operačných systémov Unix® a Microsoft Windows®. Má byť dobre použiteľný a okrem iného poskytovať užívateľovi automatické doplňovanie identifikátorov a mien súborov. Tiež by mal podporovať prácu s COM objektami, pretože sa tým výrazne rozširujú možnosti jeho použitia.

# Kapitola 2
# Prehľad problematiky

Medzi spomínanými operačnými systémami existujú výrazné rozdiely. Ideálne by bolo, keby implementovaný shell (v ďalšom texte ho budeme označovať pracovným názvom `losh`) umožňoval písať prenositeľný kód. Preklenúť tieto rozdiely do takej miery je však nepredstaviteľné. Shell by sa však mal pokúsiť čo najviac ich zakryť.

## *2.1 Operačné systémy*

### Rozdiely vo vonkajšej podobe operačných systémov

Rozdiely, ktoré sa priamo dotýkajú rozhrania, cez ktoré užívateľ vníma systém, je ťažké zakryť. Čo sa týka Unixu, táto rodina je značne rozvetvená a jednotlivé operačné systémy dosť rozdielne, všetky princípy súvisiace so shellom však majú rovnaké. Jednotlivé verzie operačného systému Windows sú v týchto princípoch tiež jednotné, zdedili ich od svojho predchodcu, operačného systému Microsoft DOS. Tieto rozdiely sú však značné medzi týmito rodinami. Na prvý pohľad je to najmä používanie iných oddelovačov adresárov v ceste a viac adresárových stromov v prípade Windows na rozdiel od jedného stromu v Unixe[1], to sú však len kozmetické rozdiely, odstránenie ktorých nie je veľký problém. V projekte implementovaný shell napríklad spája všetky stromy diskových oddielov vo Windows do jedného. Vážny problém však tvoria koncepty, ktoré v druhej rodine nemajú svoj proťajšok. Takými sú napríklad systém prístupových práv (MS DOS a Windows 95, 98 a Me nemajú) a atribúty súborov (pôvodný Unix nemá). Obe rodiny však v poslednej dobe implementujú navzájom si podobný systém práv prostredníctvom Access Control List-ov.

1 Za zmienku tiež stojí fakt, že v Unixovom adresárovom strome sa nachádzajú aj ďalšie objekty, ako napríklad zariadenia, tiež v ňom zvyknú byť pripojené virtuálne súborové systémy, napr. procfs, debugfs. Pomenované pajpy a sockety existujú v Unixe aj vo Windows.

### Rozdiely v rozhraní OS používanom shellom

Naopak rozdiely v princípoch používaných pri interakcii OS a shellu je možné úplne skryť. Ovplyvňujú len spôsob implementácie funkcí shellu.

Jeden z rozdielov je v predávaní parametrov vytváranému procesu. V Unixe je shell zodpovedný za rozdelenie príkazu na jednotlivé parametre a interpretáciu masiek ciest súborov v nich, to znamená, že vznikajúci proces dostane ako argumenty cesty k súborom. Na rozdiel od toho vo Windows príkazový interpret odovzdá celý príkaz vznikajúcemu procesu, ktorý ho rozdelí na jednotlivé argumenty[2]. Taktiež je zodpovednosťou procesu, prostredníctvom k tomu určených systémových volaní, interpretovať masky ciest súborov. Dôsledkom toho je tiež fakt, že názvy súborov nemôžu obsahovať niektoré znaky[3].

Ďalším rozdielom je tiež spôsob vytvárania procesov. Vo Windows sa procesy vytvárajú volaním `CreateProcess` a všetky požadované vlastnosti a nastavenia sa odovzdávajú systému v argumentoch tohto volania. V Unixe najprv rodičovský proces zavolá `fork()`, operačný systém proces naklonuje, v rodičovskom procese vráti `fork` identifikačné číslo (PID) synovského procesu a v synovskom procese vráti hodnotu `0`. Synovský proces potom môže aplikovať požadované nastavenia štandardným spôsobom. Príkladom je zavretie deskriptorov súborov, ktoré by musel v prístupe z Windows rodičovský proces označovať za nededitelné špeciálnymi volaniami. Synovský proces v Unixe potom zavolá `exec(...)` ktorý načíta do pamäti nový obraz spustiteľného súboru.

Rozdiel je tiež v práci s konzolou, ostatné rozdiely však nie sú konceptuálne.

2 keďže jazyk C bol vyvynutý pre Unix, má funkcia main argumenty prispôsobené Unixovému spôsobu ich odovzdávania. Preto vo Windows, kde je za rozdelenie na jednotlivé argumenty zodpovedný vznikajúci proces, toto rozdelenie vykováva C runtime pred zavolaním funkcie main.

3 v Unixe sú zakázané len 2 znaky v názve súboru, a to lomítko a nulový znak označujúci koniec reťazca v C.

## *2.2 Príkazové interprety*

V Unixe existuje niekoľko príkazových interpretov, napr. `bash`, `csh`, líšiace sa prevažne len syntaxou. Okrem toho sa na podobné účely sa niekedy používajú aj iné interpretované jazyky, ako napríklad `python` alebo `perl`. Vo Windows je klasickým a donedávna jediným príkazovým interpretom `cmd.exe`, k najnovších verziám Windows je však možné používať `Windows PowerShell`. Cieľom projektu je, aby implementovaný shell nahrádzal klasické príkazové interprety týchto systémov. Zamerajme sa preto na ich vlastnosti.

### Funkcia príkazových interpretov

Základnou funkciou shellu je spúšťať procesy, so všetkými možnosťami, ktoré k tomu operačný systém ponúka. Samozrejmosťou sú argumenty pre spúšťaný proces, ale tiež aj nastavovanie environmentálnych premenných. V koncepcii týchto OS, v ktorej procesy spracúvajú svoj štandardný vstup a produkujú výstup, sa procesom na vstup a výstup prikladajú nielen súbory, ale aj výstup, resp. vstup iného procesu, čo dáva príkazovým interpretom možnosť riešiť komplexnejšie problémy. V Unixe majú tiež shelli možnosť pozastavovať spustené úlohy alebo ich spúšťať na pozadí. Túto vlastnosť budeme nazývať „job control“.

Okrem toho shelli majú skriptovací aspekt, čo sa prejavuje cyklami a inými riadiacimi štruktúrami v ich jazykoch. Ďalšie vlastnosti shellu sú len priblížením užívateľovi, medzi ne patrí napr. automatické doplňovanie alebo aliasy v `bash`i.

### Existujúce príkazové interprety

Tomuto popisu odpovedajú klasické nástroje, ktorými sú `bash` a `cmd.exe`. Prvý z nich je o niečo komplexnejší, čo je dôsledkom doby, v ktorej vznikal, užívateľmi a orientáciou daných operačných systémov.

Na druhú stranu `Windows Powershell` nezodpovedá tejto klasickej definícii, pretože princíp programovania v ňom nie je postavený na

spúšťaní procesov. Na rozdiel od toho používa pri dávkovom spracovaní hlavne tzv. commandlety, čo sú moduly implementované nie ako samostatné procesy, ale ako objekty v knižniciach systému. Existencia tohto protipríkladu nás nabáda hľadať obecnejšiu podstatu príkazových interpretov. V rozšírenej podobe by sme mohli shell definovať ako nástroj administrácie systému. Zatiaľčo v Unixe je administrácia postavená na programoch a editácii konfiguračných súborov, Windows poskytuje objektový prístup, napríklad vo forme WMI alebo .NET, čo je pre narastajúcu komplexitu operačných systémov veľká výhoda. Takáto koncepcia neposkytuje `cmd.exe` také pole realizácie, ako má `bash` v Unixe, čo je aj dôvod jeho slabého postavenia, naopak `Windows PowerShell` je práve preto veľmi silným nástrojom.

# Kapitola 3

# Základné princípy objektovo orientovaného shellu

Cieľom projektu a tejto práce je vniesť objektovo orientované programovanie do shellu. Kľúčovou úlohou je zladiť ich princípy a vytvoriť vhodnú koncepciu. Nesprávne rozhodnutia v tejto fáze totiž môžu spôsobiť nevhodnosť výslednej aplikácie, alebo problémy s implementáciou niektorých jej vlastností.

## *3.1 Jednotlivé koncepcie*

V úvode sa najprv zamyslíme nad samotnými koncepciami

### Pajpa[4]

Jednou zo základných vlastností procesov v ich pôvodnej koncepcii bolo, že spracovávali súbor dát zvaný štandardný vstup a tak produkovali štandardný výstup. Shell má pritom možnosť nedať vytváranému procesu za vstup terminál, alebo súbor na disku, ale výstup iného procesu, čím je možné riešiť omnoho komplexnejšie problémy. Tento princíp je v protiklade s klasickým procedurálnym programovaním, ktoré používa prvky ako premenné alebo cykly. Hoci sa aj tieto prvky v shelloch vyskytujú, typickou vlastnosťou príkazových interpretov je dôraz kladený na dávkové spracovávanie.

### Objekty

Hlavnou výhodou objektovo orientovaného programovania je intuitívnejší pohľad zo strany programátora, ale tiež väčšia možnosť implementovať nápovedu vo forme automatického doplňovania

4 toto slovo je v podstate programátorský slang pochádzajúci z angličtiny. Nanešťastie ale v slovenčine nie je ustálený iný názov. Jeho možným ekvivalentom je rúra, ktoré pre zmenu ale pochádza z nemčiny. Najslovanskejšie alternatívy popisujúce objekty v angličtine popisované slovom pipe sú potrubie, prípadne rôzne druhy produktovodov (ropovod, vodovod, ...). S trochou nadsádzky by sa teda „pipe“ mohlo prekladať ako „dátovod“, prípadne v tejto práci „objektovod“.

identifikátorov. V prístupe klasického shellu totiž existuje veľké množstvo príkazov, a vybratie konkrétneho príkazu väčšinou nezužuje okruh prípadných argumentov[5]. Naopak pri objektovo orientovanom prístupe, keď sa v príkazovom riadku píše najprv objekt a až potom príkaz, ktorý sa má na objekte vykonať, je možné informáciu o objekte využiť na redukciu navrhovaných príkazov v ideálnom prípade do takej miery ako to robia grafické súborové manažéry, napr. `Windows Explorer`, keď podľa typu súboru zobrazujú editory, v ktorých sa daný súbor dá otvoriť. Na druhú stranu, ako sme uviedli v úvode, shelly už nemajú funkciu súborových manažérov a ostala im hlavne skriptovacia funkcia, preto by to nebolo účelné.

### Typovanosť v OOP

Existuje škála prístupov k tomu, aký význam sa v OOP kladie typom objektov. V kompilovaných jazykoch, napr. v C++, je nevyhnutné, aby mal každý objekt presne definovaný typ. Hlavnou výhodou tohto prístupu je rýchlosť prístupu k položkám, ale tiež odhalenie niektorých druhov chýb už pri kompilácii. Nevýhodou je nepružnosť v podobe nutnosti písania deklarácii typov, premenných a argumentov metód. Príkladom zmäkčenia tejto nevýhody je typ `Variant` vo `Visual Basicu`. Na druhú stranu interpretované jazyky zväčša typom neprikladajú veľký význam. Užívateľ v nich deklaruje triedy a ich metódy, do premenných však môže priraďovať bez obmedzení. Pri extrémnejšom prístupe, ako príklad slúži jazyk `ruby`, typy a triedy vôbec nehrajú rolu a každý objekt je potenciálne jedinečný. Vzhľadom k tomu, že shell je interaktívny a očakáva sa od neho úspora stlačených kláves, deklarovanie typov sa preň nehodí.

## *3.2 Zladenie OOP a dávkového prístupu*

Vyššie spomínané princípy však nie sú úplne zlúčiteľné, dokonca sú si istým spôsobom protichodné. Základom OOP je totiž individuálny

5 `bash` má ale veľmi dobre prepracovaný systém automatického doplňovania, ktorý (pri dôslednej konfigurácii) napríklad pri stlačení tabulátoru za príkazom `mount` nenavrhuje súbory ale položky z `/etc/fstab`.

prístup k objektom, napríklad vo forme virtuálnych metód. Objekty môžu na volanie tej istej funkcie reagovať odlišne, zatiaľčo v dávkovom prístupe sa najprv vybere modul, do ktorého sa potom spracovávané dáta posielajú. Kompromis je ťažké nájsť. Jedna možnosť je pre každý objekt zvlášť určovať, ktorá metóda sa zavolá, čo znamená potlačenie dávkového princípu, túto sémantiku má v `loshi` operátor „`:`“. Druhá možnosť je zaviesť prísne typovanie a nepodporovať polymorfizmus, nevýhodou tohto prístupu sú nutné deklarácie objektov. Tretím prístupom je určiť polymorfne modul podľa prvého objektu prechádzajúceho pajpou. Tento prístup v `loshi` používa operátor „`|`“. Nevýhodou je, že pajpou môžu tiecť rôzne objekty, preto to modul musí ošetrovať.

## *3.3 Zvolený prístup*

Pri návrhu objektovo-orientovaného jazyka pre shell som zvolil nasledovný prístup: Objekty sú usporiadané do stromu, ktorého koreň sa označuje `/`. Pritom objekty nemajú typy, t.j. neexistujú triedy, ktoré by určovali, aké položky objekt má. Položky objektu sú vytvárané za behu a to priradením `:=`. Napríklad príkaz `/:a:b:=@text` vytvorí objektu `/` položku `a`, tej vytvorí položku `b` a do nej priradí reťazec `text`.

Každopádne pojem triedy a dedičnosti je paradigmou, bez ktorej by sa len ťažko dala sprístupniť intuitívnosť OOP v shelli. Preto sú obsiahnuté aj v jazyku `loshu`, aj keď trochu netradične, a to zavedením dedičnosti medzi objektami. Funkciu triedy má objekt, ktorý má svoje položky a mechanizmus dedičnosti funguje tak, že inštancia dedí tieto položky, okrem tých, ktoré má prekryté[6]. Takýmito položkami môžu byť metódy, čo sú tiež objekty, trochu špeciálne v tom zmysle, že akceptujú použitie operátoru volania `()`. Niekoľko metód je vstavaných priamo v shelli, okrem toho je ako metóda použiteľný špeciálny typ objektu – blok, t.j. kód uzavrený vo svorkových zátvorkách `{}`.

6 tj. okrem prípadov, keď má inštancia položku s rovnakým menom ako trieda, v takom prípade je prístupná položka inštancie

### 3.3.a Sémantika operátorov

V tomto odstavci definujeme hodnotu výrazov. Predpokladáme, že operandy majú nejakú hodnotu a na základe nej definujeme hodnotu výrazu, ktorý vznikne aplikovaním operátoru na tieto operandy. Hlavným princípom `loshu` je, že hodnotou výrazu nie je objekt, ale zoznam objektov. Pri vyhodnocovaní v skutočnosti nepoznáme hodnotu operandov, ale len jeden objekt zo zoznamu. Za ním nasledujúce objekty ešte nie sú vyhodnotené a predchádzajúce sú už zabudnuté. Pri popise jednotlivých operátorov si všimnime, že takýto prístup je možný, pretože ich sémantika je istým spôsobom lokálna.

Operátor `o:i` nahradzuje každý objekt zoznamu, na ktorý sa vyhodnotí ľavý operand `o`, jeho položkou, pričom používa pravý operand `i` ako identifikátor.

Podobne operátor `c(p)` postupne vyberie každý objekt zo zoznamu, ktorý je hodnotou ľavého operandu `c`, a aplikuje naň zavolanie, pričom ako volajúci objekt je uvedený objekt, ktorého položkou bola volaná metóda[7]. Operand `p` sa odovzdá zavolanej metóde ako parameter. Z výsledkov volaní týchto metód sa zreťazí zoznam, ktorý sa považuje za vyhodnotenie výrazu s operátorom volania.

Použitie operátoru `o/s` je skratka za volanie špeciálne pomenovanej metódy `_item_`[8]. Objekt adresár takto sprístupňuje svoje položky, iné objekty môžu obecne robiť niečo iné.

Ternárny operátor `o|i(p)` vybere položku prvého prvku operandu `o`, a ten považuje za metódu, ktorá bude spracovávať celý zoznam – hodnotu tohto operandu `o`. Ako parametre odovzdá tejto metóde hodnotu výrazu v zátvorkách.

Ternárny operátor `o:i:=v` má tri operandy, a to objekt `o`, identifikátor `i` a pravú stranu `v`. Aplikácia tohto operátoru spôsobí dosadenie prvého prvku hodnoty operandu `v` do objektu pod identifikátor `i`. Dôvod, prečo sa zavádza tento ternárny operátor, a nie

7 zoznam, ktorý dostane metóda ako volajúceho, je jednoprvkový
8 tj. `o/s` je ekvivalentné `o:_item_(s)`

binárny operátor `:=` je ten, že by sa tým v gramatike stali korektné výrazy, ktoré nemajú dobrý zmysel. Zápisom ...`a():=b` by sa napríklad umožňovalo dostadenie do výsledku funkcie.

Ďalšími operátormi sú zreťazenie zoznamov „`,`“, blok `{}` a v ňom používaný oddelovač príkazov „`;`“ a operátor `^`, ktorý v bloku referuje na objekt, na ktorom bol blok zavolaný (ekvivalent kľúčového slova `this` v C++).

Pretože operátory takto spracovávajú zoznamy objektov, implementovaný príkazový interpret sa nazýva zoznamovo orientovaný shell a od toho je odvodená anglická skratka **`losh`**.

### 3.3.b Rozdiel medzi operátormi `:` a `|`

Štandardný spôsob spracovania zoznamu metódou je pomocou operátoru `|`. Ako sme už uviedli, funguje tak, že losh vyberie prvý objekt vstupného zoznamu a jeho položku danú identifikátorom určí za metódu, ktorej dá na spracovanie celý zoznam.

Operátory `:` a `()` sú v `losh`i hlavne z iných dôvodov, každopádne ich ale užívateľ môže zkombinovať. Výsledok je pritom odlišný od použitia operátora `|`. Podľa sémantiky `:` sa vstupný zoznam nahradzuje zoznamom, v ktorom je každý objekt položkou príslušného objektu vstupného zoznamu. Ak teda máme zoznam objektov a identifikátor na pravej strane `:` u každého z nich určuje metódu, výsledok operácie je zoznam metód. Použitie `()` na tento zoznam každú metódu zavolá a ich výsledky zreťazí. Každá metóda má ako vstup jeden objekt a to ten, ktorého položkou bola.

V prípade, že podstatou metódy je každý zo vstupných objektov nejak ztransformovať a poslať ďalej, oba operátory vedú k rovnakému výsledku. Rozdiel však nastáva u metód, ktoré nemajú takýto lokálny charakter. Napríklad metóda `count`, ktorá vracia počet vstupných objektov pri použití s operátorom `|` funguje intuitívne. Ak však užívateľ použije operátor `:`, losh najprv vygeneruje zoznam tákýchto metód

(každému objektu zoznamu sa vyberie položka count), a potom každú zavolá na jednoprvkovom zozname. Preto je výsledkom zoznam jednotiek.

### 3.3.c Interpretácia výsledného zoznamu

Väčšina typov objektov vo výslednom zozname ignoruje. Predpokladá sa totiž, že želaný zmysel mali vedľajšie efekty ich spracovania metódami. Napríklad metóda echo vypíše reťazec ale tiež ho vráti, čo umožňuje reťazec spracovávať ďalšou metódou. Ak sa vo vyhodnotenom zozname nachádzajú reťazce, predpokladá sa, že ak ich užívateľ chcel vypísať, použil túto metódu.

Jedným zo špeciálnych typov objektov je výnimka. Tú vloží do svojho výstupného zoznamu niektorá metóda v neštandardnej situácii a jej spracovanie ostatnými metódami ju prenáša až do konečného zoznamu. Tam sa výnimky, na rozdiel od ostatných typov objektov neignorujú, ale užívateľské rozhranie ich vypíše užívateľovi.

Druhým typom objektov, ktoré sa vo výslednom zozname hľadajú, sú príkazy na spustenie procesov. Takýto objekt vzniká, okrem iného, v metóde exec a môže byť ďalšími metódami upravovaný. Počas vyhodnocovania výrazu tieto metódy budujú komplexný príkaz, ktorý obsahuje procesy pospájané pajpami, s presmerovanými vstupmi a výstupmi, s nastavenými argumentmi a premennými environmentu. Po vyhodnotení príkazu teda nastáva ďalšia fáza práce shellu, v ktorej spúšťa tieto procesy.

### 3.3.d COM objekty

Ku COM objektom musí byť v porovnaní s objektami `loshu` zachovaný iný prístup. Hlavný rozdiel je v tom, že objekty shellu vytvárame, pričom COM objekty sú dané ich programátorom, môžeme ich využívať, ale nie meniť. Napríklad COM objekty sú typované, nie je možné im pridávať položky, tiež sa na ne nemôže vzťahovať dedičnosť podľa koncepcie v `loshi`. Podobne pri ich návrhu a programovaní nebol podporovaný

dávkový prístup. Úlohou shellu v tejto oblasti je teda vytvárať COM objekty, volať ich metódy a zobrazovať ich v rámci automatického doplňovania.

### 3.3.e Syntax

V prvom rade by sme mali určiť znaky so špeciálnym významom. Keďže sa v cestách súborov budú musieť escapeovať, mali by byť, pokiaľ možno, menej používané. Z tohto dôvodu sa operátor „`:`“ nenazýva „`.`“. Týmito znakmi sú hlavne znaky tvoriace jednotlivé operátory, konkrétne sú to `:|/,;(){}^@#\"` a medzera. Ak sa takýto znak vyskytuje v identifikátore alebo názve súboru, musí byť pred neho pridaný znak „`\`“, alebo musí byť identifikátor uvedený v uvodzovkách. Okolo operátorov sa na ich zvýraznenie môžu vkladať medzery.

Aby sa v `loshi` oddelili identifikátory od číselných a hlavne reťazcových konštánt, token začínajúci `#` sa interpretuje ako číslo, podobne reťazec sa zapisuje so znakom `@` pred ním, pričom do uvodzoviek sa dáva len v prípade, že obsahuje špeciálne znaky (vrátane medzery). Cesty v adresárovom strome sa píšu ako postupná aplikácia operátora `/` na objekt `/`. Na Unixe objekt `/` reprezentuje koreň adresárového stromu a na Windows spája stromy tak, že napríklad `//c` reprezentuje adresár `c:\`. Napríklad `//home` reprezentuje na Unixe adresár `/home`, `//c/“Program Files“` na Windows `c:\Program Files`. V Unixe sa teda pred cestu vkladá `/`, čo korešponduje s vkladaním `@` a `#` pred reťazcové, resp. číselné konštanty. Objekt `/:cd` odkazuje na aktuálny adresár, preto `/:cd/...` realizuje zadávanie relatívnej cesty.

Neescapeované špeciálne znaky tvoria operátory, pričom ich priorita je nasledovná: Najvyššiu prioritu majú operátory `:`, `|` a `/`. Nižšiu prioritu má operátor volania `()`, nasledovaný operátorom „`,`“, ďalej „`;`“ a najnižšiu prioritu majú `{}`. Na explicitné určenie priority operátorov sa používajú zátvorky.

### 3.3.f Nevýhoda takto definovaného jazyka

Takáto syntax je typická pre objektovo orientovaný návrh. Jej hlavným problémom je, že nie je šetrná. Príkaz, ktorý sa v `bash`i znie

```
cat /etc/passwd | wc -l
```

sa bude musieť zapisovať

```
/:exec(@cat, //etc/passwd):exec(@wc, @-l)
```

Obecne sa pritom od shellu očakáva úspora stlačených kláves. Je to istým spôsobom daň za vnášanie objektovo orientovaného programovania do shellu. Na druhú stranu, ako som napísal v úvode, v shelli je už v popredí jeho skriptovacia funkcia, čím sa táto nevýhoda z časti potláča. Výhodou objektového prístupu je, že užívateľovi umožňuje vytvárať objektom metódy a kombinovať ich. Napríklad môže postupom uvedeným v kapitole 5.3 pridať súborom metódu `cat`, tak že zápis `//etc/passwd:cat()` zavolá `/:exec(@cat, //etc/passwd)`.

Navyše, ak by sa mal v objektovo orientovanom shelli zaviesť tak krátky zápis, narušilo by to koncepciu daného jazyka možno do takej miery, že by bolo rozumnejšie zostať u princípov klasických shellov.

## *3.4 Windows PowerShell*

Iný prístup je zvolený vo Windows PowerShelli. Jeho tvorcovia nepoužili, ako nový princíp programovania zanesený do shellu, objektovo orientované programovanie. Jeden z troch základných princípov OOP - zapúzdrenie - definuje objekt ako dáta spoločne s na nich definovanými metódami. Objekty v PowerShelli sú iba dáta, metódy v PowerShelli neexistujú, ich funkciu spĺňajú moduly, tzv. commandlety. Tento rozdiel vôbec nie je formálny, v koncepte Windows PowerShellu totiž nemôže existovať ani ďalší základný princíp OOP, polymorfizmus a jeho nástroj - virtuálne metódy. Namiesto toho sa PowerShell definovaním modulov - commandletov drží tradície príkazových interpretov - dávkového spracovávania. Nie je preto prekvapujúce, že v otázke zápisu príkazu na spustenie procesov, rozoberanom v kapitole

3.3.f, zachováva výhody krátkosti zápisu v `bash`i a `cmd`.

# Kapitola 4
# Spôsob implementácie

Na implementáciu týchto koncepcii som zvolil jazyk C++, ktorý je podľa môjho názoru dobrým kompromisom medzi C a vyššími jazykmi (Java) pre tieto účely. Veľká časť problémov počas implementácie bola technického charakteru, alebo s priamočiarym riešením. Tie, ktorých riešenie vyžadovalo závažné rozhodnutia, prípadne bolo iným spôsobom netriviálne, popíšeme.

## *4.1 Counted pointers*

Vzhľadom k tomu, že objekty shellu sú medzi sebou poprepájané rôznymi vzťahmi (položka, inštancia „triedy“ = objektu, …), neexistuje rozumný spôsob určenia „vlastníctva“ objektu, preto zodpovednosť za deštrukciu objektu (a odalokovanie príslušnej pamäti) musí byť zdieľaná medzi viaceré objekty, resp. ten z nich, ktorý ostane ako posledný. Existovali dve možnosti, ako sa s týmto problémom vysporiadať:

- klasické counted pointers na úrovni objektov C++
- počítanie referencí medzi objektami shellu

V skutočnosti objekty shellu poznajú referencie (trieda `Mediator`) z dôvodov popísaných v ďalších kapitolách, takže implementácia touto cestou bola možná, aj keď sa tieto referencie používajú iba pri vzťahu objekt – položka. Táto implementácia by bola rýchlejšia, ale značne neprehľadná. Na druhú stranu klasické counted pointers sa použili aj na objekty C++, ktoré nereprezentujú objekty shellu. Tiež boli použité aj na technickú realizáciu atómov (popísanej v ďalšej kapitole), aj keď bolo potrebné ich mierne obohatenie. Preto nebolo vhodné použiť niektorú z knihovných implementácii counted pointrov.

### Problém cyklov

Problémom riešenia pomocou counted pointrov je možný vznik na

seba do kruhu odkazujúcich objektov. Takýto kruh sa neuvoľní ani keď prestane byť dosažiteľný. V projekte sa preto vyhýba takýmto situáciam, a ak sú nevyhnutné, používa sa trik s implementáciou objektu dvoma objektami[9], ktoré majú medzi sebou klasický C pointer prerušujúci takýto kruh.

Problém, ktorý v projekte nie je ošetrený, môže nastať, ak užívateľ priradí objektu ako položku samého seba, alebo objektu dá ako položku jeho vlastnú inštanciu, prípadne môže vytvoriť aj cykly väčšej dĺžky[10]. Tento problém sa dá riešiť buď zavedení zložitejších mechanizmov (garbage collection), alebo jednoducho zakázaním vytvárania takýchto neštandardných väzieb, avšak overenie, či takáto zakázaná situácia nastáva je príliš náročné na čas, ak by sa malo vykonať pri každom priradení.

### Triedy

Counted pointers sú implementované spôsobom, ktorý Alexandrescu v [1] nazýva „Intrusive reference counting“. Na rozdiel od iných spôsobov je počet referencí na objekt uložený priamo v objekte. Ako Alexandrescu uvádza, nevýhoda tohto spôsobu je, že vyžaduje spoluprácu objeku. Preto všetky objekty, na ktoré môže ukazovať counted pointer, sú potomkami triedy `CountedPtrTarget`. Táto trieda má metódy `CPTarget_Keep()` a `CPTarget_Free()` ktoré zväčšujú resp. zmenšujú počet ukazujúcich pointrov a prípadne mažú objekt. Pritom sa objekty tejto triedy samy dealokujú, preto musí byť vždy vytvorená dynamicky, čo je zaistené deklarovaním deštruktora ako `protected`.

Trieda `CountedPtrNT` je counted pointer na `CountedPtrTarget`, šablóna

```
template <typename type, bool ref> class CountedPtr
```

je counted pointer na `type`, pričom ak `ref==true`, nemôže mať

9 to je prípad triedy `ItemList` a k nej pomocnej triedy `ItemListBackPointer`

10 príkladom zložitejšieho cyklu je situácia, ktorá aj nastáva: koreň (objekt `/`) je inštanciou objektu `/:cls:object`, ale z definície sa nemôže stať nedosažiteľný z koreňa, preto nenastáva situácia, že by mal zaniknúť.

hodnotu `NULL`, takéto odkazy budeme nazývať referenciami[11]. Táto šablóna tiež implementuje automatické konverzie pointrov povolené v C++, tj. z potomka na predka.

## 4.2 Atómy

Veľmi častá operácia je vyhľadanie položky objektu shellu. Vyžaduje porovnávanie hľadaného identifikátora s niekoľkými identifikátormi položiek objektu. Ak by boli identifikátory reprezentované ako reťazce, ich porovnávanie by malo lineárnu časovú zložitosť (vzhľadom k dĺžke reťazca). Preto sa reťazce nahrádzajú hodnotami, ktoré sa potom porovnávajú v konštantnom čase. Typická implementácia je takáto: Existuje modul, ktorý prekladá reťazce do čísel. Musí si pamätať všetky reťazce a k nim priradené čísla, aby k tomu istému reťazcu vracal to isté číslo. Okrem toho je žiadúce, aby vedel určiť, ktoré reťazce sa už nepoužívajú a zmazať ich, preto si musí počítať referencie. V tomto sa problém začína podobať na problém z kapitoli o counted pointers. Preto som zvolil takúto implementáciu:

Identifikátor je reprezentovaný triedou `Identifier`, ktorá je potomkom triedy `CountedPtrTarget`. `Atom` je `typedef` na counted pointer na `Identifier`. Trieda `AtomKeeper` má metódu `GetAtom(string)`, ktorá ako jediná vytvára objekty `Identifier`, a to iba v prípade, že daný reťazec ešťe nemá príslušný `Identifier`, ktorý by ho reprezentoval. V opačnom prípade vráti metóda `GetAtom` counted pointer na už existujúci `Identifier`, ktorý takto ostáva ako jediný. Ak sa potom majú porovnávať dva identifikátory, porovnajú sa ako pointre, pričom sa využíva to, že jazyk C zavádza lineárne usporiadanie nad pointrami[12].

11 V súbore `uniheader.h`, ktorý je inkludovaný skoro každým súborov v projekte, sú pre všetky používané counted pointry zavedené označenia (`typedef`) `PType`, kde `Type` je typ, na ktorý sa ukazuje a analogicky `RType` pre referencie. Používajú sa na to makrá `DEF_PTR` a `DEF_REF`.

12 pre vyhľadávanie v červeno-čiernych stromoch `std::set` je potrebný nielen operátor `==`, ale aj `<`

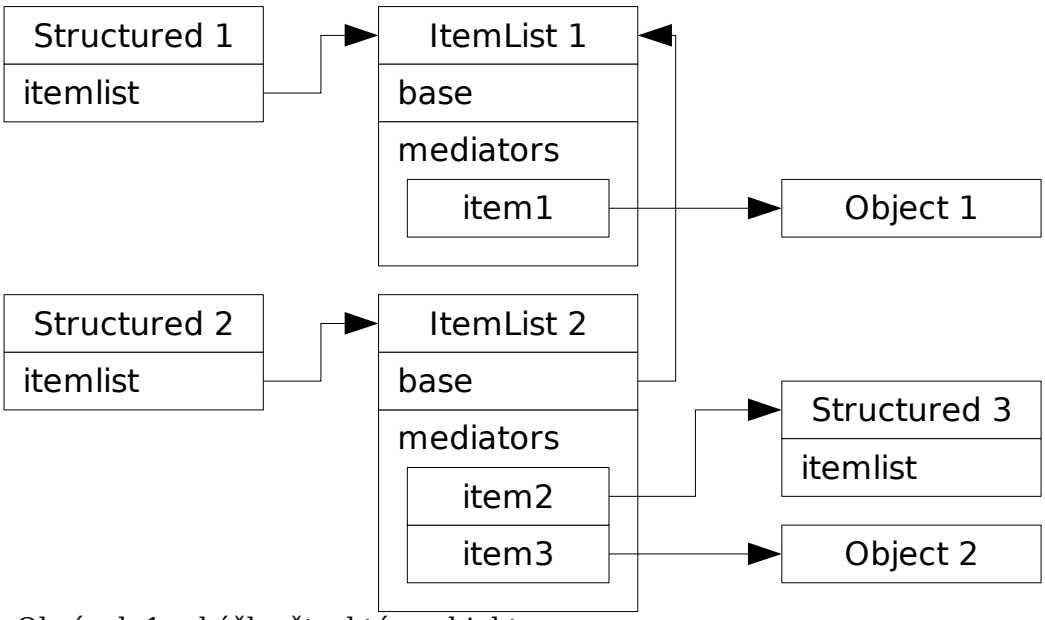
### Problém držania identifikátorov

Táto implementácia v sebe ale skrýva jeden problém, a to že `AtomKeeper`, ktorý si uchováva pointry na všetky objekty `Identifier`, si ich nemôže uchovávať ako counted pointry. Keby tak robil, počet referencí na `Identifier` by nikdy neklesol na 0 a identifikátory by sa hromadili. Tento problém som riešil zavedením tzv. ghost pointrov, čo sú pointry, ktoré sa nepočítajú; implementované sú ako normálne C pointry. Objekt typu `CountedPtrTarget` môže mať vlastníka typu `GhostCage`, ktorého pozná z konštruktoru. Keď počet counted pointrov ukazujúcich na `CountedPtrTarget` klesne na 0, objekt sa zmaže, ale ešte pred tým je podaná správa vlastníkovi. V prípade atómov je vlastník `AtomKeeper`, ktorý si svoj nepočítaný C pointer zmaže.

## *4.3 Vnútorná štruktúra objektov*

Všetky objekty okrem COM objektov a niektorých špeciálnych objektov (identifikátory, výnimky, zoznamy, ...), teda objekty, ktoré majú položky a podporujú dedičnosť, sú potomkami triedy `Structured`. Samotnú dedičnosť a vlastnenie položiek implementuje trieda `ItemList`, na ktorú má `Structured` odkaz.

Objekt `ItemList` obsahuje mapovanie identifikátorov na objekty a okrem toho má odkaz na iný `ItemList`, ktorý reprezentuje jeho predka vzhľadom k dedičnosti. Všetky objekty `ItemList` sú teda usporiadané v strome. Dedičnosť je implementovaná tak, že pri dotaze na položku `ItemList` najskôr zistí, či má mapovanie pre daný identifikátor, a ak nie, pošle dotaz rekurzívne na `ItemList`, ktorý je jeho predkom.



Obrázok 1: ukážka štruktúry objektov

V príklade na tomto obrázku je objekt `Structured2` potomkom objektu `Structured1`, pretože `ItemList2.base` ukazuje na `ItemList1`. Pri dotaze na položku `item1` objektu `Structured2` sa v `ItemList2` položka `item1` síce nenájde, ale dotaz sa rekurzívne odovzdá `ItemListu 1`, ktorý je predkom `ItemListu 2`, ten položku nájde. V tomto príklade má teda `Structured1` položku `item1` s hodnotou `Object1`. `Structured2` má okrem toho aj položku `item2` s hodnotou `Structured3` a položku `item3` s hodnotou `Object2`.

### Vytvorenie inštancie alebo klonu objektu

V prípade vytvárania inštancie vytvoríme nový `Structured` objekt s novým `ItemListom` a tomu ako predka dáme `ItemList` pôvodného objektu.

V prípade klonu dáme novovytvorenému objektu ten istý `ItemList`, ako pôvodnému objektu, a tento `ItemList` označíme príznakom „zdieľaný“. Keď sa bude mať nad ním previesť nejaká zmena, vloží sa mu potomok, a na ňom sa zmena vykoná.

### Dotaz

Aby bolo možné priradenie, dotaz nemôže vracať objekt, ale akúsi

referenciu naň. Tou je trieda `Mediator`, ktorá okrem odkazu na objekt obsahuje aj k nemu príslušný identifikátor. Mapovanie v `ItemListe` potom reprezentuje množina objektov typu `Mediator`.

Ak mapovanie pre príslušný identifikátor neexistuje a rekurzíve sa získa položka zdedená od predka, `ItemList` si pre daný identifikátor vloží ako položku inštanciu dedeného objektu, ktorú aj vráti. Takéto chovanie je nutné kvôli správnemu chovaniu v komplexných prípadoch[13]. Pre správne priraďovanie dotaz vracia odkaz na príslušný `Mediator` spolu s informáciou, ktorý objekt ho vydal, spolu obalené v triede `MedSpec`.

#### Priradenie

Ako ľavý operand má priradenie triedu `MedSpec`. Ak zistí, že objekt, ktorý ho vydal je odlišný od objektu v ktorom sa položka nachádza, priraďuje položku pôvodne dotazovanému objektu, čím sa vyhne zmene jeho predka a preťaží danú položku.

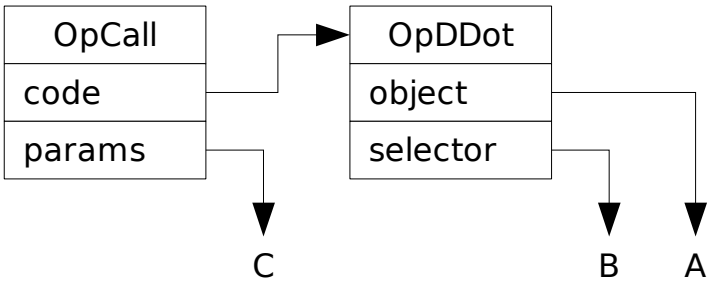
### *4.4 Parser*

Hlavnou úlohou tohto modulu je z príkazu vo forme textu vytvoriť vnútornú reprezentáciu príkazu. Tá je tvorená podľa návrhového vzoru Interpret, tak ako to popisuje GoF [2]: Každé odvodzovacie pravidlo (nižšie popísanej) gramatiky je reprezentované triedou, ktorá má ako položky odkazy na objekty zodpovedajúce jednotlivým symbolom na pravej strane daného pravidla. Tieto odkazy vytvárajú tzv. syntaktický strom. Napríklad trieda `OpDDot`, reprezentujúca pravidlo

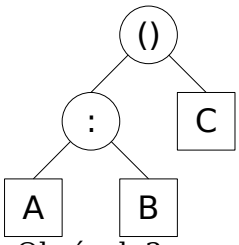
```
SELEKCIA -> VOLANIE : IDENTIFIKATOR
```

má položky `object` (odpovedajúci ľavému operandu) a `selector` (pravému). Podobne trieda `OpCall` má položky `code` a `params`.

13 konkrétne inak by po zadaní
`/:a:b:c:=@a; /:d:e:=/:a:b:new(); /:d:e:c:=@d; /:a:b:c:echo()`
echo vypísalo „d“, čo by znamenalo, že inštancia ovplyvnila triedu.




Obrázok 2: Ukážka vnútornej reprezentácie kódu




Obrázok 3: Ten istý kód schematicky

Reprezentácia kódu `A:B(C)` je znázornená na obrázku 2 a na obrázku 3 v schematickej forme, pričom `A`, `B` a `C` sú obecne tiež objekty reprezentujúce kód.

Modul `Parser` má dva podmoduly, oba implementované vnútri triedy `StdParser`. Prvý sa stará o delenie vstupu na tokeny, druhý spracováva postupnosť tokenov. Kvôli automatickému doplňovaniu sa musí kód spracovávať tak, ako po znakoch prichádza, preto sú podmoduly implementované automatmi, ktoré pracujú súčasne, a parser mení každým prichádzajúcim znakom svoj vnútorný stav.

## Delenie na tokeny

Problémom, ktorý tento podmodul rieši je spracovanie uvodzoviek a escape sekvencí. Napríklad výraz `ab:cd` tvorí 3 tokeny, a to `ab`, `:` a `cd`, ale `ab\:cd` a aj `a“b:cd“` tvoria jeden token, a to `ab:cd`. K rozlišovaniu sa používa konečný automat, ktorý v stave uchováva, či je za neuzavretou uvodzovkou, za spätným lomítkom alebo prípadnú kombináciu uvedeného. Prijíma znaky funkciou `AcceptChar`, normálnymi znakmi buduje premennú `token`, v prípade znakov " alebo \ mení svoj stav a v prípade, že nie je medzi uvodzovkami ani za spätným lomítkom, t.j. je v stave `0`, a spracovávaný znak je špeciálny znak (napr. `:|,;{}`), príslušne zavolá `AcceptToken`, čo je funkcia implementovaná druhým podmodulom.

## Spracovanie tokenov

Jazyk `loshu` sa dá popísať touto bezkontextovou gramatikou:

```
BLOK           -> { PRIKAZY ; }
PRIKAZY        -> PRIKAZY ; PRIKAZ
PRIKAZ         -> PRIRADENIE
                  VOLANIE
SEKVENCIA      -> SEKVENCIA , VOLANIE
PRIRADENIE     -> VOLANIE : IDENTIFIKATOR := VOLANIE
VOLANIE        -> SELEKCIA ( SEKVENCIA )
                  IDENTIFIKATOR
SELEKCIA       -> VOLANIE : IDENTIFIKATOR
                  VOLANIE | IDENTIFIKATOR
                  VOLANIE / IDENTIFIKATOR
                  VOLANIE
                  BLOK
IDENTIFIKATOR -> TOKEN
```

Na spracovávanie tokenov sa používa zásobníkový automat s mechanizmom podobným bezkontextovej gramatike. Podmodul spracovávajúci tokeny používa zásobník objektov a paralelne zásobník horeuvedených neterminálnych symbolov gramatiky, pričom každý symbol je kódovaný znakom a celý zásobník je reprezentovaný reťazcom. Podľa konca reťazca potom parser zisťuje, ktoré pravidlo môže použiť. Napríklad keď reťazec končí `S(Q)`, parser vyberie zo zásobíka objektov 4 objekty, nahradí ich z nich zloženým objektom a `S(Q)` prepíše na `C`. Ak sa nenájde netriviálne prepisovanie pravidlo, aplikuje sa triviálne, t.j. napríklad `SELEKCIA -> VOLANIE`. Oba zásobníky majú vždy rovnakú veľkosť a znak a objekt v rovnakej hĺbke si odpovedajú.

### Automatické doplňovanie identifikátorov

Pre automatické doplňovanie, za ktoré je tiež zodpovedný parser, je podstatný len objekt na vrchole tohto zásobníka. Jednotlivé objekty na zásobníku reprezentujú časti kódu, ktoré ešte nemohli byť uzavreté, pričom vrchol reprezentuje najvnútornejší z nich. Preto, keď je rozpísaný napríklad príkaz `/:a:=(((/:a`, na spodku zásobníku je priradenie,

potom nasledujú 4 objekty odpovedajúce neuzavretým zátvorkám, na vrchu je objekt odpovedajúci výrazu `/:a`. Nápoveda, ktorú poskytuje automatické doplňovanie nezávisí od stupňa a typu vnorenia výrazu `/:a`[14].

Parser si teda vyberie objekt na vrchole svojho zásobníka, a ten objekt čiastočne vyhodnotí pomocou funckie `GetPrototype`. Tá v prípade operátorov `:`, `|` a `/` získava štandardným vyhodnocovaním prvý prvok zoznamu. Operátor volania však nevykonáva, pretože volanie metód by malo nežiadúce vedľajšie efekty. Miesto toho vyberá položku príslušnej metódy s názvom `_retval_`, ak existuje, tá sa považuje za prototyp návratovej hodnoty. Na objekt, ktorý vzíde z tohto vyhodnocovania, zavolá metódu `DoubleDotItems` alebo `SlashItems`[15], ktorá vráti zodpovedajúci zoznam identifikátorov.

## *4.5 Vykonávanie kódu*

### Zoznamy

Každé volanie sa vyhodnocuje na zoznam objektov. Takéto zoznamy postupne tečú cez pajpy. Preto je ideálna implementácia zoznamu na spôsob v prologu, tj. zoznam je buď prázdny zoznam (trieda `Null`), alebo odkazy na objekt (nazývaný hlava) a zbytok zoznamu (telo) zhrnuté v triede `List`. Tento prístup umožňuje postupne, v rámci spracovávania kódu s pajpou, odtrhávať zo začiatku zoznamu objekty s tým, že zbytok zoznamu ostáva zoznamom na ďalšie spracovanie. Navyše, telo nemusí byť ešte vyhodnotené, môže to byť ďalšia metóda, ktorá zoznam postupne generuje.

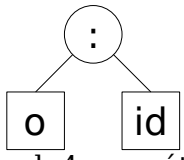
Základnou procedúrou nad kódom je teda rozdelenie na hlavu a telo, volá sa `HeadTail`. V klasickom prípade funguje tak, že volanie `HeadTail` na objekt, ktorý je metóda, vracia zoznam, ktorého hlava je

14 v skutočnosti ide často o objekt, ktorý je na zásobníku druhý zhora, pretože vrchol často reprezentuje posledný operátor (jeden z :|/). Výraz „na vrchole zásobníka“ je skôr naznačením istej lokálnosti tejto operácie.

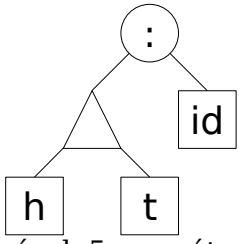
15 `DoubleDotItems` v prípade, že sa doplňuje za operátor `:` alebo `|`, `SlashItems` v prípade operátoru `/`.

metódou vracaný objekt a telo je táto metóda sama, v prípadne zmenenom stave. Ak je nejaká metóda, operátor, alebo iný kus kódu požiadaný cez pajpu o objekt, typicky to znamená, že tiež zavolá `HeadTail` na objekt jemu podriadený[16], a spracuje hlavu vráteného zoznamu. Takto volanie `HeadTail` spúšťa kaskádu volaní v ktorej si každá časť vykonávaného kódu žiada objekt na spracovanie od predchádzajúcej časti kódu.

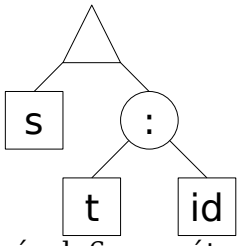
Tento systém ilustrujeme na príklade operátora „`:`“. Tento operátor je reprezentovaný C++ objektom, ktorý má dve položky: ľavý a pravý operand. Pred zavolaním `HeadTail` je v stave znázornenom na obrázku 4. Najprv zavolá `HeadTail` na svoj operand `o`. Ten môže vrátiť `Null` signalizujúci koniec zoznamu, v takom prípade aj operátor `:` ukončí výstupný zoznam vrátením `Null`. Ak naopak ľavý operand nie je prázdny zoznam, HeadTail vrátil hlavu a telo a situácia je taká, ako na obrázku 5. V takom prípade operátor nájde položku objektu `h` s identifikátorom `id`, a vráti ju v hlave zoznamu (označená „`s`“), pričom ako telo vráti sám seba s telom pôvodného ľavého operandu ako novým ľavým operandom. Tento výsledok volania `HeadTail` na operátor „`:`“ je znázornený na obrázku 6. Týmto sa tvorí zoznam o úroveň vyššie zo zoznamu na nižšej úrovni.



Obrázok 4: operátor ":" pred rozdelením operandu



Obrázok 5: operátor ":" po spracovaní jedného objektu



Obrázok 6: operátor ":" po rozdelení ľavého operandu na hlavu a telo

## Operátor volania

Transformácia `HeadTail` nepostačuje na problém rozlišovania medzi zavolanou a nezavolanou metódou, resp. blokom. Ak totiž zadá užívateľ

16 v prípade metódy je to volajúci objekt, v pripade operátoru operand

príkaz `/:cls:string:echonl:={^|catnl|echo;}`, má sa priradiť blok a tento blok nemá byť zavolaný, preto musí `HeadTail` vrátiť jednoprvkový zoznam s blokom ako prvkom. Naopak, príkaz `/:a:={@retazec|catnl;}()` má blok zavolať a priradiť sa má návratová hodnota metódy. Preto ďalšia transformácia nad objektami je `SetCallParams`, ktorá vracia objekt, ktorý bude, na rozdiel od pôvodného objektu, na volanie `HeadTail` bude reagovať vyhodnocovaním. Napríklad objekt typu `OpBlock` vracia pri volaní `HeadTail` ako hlavu samého seba a ako telo prázdny zoznam, tj. objekt `Null`[17]. Pri volaní `SetCallParams` ale vracia objekt typu `BlockCall` a ten sa delí na hlavu a telo vykonávaním v bloku obsiahnutého kódu. Podobný prípad je operátor priradenia a k nemu príslušiaca metóda `SetAssignment`.

### Metódy

Mechanizmus `HeadTail` používajú aj niektoré metódy, konktrétne tie, ktoré vytvárajú z jedného pritečeného objektu veľké množstvo objektov. Ide napríklad o metódu adresára `_item_`, ktorá posiela do výstupnej pajpy položky adresára. Funguje tak, že otvorí[18] adresár a na každé zavolanie `HeadTail` z neho prečíta jednu položku. Podobne funguje metóda `lines`, ktorá postupne vracia riadky súborov. Tento mechanizmus je však zbytočne komplikovaný pre metódy iného typu, konkrétne tie, ktoré k jednému vstupnému objektu generujú len obmedzené množstvo výstupných objektov. Pre tieto metódy má trieda `Method` spoločnú implementáciu `HeadTail`, ktorá funguje tak, že metóda obsahuje frontu, z ktoréj vyberá objekty na vrátenie. Keď je fronta prázdna, načíta objekt zo vstupnej pajpy a zavolá naň funkciu `Process`, v ktorej sa jednotlivé vstavané metódy líšia. Vstavaná metóda shellu môže počas vykonávania `Process` volaním `Return` pridávať

17 rovnako to robia všetky objekty, ktoré nevykonávajú žiadny kód a ich hodnota sú ony samé, tj. napríklad čísla, reťazce, atď.

18 na Unixe `opendir` a `readdir`, na Windows `FindNextFile`, pričom ale používa spoločné rozhranie vytvorené triedou `DirList` (a jej potomkami). Metóda `_item_` je teda stavová, nesie v sebe štruktúru `DIR`, resp. `HANDLE`.

objekty do vyššie spomínanej fronty. Ak `Proces` nepridá do fronty žiadny objekt, buffer ostáva prázdny, `HeadTail` načíta ďalší objekt zo vstupnej pajpy a na ňom znova zavolá `Proces`[19].

## *4.6 Užívateľské rozhranie*

Shell má dve uzívateľské rozhrania. Jedno je triviálne, ktoré iba číta svoj štandardný vstup a píše na štandardný výstup, vhodné je najmä na spúšťanie skriptov. Druhé ponúka užívateľovi napríklad automatické doplňovanie identifikátorov a názvov súborov (ďalej len intelli sence). V tejto podkapitole sa budeme venovať problémom spojeným s implementáciu druhého z nich.

### Zobrazovanie

Spoločné rozhranie pre konzoly oboch operačných systémov je vytvorené triedou `Console`. Aj keď existujú knižnice, ktoré podporujú prácu s oknami na konzoli, napr. `ncurses` v Unixe, nie sú pre oba systémy jednotné. Preto spoločná abstrakcia, tvorená triedou `Console`, obsahuje v podstate len prebratie a odovzdanie terminálu a výpis textu na konkrétnu pozíciu. Nad touto abstrakciou sú postavený potomkovia triedy `Window`, ktorý majú na starosti zobrazovanie okien. Tých je 5, konkrétne posledný riadok, v ktorom sa zadáva príkaz, zbytok obrazovky zobrazujúci výstup, okno histórie, potvrdzovania príkazov, a okno v ktorom sa zobrazuje intelli sence. Základnou vlastnosťou okien je schopnosť reagovať na stlačené klávesy a vypísať svoj obsah do príslušného priestoru.

### Spúšťanie procesov

Hlavný problém pri implementácii režimu s užívateľským rozhraním bol, že Unixový terminál obecne nepodporuje čítanie už zapísaných

19 Metódy však musia nezabudnúť nakoniec vrátiť `Null` objekt, čím ukončia svoj výstup. Ak tak neurobia a `Process` už nikdy nezavolá `Return`, `HeadTail` sa zacyklí. Na druhú stranu sa môžu spoľahnúť na to, že na konci čítaného zoznamu sa tiež nachádza `Null` objekt, preto `Process` s `Null` objektom ako argumentom typicky volá `Return` s tým istým argumentom. Podobne metódy odovzdávajú cez `Return` argument volania `Process` aj v prípade výnimiek.

znakov. Samozrejme, znaky, ktoré zapisuje proces si môže zapisovať aj do svojej pamäti, nemôže ale zistiť, čo na terminál zapísal iný proces, napríklad ten, ktorý shell spustil. Preto shell nevie uviesť do pôvodného stavu časť obrazovky prepísanú pri zobrazovaní intelli sence alebo histórie.

Tento problém sa riešil tak, že shell pri spúšťaní procesov nedáva poslednému procesu ako štandardný výstup terminál, ale pajpu, z ktorej shell počas čakania na skončenie procesov priebežne číta zapísaný výstup, ukladá si ho a zároveň ho vypisuje na terminál. Ten je počas behu úlohy v štandardnom móde, tj. mimo módu spravovaného oknami užívateľského rozhrania `loshu`. Po skončení spustených procesov potom `losh` v okne určenom pre výstup môže zobrazovať zaznamenané dáta.

# Kapitola 5
# Používanie aplikácie

Výsledkom implementácie je spustiteľná aplikácia, ktorá sprístupňuje užívateľovi objektovo orientovaný prístup pre úlohy shellu, a to vo forme jazyka navrhnutého v 3. kapitole. V tejto kapitole popíšeme spôsob jej používania. Aplikácia je určená pre Unix a Windows, pričom na oboch systémoch sa správa identicky, pokiaľ nie je v tejto kapitole uvedené inak.

## 5.1 Inštalácia

Inštalácia pozostáva v skopírovaní hotovej aplikácie na disk, taktiež je možné jej skompilovanie zo zdrojových súborov, ktoré sa tiež nachádzajú na CD, ktoré je prílohou tejto práce. Návod na kompliáciu je v súbore `README.txt` v priečinku so zdrojovými súbormi. Na spúšťanie aplikácie v užívateľskom režime (režimy sú popísané v ďalšom odstavci) na Unixe je potrebné mať nainštalovanú knižnicu `ncurses`.

## 5.2 Spustenie, režimy a módy

Shell je možné spustiť v dvoch užívateľských režimoch. Prvý z nich nemá žiadne užívateľské prostredie, s konzolou pracuje len prostredníctvom štandardného vstupu a výstupu a je vhodný skôr na spúšťanie shellu inými procesmi. Druhý ponúka automatické doplňovanie a históriu príkazov. Na spustenie v prvom režime zadáva užívateľ do argumentov pri spustení `--noui`, v druhom `--ui`. Ak užívateľ zadá medzi argumenty `--run` nasledovaný cestou k súboru so skriptom, tento skript sa vykoná[20]. Okrem toho sa v oboch režimoch vykoná aj skript, ktorého cesta je uvedená v premennej environmentu `LOSH_STARTUP`, ak je táto premenná nastavená.

20 kombinácia použitia `--run` a `--ui` alebo `--noui` najprv spustí skript a po jeho skončení príslušné užívateľské rozhranie. `--run` bez `--ui` a `--noui` iba vykoná skript a skončí. Ak je shell spustený bez parametrov, chová sa ako s parametrom `--ui`.

Počas behu shellu je možné voliť medzi vykonávacími módmi, ktoré sú:

- test: každý príkaz na spustenie procesov sa iba vypíše, ale nevykoná
- ask: na to, či sa majú procesy spustiť sa shell spýta užívateľa
- real: spúšťa príkazy

Na prechod medzi týmito módmi slúžia príkazy `/:mode:test()`, `/:mode:ask()` a `/:mode:real()`.

### Používanie shellu v režime s užívateľským prostredím

Písaný príkaz sa zobrazuje v poslednom riadku konzoli. Počas písania sa zobrazujú prípadné možnosti automatického doplňovania, na výber ktorých sa užívateľ používa šípky nahor a nadol a klávesu Enter. Automatické doplňovanie zobrazuje buď položky objektu alebo, v prípade písania cesty, priečinky a súbory obsiahnuté v danom priečinku, ktoré začínajú doteraz napísanou časťou názvu. Počas písania príkazu môže užívateľ tiež zobraziť stlačením F2 históriu predchádzajúcich príkazov, v ktorej sa tiež pohybuje šípkami nahor a nadol a položku vyberá klávesou Enter. Ak je zadaný príkaz, ktorý spúšťa procesy a shell je v móde „ask“, zobrazí sa okno v ktorom môže užívateľ klávesou „y“ odsúhlasiť vykonanie príkazu a klávesou „n“ vykonanie príkazu zamietnuť. Všetky príkazy odsúhlasí stlačením klávesy Enter a všetky zamietne klávesou Escape. Klávesy Page Down, Page Up, Home a End slúžia na posúvanie v okne výstupu.

## *5.3 Programovanie v Iosh*

Užívateľ na písanie príkazov používa jazyk podľa syntaxe a sémantiky popísanej v kapitole 3.3. Používa pritom nižšie popísané vstavané objekty a metódy

### 5.3.a Vstavané objekty a ich metódy

Každý objekt shellu je potomkom triedy object. Tá má tieto metódy:

| | |
|---|---|
| `new` | vracia novú inštanciu daného objektu |
| `count` | vracia počet objektov, ktoré jej natiekli pajpou |

Metódy triedy string, ktorej potomkami sú všetky reťazce

| | |
|---|---|
| `cat` | pripája za reťazce svoj argument |
| `catb` | pripája pred reťazce svoj argument |
| `catnl` | pripája znak konca riadku na koniec reťazcov |
| `match` | filtruje reťazce, ktoré matchujú wildcard uvedený v argumente |
| `write` | zapisuje reťazce do súboru, ktorý je uvedený v argumente |
| `fso` | konvertuje reťazce na cesty v súborovom systéme |
| `echo` | vypíše reťazce na výstup |

Metódy triedy integer:

| | |
|---|---|
| `to` | generuje postupnosť čísel počnúc číslom na ktorom bola zavolaná a končiac číslom v argumente |
| `sum` | vráti súčet čísel zo svojho vstupu |

Inštancie triedy fsobj[21] reprezentujú cesty v súborovom systéme. Táto trieda má nasledovné metódy

| | |
|---|---|
| `lines` | číta volajúci súbor a vracia postupne jeho riadky |
| `size` | vracia veľkosť súborov v byteoch. |
| `files` | filtruje cesty k existujúcim súborom |
| `folders` | filtruje cesty k priečinkom |
| `cd` | zmení aktuálny adresár |
| `mkdir` | vytvorí adresár |
| `move` | premiestni položky do priečinku, ktorý má uvedený v argumente |
| `copy` | analogicky k `move`, ale kopíruje |
| `name=` | premenuje súbor alebo priečinok |
| `delete` | zmaže súbor alebo priečinok |

21 skratka slovného spojenia „file system object“

Okrem toho, `integer` a `fsobj` sú triedy odvodené od triedy `string`, takže na inštanciách týchto tried je možné volať aj metódy triedy `string`. Používa sa automatická konverzia, ktorá `fsobj` nahrádza príslušnou cestou v súborovom systéme. To znamená, že príkaz

```
//home/* | catnl | echo
```

vypíše súbory a priečinky v priečinku /home. Použitie `catnl` v príkaze spôsobuje, že sa položky vypíšu každá na ďalší riadok.

Z objektu `/` sú tiež prístupné tieto metódy:

| | |
|---|---|
| `/:exec` | Vytvára príkaz na spustenie procesu |
| `/:exit` | Ukončí losh |
| `/:jobs:show` | Vypíše spustené úlohy[22] |
| `/:jobs:fg` | Pokračuje v úlohe danej číslom z argumentu na popredí |
| `/:jobs:bg` | analogicky k `fg`, ale na pozadí |
| `/:mode:test` | zmení mód na „test“ |
| `/:mode:ask` | zmení mód na „ask“ |
| `/:mode:real` | zmení mód na „real“ |
| `/:proc:show` | Vypíše spustené procesy |
| `/:proc:kill` | Ukončí proces, ktorého číslo získa z argumentu |

Objekt `/` je navyše, ako reprezentant koreňa adresárového stromu v Unixe, inštanciou triedy `fsobj`, preto má všetky metódy tried `fsobj` a `string`.

### 5.3.b Pridávanie metód

Všetky vstavané triedy sú prístupné ako položky v `/:cls`, čo umožňuje pridávať im metódy rovnako, ako ostatným objektom. Pridanie metódy sa vykonáva priradením bloku do položky objektu. Blok je kód uzavrený vo svorkových zátvorkách, pričom za jednotlivými príkazmi je bodkočiarka. Objektu teda pridáme metódu tak, že priradíme blok do

22 metódy v `/:jobs` realizujú job control a sú prístupné len na Unixe

položky s príslušným názvom. Napríklad príkaz

```
/:cls:string:echonl := {^|catnl|echo;}
```

pridá objektu `/:cls:string`, a teda triede `string`, metódu `echonl`. Vyvolanie jejto metódy, či už pomocou operátoru dvojbodky so zátvorkami alebo pajpy, na tomto objekte alebo na jeho potomkovi vykoná kód v bloku. Aby mohly byť užívateľovi poskytnuté návrhy v rámci automatického doplňovania, tejto metóde je potrebné pridať položku `_retval_`, ktorá označuje návratový typ. Príslušný príkaz by vyzeral takto:

```
/:cls:string:echonl:_retval_ := /:cls:string
```

Ako ďalší príklad uvedieme príkaz

```
/:cls:fsobj:rec := {^, ^/*:rec();}
```

ktorý triede `fsobj` pridá metódu `rec`, ktorá bude vracať obsah priečinku rekurzívne. Potom napríklad príkaz

```
/:cd | rec | size | sum | echo
```

potom vypíše veľkosť aktuálneho adresára, tj. súčet veľkostí všetkých súborov v danom priečinku a jeho podpriečinkoch.

## *5.4 Podpora COM objektov*

Shell podporuje možnosť vytvárania COM objektov implementujúcich rozhranie `IDispatch` a volanie ich metód. U objektov s rozhraním `ITypeInfo` navyše poskytuje užívateľovi automatické doplňovanie. Na tvorenie COM objektov slúži objekt `/:COMFactory`, ktorý operátorom `/` sprístupňuje class factory pre v systéme registrované COM triedy. Napríklad

```
/:COMFactory/Excel.Application
```

sa vyhodnotí na class factory vytvárajúcu inštancie aplikácie Excel. Novú inštanciu tejto triedy vytvoríme volaním `:instance()` na tejto class factory. Na COM objekte potom môže užívateľ volať metódy a získavať a nastavovať vlastnosti, pričom vždy používa syntax so

zátvorkami. Pri získavaní používa meno vlastnosti, pri nastavovaní identifikátor, ktorý vznikne pridaním znaku = na koniec mena vlastnosti. `:Method(args)` teda zavolá metódu `Method` a ako argumenty jej predá zoznam objektov, na ktorý sa vyhodnotí args[23]. `:Prop()` získa hodnotu vlastnosti a `:Prop=(arg)` ju nastaví. Pri volaní metód a získavaní hodnôt vlastností je podporované len vracanie objektov existujúcich v shelli, tj. reťazcov, čísel a iných COM objektov.

Ako príklad práce s COM objektami uvedieme príkazy

```
/:excel := /:COMFactory/Excel.Application:instance()

/:excel:Visible=(#1)

/:book := /:excel:Workbooks():Add()

/:sheet := /:book:ActiveSheet()

/:sheet:Range(@A1):Value=(@text)
```

ktoré postupne vytvoria COM objekt aplikáciu Excel, nastavením jej vlastnosti `Visible` ju zviditeľnia, vytvoria v nej nový zošit a do bunky A1 zapíšu text „text“.

## 5.5 Spúšťanie procesov

Ak spracovávanie príkazu vráti objekty, ktoré reprezentujú príkaz na spustenie procesov, tento príkaz v závislosti na nastavenom móde sa zachová tak, ako je to popísané v odstavci 5.2. Okrem toho, že niektoré metódy, ako napríklad `copy`, generujú takéto príkazy, je na spúšťanie procesov určená metóda `exec`. Ako argument pre túto metódu sa zadáva zoznam, ktorého prvý prvok je cesta k spustiteľnému súboru, alebo jeho názov v niektorom z adresárov uvedených v premennej `PATH`. Za ním nasledujú argumenty pre spustený proces. Objekt, ktorý reprezentuje príkaz na spustenie procesov, má tieto metódy

| | |
|---|---|
| `in` | presmeruje súbor z argumentu metódy na štandardný vstup |

23 na vytvorenie zoznamu z objektov je možné použiť operátor „,“, takže args môže je typicky v tvaru `#1, @text, #43`.

| | |
|---|---|
| `out` | presmeruje súbor z argumentu na štandardný výstup |
| `append` | ako `out`, ale zapisuje na koniec otvoreného súboru |
| `env` | spúšťanému procesu nastaví environmentálnu premennú[24] |
| `exec` | na koniec kolóny pripojí ďalší proces, pričom s predchádzajúcim procesom ho spojí pajpou |

Napríklad príkaz

```
/:exec(@cat) :in(//home/test) :exec(@wc, @-l)
```

spúšťa príkaz v `bash` zapísaný

```
cat < //home/test | wc -l
```

Samozrejme, tento zápis je kratší a prehľadnejší. Predpokladá sa však také používanie `losh`, že do štartovacieho skriptu si pre často vykonávané úlohy užívateľ vloží príkaz

```
/:cls:fsobj:lncount:={/:exec(@cat):in(^):exec(@wc,@-l);}
```

a potom bude volať len `//home/test | lncount`.

### Job control

Na Unixe je implementované job control, a to pomocou metód `/:jobs:show`, `/:jobs:fg` a `/:jobs:bg`. Úlohu spustenú na popredí môže užívateľ pozastaviť stlačením Ctrl-Z, čím sa vráti späť do prostredia shellu. Zoznam spustených úloh s im priradenými číslami, vypíše príkazom `/:jobs:show()`. Potom môže napríklad úlohu s číslom 3 nechať pokračovať na popredí príkazom `/:jobs:fg(#3)`, analogicky na pozadí volaním `bg`.

## *5.6 Príklady*

Pre ilustráciu uvedieme niekoľko príkladov. Príkaz

```
//home/* | folders | count | echo
```

vypíše počet podpriečinkov priečinku /home.

```
#1:to(#10) | sum | echo
```

24 implementované iba vo verzii pre Unix

vypíše 55

```
(@riadok1, @riadok2) | catnl | write(//home/test)
```

zapíše dva riadky do súboru /home/test.

```
//home/test | lines | catnl | echo
```

vypíše obsah súboru /home/test.

```
//home/testdir | mkdir
```

vytvorí adresár /home/testdir.

```
//home/testdir | cd
```

zmení aktuálny adresár.

```
//home/test | move(//home/testdir)
```

presunie súbor /home/test do adresára /home/testdir.

```
//home/testdir/*t | files | delete
```

zmaže súbory končiace znakom „t“ v adresári /home/testdir.

```
/:exit()
```

ukončí `losh`.

# Kapitola 6
# Záver

## 6.1 Skĺbenie koncepcii OOP a dávkového spracovania

Počas implementácie sa ukázalo, že kompromisy v tejto oblasti neboli zvolené veľmi vhodne. OOP v takejto podobe predstavuje prílišnú, až nežiadúcu, voľnosť. Nie veľmi koncepčná je možnosť posielania objektov rôznych typov cez pajpu do metódy, ktorá bola vybraná podľa prvého z nich. Ďalším príkladom je, že užívateľovi nič nebráni zadať príkaz obsahujúci `:count`, ktorý, na rozdiel od použitia operátora |, nedáva zmysel, pretože nevracia počet tečúcich objektov, ale prúd jedničiek. Slabá typovanosť teda nie je výhodou, ale obrátila sa proti užívateľovi. Okrem toho sa ukázalo, že do takto slabo typovaného jazyka nie je možné rozumne zaviesť `private`/`public` ochranu prístupu k položkám objektu. Užívateľ totiž môže pridať objektu metódu a obmedzenia tým obísť[25].

## 6.2 Vyhodnotenie projektu

V projekte sa veľmi dobre podarilo implementovať interpret daného objektovo orientovaného jazyka. Na druhú stranu, aby bol projekt prepracovaný, chýba mu väčšie množstvo funkcí a vstavaných metód. Napríklad pri špecifikácii sa plánovala podpora správy prístupových práv k súborom, v ideálnom prípade objekty sprístupňujúce prácu so súborovými právami v Access Control Listoch. Na druhú stranu v ostatných nástrojoch podobné možnosti tiež nie sú priamo obsiahnuté[26].

25 jedno riešenie by bolo zaviesť okamih uzamknutia, pričom k privátnym položkám by mohli pristupovať len metódy priradené pred týmto okamihom, túto koncepciu by bolo treba preskúmať hlbšie, pretože nie je jasné, či je možné rozvynúť ju do použiteľnej podoby. Nemá to však veľký význam, pretože je príliš komplikovaná v porovnaní s klasickým `private`/`public` v OOP.

26 v bashi a cmd sa volajú externé programy (napr. setfacl, cacls), Windows PowerShell sprístupňuje príslušné .NET objekty.

### Porovnanie s existujúcimi nástrojmi

`losh` spĺňa všetky základné funkcie nástrojov `bash` a `cmd`. Umožňuje programovanie s objektovo orientovaným prístupom, na druhú stranu spustenie úloh, pre ktoré si užívateľ nevytvoril metódy, tj. spustenie úloh zápisom napr. `/:exec(...):in(...)`, má v porovnaní s `bash` a `cmd` dlhší a asi neprehľadnejší zápis. Verzia pre Windows oproti `cmd` podporuje COM objekty, ale vo Windows sú aj iné nástroje na skriptovanie COM objektov, napríklad Windows Scripting Host. Ako som už napísal, `Windows PowerShell` je koncepčnejší a aj prepracovanejší. Výhodou `loshu` je, že je dostupný pre obe rodiny operačných systémov.

Naopak výhodou všetkých zaužívaných nástrojov, a to nielen vo sfére príkazových interpretov, ale obecne aplikácii, je, že sú užívateľovi známe a je na ne zvyknutý. Ich používanie tiež dávalo podnety na ich rozšírenia, napríklad aliasy v `bash`i.

Veľmi dôležitá vlastnosť nástrojov určených na tieto účely je ich stabilita, ktorá sa do detailov buduje tiež až používaním, pri ktorom sa objavujú aj veľmi špecifické chyby.

## *6.3 Modulárne orientované programovanie*

Jednou z možností, ako poskytnúť užívateľovi príkazového riadku pohodlie nových smerov v programovaní je opustiť priestor objektovo orientovaného a dokonca aj procedurálneho programovania. Ako sme uviedli, tieto princípy sú v mnohých ohľadoch protichodné s dávkovým prístupom. Idea modulárne orientovaného programovania pre účely shellu je takáto: Shell definuje moduly, ktoré majú typované rozhrania, tieto moduly pracujú s objektami, čo sú len dáta bez metód. Rozhrania môžu mať rôzny charakter, niektoré môžu sprostredkovávať jeden objekt, iné prúd, a to buď ťahaný, alebo tlačený[27], alebo môžu mať iný

27 pajpy v implementovanom shelli fungujú na princípe ťahania, tj. modul si pýta od svojho predchodcu ďalší objekt, o princíp tlačenia ide keď modul spracovaný objekt posiela nasledovníkovi. Oba tieto princípy sú výhodné v iných prípadoch a filozofia rozhraní umožňuje ich súčasné použitie

význam. Objekty normálne prístupné pomocou premenných je tiež vhodné nahradiť spájaním modulov[28].

Tento prístup je podľa môjho názoru omnoho lepší ako ten, ktorý bol použitý pri návrhu tohto projektu. Táto koncepcia viac konzistentná s dávkovým spracovaním, takže nadväzuje na tradíciu príkazových riadkov, pritom je nekonzistentná s OOP. Taktiež by jej implementácia bola jednoduchšia. Dôvod, prečo nebola použitá je, že tento názor, ako aj ucelený pohľad na túto koncepciu, som získal až v záverečnej fázi projektu. Nie je mi známa implementácia tejto koncepcie v jej principiálnej podobe, ale jej jadro je dobre viditeľné vo `Windows PowerShell`i.

Nekonzistencia s OOP pritom nie je nevýhodou. Rozvoj myšlienky modulárne orientované programovanie prináša užívateľovi tiež konfort, tak ako je to vidieť práve vo `Windows PowerShell`i.

28 napríklad modul filter, ktorý púšťa na výstup tie objekty, ktoré spĺňajú určitú podmienku. Táto podmienka sa typicky odkazuje na daný objekt. Podobný prípad je modul nahradzujúci for each cyklus. Problém sa dá, namiesto premennej, riešiť napojením podmienky späť na modul do ďalšieho rozhrania. Týmto vzniká štruktúra modulov vo forme obecného grafu, ktorého nevýhodou ale je linearizácia a zápis do príkazu.

# Literatura

[1] Alexandrescu A.: *Modern C++ Design*, Addison Wesley, 2001

[2] Gamma E., Helm R., Johnson R., Vlissides J.: *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, 1998