SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Evidenčné číslo: FEI-5384-64685

UNIVERZÁLNE, PLATFORMOVO NEZÁVISLÉ KONZOLOVÉ ROZHRANIE

DIPLOMOVÁ PRÁCA

2018 **Bc. Juraj Vraniak**

SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Evidenčné číslo: FEI-5384-64685

UNIVERZÁLNE, PLATFORMOVO NEZÁVISLÉ KONZOLOVÉ ROZHRANIE

DIPLOMOVÁ PRÁCA

Študijný program:	Aplikovaná informatika
Číslo študijného odboru:	2511
Názov študijného odboru:	9.2.9 Aplikovaná informatika
Školiace pracovisko:	Ústav informatiky a matematiky
Vedúci záverečnej práce:	RnDr. Igor Kossaczký, CSc.
Konzultant:	Rndr. Peter Praženica, Ing. Gabriel Szabó

Bratislava 2018 **Bc. Juraj Vraniak**

Základné údaje

Typ práce:	Diplomová práca
Názov témy:	**Akupunktúrne body – hľadanie, meranie a zobrazovanie**
Stav prihlásenia:	schválené
Stav témy:	schválené (prof. Dr. Ing. Miloš Oravec - Garant študijného programu)
Vedúci práce:	doc. Ing. Marek Kukučka, PhD.
Fakulta:	Fakulta elektrotechniky a informatiky
Garantujúce pracovisko:	Ústav informatiky a matematiky - FEI
Max. počet študentov:	1
Akademický rok:	2016/2017
Navrhol:	doc. Ing. Marek Kukučka, PhD.
Abstrakt:	Akupunktúra patrí medzi najstaršie liečebné praktiky sveta a je to jedna z kľúčových častí tradičnej čínskej medicín Keďže použitie liečebných metód odvodených od akupunktúry je stále viac rozšírené, z medicínskeho pohľadu nanajvýš aktuálne venovať sa základnému výskumu v tejto oblasti a pokúsiť sa objasniť základné fyziologick a biofyzikálne mechanizmy stojace za preukázanými klinickými efektami. Z prehľadu publikovaných elektrickýc vlastností akupunktúrnych bodov a dráh vyplýva potreba dôsledného overenia hypotézy elektrickej rozoznateľnos akupunktúrnych štruktúr. Očakáva sa, že môžu mať nižšiu impedanciu a vyššiu kapacitu oproti okolitým kontrolný bodom na pokožke. Výstupom mapovania pokožky budú 2D a 3D napäťové/impedančné mapy z povrchu tela. S tým prístupom bude možné nielen lokalizovať prípadný akupunktúrny bod, ale aj študovať jeho ohraničenie, povrchov elektrickú štruktúru či jeho veľkosť. Súčasťou výskumu je aj realizovanie meraní závislosti impedancie od frekvenc v akustickom pásme frekvencií 100 Hz – 20 kHz a vplyvu rôznych parametrov na rozoznávanie pozície, tva a štruktúry akupunktúrnych bodov.

Obmedzenie k téme

Na prihlásenie riešiteľa na tému je potrebné splnenie jedného z nasledujúcich obmedzení

Obmedzenie na študijný program

Tabuľka zobrazuje obmedzenie na študijný program, odbor, špecializáciu, ktorý musí mať študent zapísaný, aby sa mohol na danú tému

Program	Zameranie	Špecializácia
I-API aplikovaná informatika	I-API-MSUS Modelovanie a simulácia udalostných systémov	-- *nezadané* --
I-API aplikovaná informatika	I-API-ITVR IT v riadení a rozhodovaní	-- *nezadané* --

Obmedzenie na predmety

Tabuľka zobrazuje obmedzenia na predmet, ktorý musí mať študent odštudovaný, aby sa mohol na danú tému prihlásiť.

SÚHRN

SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Študijný program:	Aplikovaná informatika
Autor:	Bc. Juraj Vraniak
Diplomová práca:	Univerzálne, platformovo nezávislé konzolové rozhranie
Vedúci záverečnej práce:	RnDr. Igor Kossaczký, CSc.
Konzultant:	Rndr. Peter Praženica, Ing. Gabriel Szabó
Miesto a rok predloženia práce:	Bratislava 2018

Diplomová práca sa zameriava na analýzu existujúcich skriptovacích jazykov a návrh nového univerzálneho konzolového rozhrania, ktoré je zamerané na adminstrátorské úlohy. Práca v úvode analyzuje operačné systémy, existujúce skriptovacie jazyky, ako aj emulátory, ktoré sprístupňujú funkcionality platformovo špecifických jazykov pre ostatné platformy. Práca sa tiež venuje spôsobom prekladu zo zdojového kódu na bytcode, ktorý je následne spúšťaný na počítači. V ďalšej časti sa práca zaoberá návrhovými vzormi, ktoré budú využité pri vytváraní aplikácie, a umožnia vytvoriť lepšie čitateľný, udržiavateľný a modifikovateľný kód. V práci je zahrnutý opis implementácie navrhnutého riešenia, ako aj jeho následné testovanie. Výsledkom práce je plne funkčné administrátorské rozhranie, ktoré bude jednoducho rozšíriteľné pomocou pluginov, umožňuje pracovať v interaktívnom, ako aj skriptovacom móduse, na rôznych operačných systémoch podporovaných JVM.

Kľúčové slová: skriptovací jazyk, analýza prekladu, prekladač, plugin, architektúra, návrhové vzory

ABSTRACT

SLOVAK UNIVERSITY OF TECHNOLOGY IN BRATISLAVA
FACULTY OF ELECTRICAL ENGINEERING AND INFORMATION TECHNOLOGY

Study Programme:	Applied Informatics
Author:	Bc. Juraj Vraniak
Master's thesis:	Universal, platform independent console interface
Supervisor:	RnDr. Igor Kossaczký, CSc.
Consultant:	Rndr. Peter Praženica, Ing. Gabriel Szabó
Place and year of submission:	Bratislava 2018

Diploma thesis aims to analyze existing scripting languages and suggest new universal console interface, that aims focus on administrative tasks. Paper first analyze operating systems, existing scripting languages as well as emulators, which provides functionality of platform specific languages to other platforms. Paper also looks on ways how source code is translated to bytecode, which is later on run on PC. In the next part of paper look on design patterns, which will be used during creating of application and enable us to write more readable, maintainable and modifiable code. The output of work is fully functional administrative interface, which can be easily extended by plugins, which allow us to work in interactive as well as in scripting mode, on different operating systems which are supported by JVM.

Keywords: scripting language, translation analysis, translator, plugin, architecture, design patterns

Vyhlásenie autora

Podpísaný Bc. Juraj Vraniak čestne vyhlasujem, že som diplomovú prácu Univerzálne, platformovo nezávislé konzolové rozhranie vypracoval na základe poznatkov získaných počas štúdia a informácií z dostupnej literatúry uvedenej v práci.

Vedúcim mojej diplomovej práce bol RnDr. Igor Kossaczký, CSc.

Bratislava, dňa 15.5.2018

.....

podpis autora

Poďakovanie

Touto cestou by som sa chcel poďakovať vedúcim práce RnDr. Igorovi Kossaczkému, CSc, Rndr. Peterovi Praženicovi, Ing. Gabrielovi Szabóovi za cenné rady, odbornú pomoc, trpezlivosť a konzultácie pri vytvorení diplomovej práce.

Obsah

Zoznam obrázkov a tabuliek

Zoznam skratiek

API	Aplikačné rozhranie
GUI	Graphical User Interface
jar	Java Archive
JVM	Java Virtual Machine
OS	Operačný systém
PC	Personal Computer
resp.	respektíve

Zoznam algoritmov

Úvod

Diplomová práca sa zameriava na analýzu existujúcich skriptovacích jazykov a návrh nového univerzálneho konzolového rozhrania, ktoré je orientované na administrátorské úlohy. Výsledkom a zároveň cieľom práce je vytvoriť plne funkčné administrátorské rozhranie, ktoré bude jednoducho rozšíriteľné pomocou pluginov, umožňujúce pracovať v interaktívnom, ako aj skriptovacom móduse, na rôznych operačných systémoch podporovaných JVM.

V prvej časti práce vychádzame z teoretických poznatkov, charakterizujeme operačné systémy, ich rôznorodosť a softvérové vybavenie. Absencia jednotnej platformy na vytváranie skrípt, vo väčšine prípadov, vyžaduje duplikovanie alebo viacnásobnú implementáciu skriptu pre konkrétny operačný systém. Čiastočným riešením uvedeného problému je použitie skriptovacieho jazyka s podporou cieľových platform. Zásadným nedostatkom skriptovacích jazykov pri riešení problému je nedostatok syntaktických a funkčných konštrukcií, ktoré sú overené časom a široko používané, ako napríklad pajpa alebo presmerovanie štandardného a chybového výstupu. Okrem skriptovacích jazykov, sme otestovali emulátory, ktoré zabezpečujú preklad platformovo špecifického jazyka do jazyka spustiteľného na konkrétnej platforme. Ďalšia časť teoretickej analýzy podrobne charakterizuje preklad jazykov, kde sú opísané postupy fungovania kompilátorov programovacích jazykov. V nadväznosti na spracovanú problematiku sme špecifikovali prípady použitia aplikácie, stručne popísali programovací jazyk, vymedzili návrhové vzory, ktoré nám umožnili sprehľadniť zdrojový kód a na záver sme podrobne opísali návrh aplikácie spolu s prvotným návrhom tried.

Výsledkom diplomovej práce je administrátorské rozhranie, rozšíriteľné pomocou pluginov, ktoré umožňuje pracovať v interaktívnom, ako aj skriptovacom móduse, na rôznych operačných systémoch podporovaných JVM. Okrem uvedeného sú výstupom práce aj možnosti a predpoklady integrácie s ďalšími enterprise nástrojmi, a v neposledom rade poskytneme knižnicu obsahujúcu rozhrania, pomocou ktorých sa dá funkcionalita systému jednoducho rozšíriť.

1 Analýza

1.1 Operačné systémy

Informatika a informačné technológie je pomerne mladá vedná disciplína. Jej začiatky je možné datovať od druhej polovice dvadsiateho storočia, čo momentálne predstavuje takmer sedemdesiat rokov. Za tento čas informatika zaznamenala enormný rast vo vývoji hardvéru, ako aj softvéru. Operačný systém je základná časť akéhokoľvek počítačového sytému, predstavuje softvér, ktorý umožňuje počítačom pracovať. V poslednej dobe, oblasť operačných systémov prechádza rapídnymi zmenami, pretože počítače sa stali súčasťou každodenného života, a to od malých zariadení, napríklad v automobiloch, až po najsofisitkovanejšie servery nadnárodných spoločností. Aj napriek tomu, že v dnešnej dobe poznáme mnohé operačné systémy, v práci sa zameriame na Windows, Mac OS, Unix a Linux.[3]

1.1.1 Windows

Microsoft Windows uviedol svoje prvé operačné systémy v novembri roku 1985 ako nadstavbu MS DOS. Jeho popularita rýchlo rástla až vyvrcholila dominantným postavením na trhu v osobných počítačoch. V roku 1993 začal vydávať špecializované operačné systémy, ktoré prinášali novú funkcionalitu pre počítače používané ako servery.[4] Pre účely automatizácie sa na Windows serveroch používajú hlavne powershell scripty, písané v rovnomennom jazyku Powershell[5].

1.1.1.1 Podiel na trhu Microsoft je aj vzhľadom na svoju históriu najobľúbenejším operačným systémom v segmente osobných počítačov. Podľa webovej stránky statcounter.com[6] a netmarketshare.com[7] má 81,73% resp. 88,42% podiel na trhu.

Podľa w3techs.com[8] je serverový operačný systém Windows používaný na 32.0% počítačoch.

1.1.1.2 Predinštalovaný softvér Windows operačné systémy ponúkajú základný balík nástrojov a programov. Serverové aj neserverové verzie Windowsu ponúkajú Powershell, ktorý je dostupný od inštalácie. Oba systémy podporujú aj takzvaný Command prompt alebo príkazový riadok, ktorý je alternatívou k Powershellu. Akékoľvek ďalšie programy je potrebné stiahnuť a doinštalovať.

1.1.2 MacOs

Mac okrem iného ponúka serverovú verziu operačného systému pod názvom OS X Server, ktorý začal písať svoju históriu v roku 2001, avšak neteší sa takej obľube ako Windows, Unix alebo Linux server. OS X server nepoužíva špecifický skriptovací jazyk, pričom

poskytuje možnosť výberu skriptovacieho jazyka, ako napríklad: Python, JavaScript, Perl, AppleScript, Swift alebo Ruby. Každý z uvedených jazykov prináša určité plusy, ako aj mínusy.

1.1.2.1 Predinštalovaný softvér Predinštalovaný softvér pre developerov na Mac OS je Python, AppleScript, Ruby, Bash, Ojective-c. Donedávna bola štandardom aj Java, avšak Apple sa rozhodol pre radikálny krok vylúčiť Javu a propagovať Objective-c.

1.1.2.2 Podiel na trhu Popularita počítačov s predinštalovaným operačným systémom MacOs sa mierne zvyšuje, čo je možné vidieť aj na obrázku na konci sekcie operačných systémov. Podľa webovej stránky statcounter.com[6] a netmarketshare.com[7] mu patrí 13,18% resp. 9,19% na trhu. Serverové verzie MacOs podľa stránky w3techs.com[8] sú na menej ako 0.1% zariadeniach.

1.1.3 Unix

Patrí medzi prvé operačné systémy pre servery, ktorého vývoj začal v roku 1970 a v priebehu rokov vzniklo veľa nových verzií Unixu a Linuxu. V minulosti boli Unixové servery veľmi obľúbené, avšak v súčasnosti sú na ústupe, a to najmä kvôli vyšším obstarávacím a prevádzkovým nákladom. Pre účely Unixu sa vytvoril Unix shell, dostupný v rôznych obmenách, ktorý je často vyhľadávaným jazykom medzi administrátormi a automatizačnými programátormi.

1.1.3.1 Predinštalovaný softvér Na väčšine unixových systémoch je predinštalovaný Shell a Open JDK.

1.1.3.2 Podiel na trhu Podľa stránky w3techs.com[8] sa Unixové systémy používajú na rovných 68.0% počítačov.

1.1.4 Linux

Linux je všeobecný názov pre širokú zostavu Linux distribúcií, ktoré používajú Linux Kernel. Linux Kernel bol prvýkrát verejnosti predstavený v roku 1991 a odvtedy bol rozšírený na najviac platforiem. Momentálne je jediným používaným operačným systémom na TOP 500 superpočítačoch (mainframe). Skriptovacím jazykom pre Linux je Unix Shell resp. jeho najrozšírenejšia forma Bash.

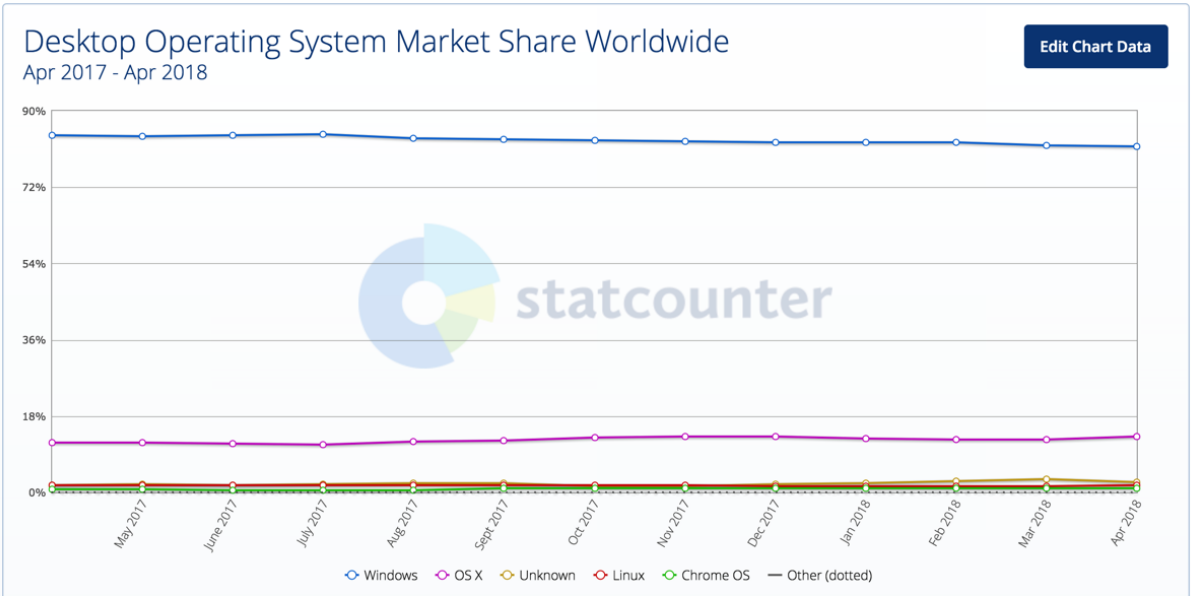
1.1.4.1 Predinštalovaný softvér Predinštalovaný softvér vo väčšine distribúciách Linuxu sú Bash, Open JDK - Java, niektoré distribúcie ponúkajú Python. RedHat začína s podporou .NET frameworku.

1.1.4.2 Podiel na trhu Linux je k dispozícii v mnohých formách, tak aby vyhovoval rôznym potrebám, od spotrebiteľsky orientovaných systémov pre domáce použitie až po

distribúcie použiteľné v špecifických odvetviach. Podľa webovej stránky statcounter.com[6] a netmarketshare.com[7] mu patrí 1,66% resp. 1,93%. Podľa stránky w3techs.com[8] je operačný systém Linux na 41.0% počítačoch.

1.1.5 Porovnanie podielov operačných systémov

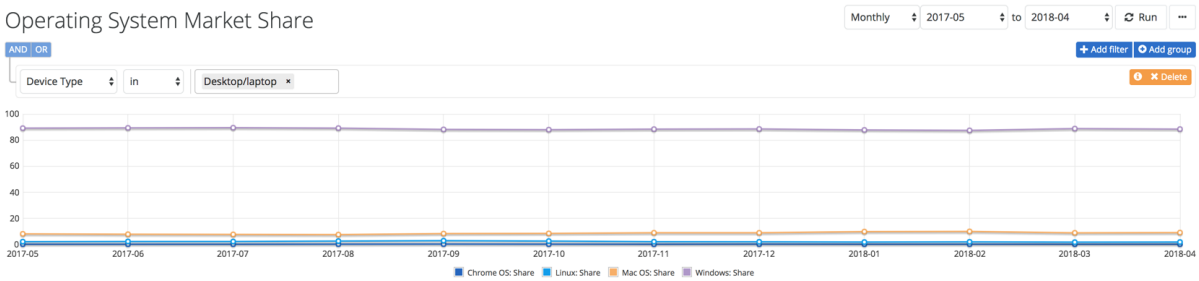
Nasledujúce grafy zobrazujú podiel operačných systémov na trhu v segmente osobných počítačov, ako aj v segmente serverov. Zaujímavý graf je vývoj trendu využívania serverových operačných systémov, z ktorého vidíme, že unixové a linuxové servery zvyšujú svoje podiely na trhu. Naopak, Windows v posledných mesiacoch stratil pár percent. Taktiež je vidieť, že linuxové a unixové systémy pokrývajú viac ako polovicu web stránok, ktoré majú najvyššie hodnotenie. Je potrebné dodať, že percentá na posledných dvoch obrázkoch pre Unix a Linux, nemožeme sčítavať, nakoľko na w3techs berú Linux ako podmnožinu Unix OS. Teda napríklad štatistika z predposledného obrázku hovorí, že podiel Unixu na trhu je 68% z toho väčšiu časť tvorí práve OS Linux s 41.1% podielom.



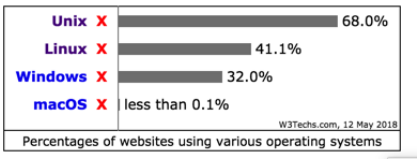
Obrázok 1: OS - podiel na trhu podľa statcounter[6]

1.2 Programovacie jazyky

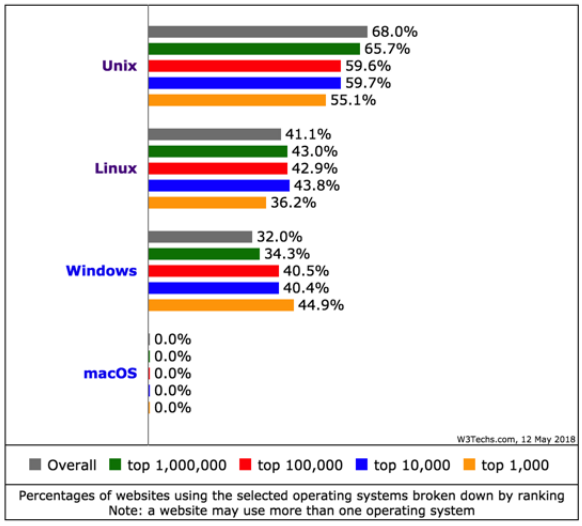
S príchodom osobných počítačov, no najmä serverov, sa programátori zaujímali o automatizáciu procesov, ktoré na danom stroji bolo spočiatku potrebné spúšťať manuálne. Keďže tieto úlohy neboli natoľko komplexné ako samotné programy, ktoré spúštali, bolo vhodné na úlohy využiť a vytvoriť skriptovacie jazyky. V nasledujúcej časti priblížime niekoľko programovacích jazykov, ktoré sa v dnešnej dobe bežne používajú na tvorbu



Obrázok 2: OS - podiel na trhu podľa netmarketshare[7]



Obrázok 3: Serverové OS - podiel na trhu podľa w3techs[8]



Obrázok 4: Serverové OS - podiel na trhu a rozdelenie podľa hodnotenia stránok podľa w3techs[8]

automatizovaných skriptov.

1.2.1 Shell

Je skriptovací jazyk pre unixové distribúcie, ktorý do súčasnosti prešiel rôznymi zmenami a rozšíreniami. Verzie Shellu predstavujú: sh, csh, ksh,tcsh, bash. Verzia Bash je momentálne najobľúbenejšou, avšak zsh je verzia Shellu, ktorá má najviac rôznych rozšírení funkcionality, ako aj veľa priaznivcov medzi developermi. V nasledujúcich častiach všeobecne zhodnotíme jednotlivé výhody a nevýhody skriptovacieho jazyka Shell.

1.2.1.1 Výhody

- automatizácia často opakujúcich sa úloh,
- možnosť zbiehať zložené príkazy, ako jednoriadkový príkaz - tzv. reťazenie príkazov,
- jednoduchý jazyk na používanie,
- výborne spracované manuálové stránky,
- Unix Shell je portabilný naprieč platformami Linuxu/Unixu,
- jednoduché plánovanie automatických úloh.

1.2.1.2 Nevýhody

- najväčšou nevýhodou je skutočnosť, že prirodzene nefunguje pod Windows OS, vo všeobecnosti je neprenosný medzi platformami, pričom na sprostredkovanie funkcionality používa rôzne emulátory a nástroje tretích strán,
- pomalé vykonávanie príkazov pri porovnaní s inými programovacími jazykmi,
- nový proces pre skoro každý spustený príkaz,
- zložitejší na zapamätanie rôznych prepínačov, ktoré dané príkazy podporujú,
- nejednotnosť prepínačov,
- Shell nepridáva vlastné príkazy, používa len tie, ktoré sú dostupné na konkrétnom počítači.

1.2.1.3 Popis a zhodnotenie jazyka

Unix Shell je obľúbeným scriptovacím jazykom, vhodným na automatizovanie každodenných operácií. Je jedným z najpoužívanejších skriptovacích jazykov, nakoľko všetky Linuxové, Unixové servery využívajú práve tento jazyk ako svoj primárny. V nasledujúcich častiach popíšeme Bash, ktorý je najrozšírenejšou verziou Unix Shell. Zaujímavou prednosťou jazyka je pajpa. Pajpa je klasický príklad vnútro-procesorovej komunikácie: odovzdáva štandardný výstup stdout procesu na štandardný vstup stdin iného procesu, viď príklad.

```
zarrelli:~\$ ls -lah | wc -l
35
```

V uvedenom príklade sme vylistovali obsah adresára, v ktorom sa práve nachádzame, a výstupom z programu sme naplnili štandardný vstup aplikácie "wc", ktorá spočíta, koľko riadkov sa nachádza na vstupe, ktorý jej bol dodaný. Príkaz za znakom pajpy | zbieha v Subshell-i, čo znamená, že nebude schopný zmodifikovať hodnoty v rodičovskom procese. Zlyhanie príkazu v pajpe vedie k takzvanej "zlomenej pajpe", v tomto prípade exekúcia príkazov skončí. [1]

Taktiež niektoré často používané príkazy majú zmenený spôsob zápisu. Ako príklad si uvedieme príkaz for, pri ktorom bash používa následovnú syntax:

Algoritmus 1 Bash ukážka rôznych volaní for cyklu. [1]

```
#!/bin/bash
# prvý spôsob zápisu podobná vylepšenej verzii z predchádzajúceho príkladu
for placeholder in list_of_items
do
action_1 \$placeholder
action_2 \$placeholder
action_n \$placeholder
done
#kolekcia vo fore môze byť reprezentovaná vymenovaním prvkov
#priamo za "in" časťou
for i in 1 2 3 4 5
do
echo "\$i"
done
# c-like prístup
for ((i=20;i > 0;i--))
{
if (( i % 2 == 0 ))
then
echo "\$i is divisible by 2"fi
}
exit 0
```

Ako vidíme z príkladu, for používa podobnú Syntax ako ostatné jazyky, a ďalej ju rozširuje. Vyššie spomenuté použitia niesú jediné, kde druhý spôsob môže uľahčiť prácu napríklad pri prototypovaní skriptu, v ktorom Shell poskytuje možnosť vložiť parametre pre cyklus priamo z konzoly ako v nasledujúcom príklade.

Algoritmus 2 Bash ukážka volania skriptu s for cylom priamo z konzoly . [1]

```
#telo skriptu
#!/bin/bash
i=0
for cities
do
echo "City $((i++)) is: $cities"
done
exit 0

#následné volanie z konzoly
./for-pair-input.sh
Belfast Redwood Milan Paris
City 0 is: Belfast
City 1 is: Redwood
City 2 is: Milan
City 3 is: Paris
```

Syntax jazyka je náročnejšia na učenie, pretože používa rôzne prepínače, ktoré novému používateľovi nemusia byť sprvu jasné. V tabuľke uvádzame príklad prepínačov pre if, ktorý pre podmienkovú časť používa hranaté zátvorky namiesto okrúhlych, na ktoré sme zvyknutí z väčšiny programovacích jazykov. Je potrebné spomenúť, že napríklad Unix shell nepoužíva žiadne zátvorky v podmienkovej časti príkazu, na ukončenie podmienkovej časti sa používa bodkočiarka, čo spôsobuje problémy pri prenositeľnosti. If ponúka aj ďalšie prepínače, no zhodnotili sme, že pre ilustráciu budú postačovať aj príklady uvedené v tabuľke č. 1 a tabuľke č. 2. Najväčšia nevýhoda je, že Shell script nie je multiplatformový jazyk, a teda ak by sme mali prostredie, v ktorom servery bežia na rôznych operačných systémoch, potrebovali by sme poznať ďalší jazyk, ktorým by sme docielili rovnaké alebo aspoň podobné výsledky.

Reťazcové porovnanie	Popis
Str1 = Str2	Vráti true, ak sa porovnávané reťazce rovnajú.
Str1 != Str2	Vráti true, ak porovnávané reťazce nie sú rovnaké.
-n Str1	R Vráti true, ak reťazec nie je null resp. o dĺžke 0.
-z Str1	Returns true, ak reťazec je null resp. o dĺžke 0.

Tabuľka 1: Ukážka reťazcových prepínačov v podmienkovom výraze if [9]

Numerické porovnanie	Popis
expr1 -eq expr2	Vráti true, ak sú porovnávané výrazy rovné.
expr1 -ne expr2	Vráti true, if ak nie sú porovnávané výrazy rovné.
expr1 -gt expr2	Vráti true, ak je hodnota premmenej expr1 väčšia než hodnota premennej expr2.
expr1 -ge expr2	Vráti true, ak je hodnota premmenej expr1 väčšia alebo rovná hodnote premennej expr2.
expr1 -lt expr2	Vráti true, ak je hodnota premmenej expr1 menšia než hodnota premennej expr2.
expr1 -le expr2	Vráti true, ak je hodnota premmenej expr1 menšia alebo rovná hodnote premennej expr2.
! expr1	Operátor "!"zneguje hodnotu premennej expr1.

Tabuľka 2: Ukážka numerických prepínačov v podmienkovom výraze if [9]

1.2.2 Powershel/Classic command line

Command line je základným skriptovacím jazykom pre Windows distribúcie, ktorý poskytuje malé API pre svojich používateľov. Aj kvôli uvedenej skutočnosti Miscrosoft predstavil nový jazyk Powershell. Powershell je kombináciou príkazového riadku, funkcionálneho programovania a objektovo-orientovaného programovania. Je založený na .NET frameworku, ktorý mu zabezpečuje istú mieru flexibility. Výhody a nevýhody Powershell zhrnieme v nasledujúcich častiach.

1.2.2.1 Výhody

- bohaté API,
- výborne riešený run-time,
- flexibilita,

- veľmi jednoduché prepnúť z .NET frameworku,
- dokáže pridávať funkcionalitu používaním tried a funkcií z .NET knižníc.

1.2.2.2 Nevýhody

- bohaté API - nejednoznačné, kedy čo použiť,
- niektoré výhody jazyka sú až nevhodne skryté pred používateľmi,
- staršie verzie serverov nie sú Powershell-om podporované tak, ako novšie,
- horšia dokumentácia v porovnaní so Shell scriptom.

1.2.2.3 Popis a zhodnotenie jazyka Powershell je obľúbený medzi programátormi a administrátormi, ktorí pracujú pod operačným systémom Windows. Donedávna, kým Powershell fungoval na .NET frameworku, ho nebolo možné používať mimo operačných systémov Windows. Avšak s príchodom frameworku .NET Core sa situácia zmenila. Spomenutý frawework je momentálne Open source, jeho zdrojové kódy boli zverejnené a je možné do neho prispievať. Okrem iného, podporuje rovnaké alebo aspoň podobné štruktúry ako Shell script. V niektorých prípadoch poskytuje rovnaké príkazy, napríklad: mv, cp, rm, ls. Jedným zo zásadných rozdielov medzi Shellom a Powershellom je skutočnosť, že kým v Shelli sú pre vstup aj výstup používané textové reťazce, ktoré je potrebné rozparsovať a interpretovať, v Powershelli je všetko presúvané ako objekt. Ide o najzásadnejší rozdiel, nakoľko ostatné veci boli pravdepodobne navrhované v spolupráci s používateľmi Shell scriptu. [10]

Pre demonštráciu rozdielov pri odovzdávaní parametrov medzi príkazmi uvádzame príklad.

Algoritmus 3 Ukážka použitia pipe v Powershell. [2]

```
function changeName(\$myObject)
{
if (\$myObject.GetType() -eq [MyType])
{
//vypíš obsah premennej
\$myObject.Name
//zmeň reťazec pre atribút name
\$myObject.Name = "NewName"
}
return \$myObject
}

// Vytvorenie objektu s argumentom OriginalName a následné použitie funkcie
//PS> \$myObject = New-Object MyType -arg "OriginalName"
//PS> \$myObject = changeName \$myNewObject
//OriginalName
//PS> \$myObject.Name
//NewName
// Ukážka s využitím pipe
//PS> \$myObject = New-Object MyType -arg "OriginalName" | changeName
//OriginalName
//PS> \$myObject.Name
//NewName
```

1.2.3 Python

Do analýzy sme zahrnuli aj programovací jazyk Python. Výber Python-u nebol náhodný, kedže je jedným z najpopulárnejších programovacích jazykov súčasnosti. Je viacúčelový, patrí mezi vyššie programovacie jazyky, objektovo-orientovaný, interaktívny, interpretovaný a extrémne používateľsky prijateľný.[11]

1.2.3.1 Výhody

- je ľahko čitateľný, tým pádom ľahšie pochopiteľný,
- syntax orientovaná na produktivitu,
- multiplatformový - po inštalácii interpretera,

- obsahuje množstvo rôznych knižníc,
- Open source.

1.2.3.2 Nevýhody

- rýchlosť,
- slabšia dokumentácia,
- nevhodný pre úlohy pracujúce s vyšším množstvom pamäte,
- nevhodný pre viac-procesorovú prácu,
- nevhodný pre vývoj na mobilných zariadeniach,
- limitovaný prístup k databázam.

1.2.3.3 Popis a zhodnotenie jazyka Ako už bolo spomenuté, Python je jedným z najobľúbenejších jazykov súčasnosti, kde prevažnú časť komunity tvoria vedci, ktorí nemajú rozsiahle programátorské znalosti. Práve jednoduchosť, čitateľnosť a pochopiteľnosť jazyka sa značnou mierou podieľajú na tomto fakte. Rovnako Python nevyžaduje manažovať pamäť a iné netriviálne záležitosti nižších programovacích jazykov. Aj napriek tomu, že jazyk je objektovo-orientovaný, skripty sa v ňom píšu jednoducho. Poskytuje štruktúry ako pajpa, možnosť pracovať s procesmi, vytvárať triedy, inštancie, jednoducho prototypovať a simulovať rôzne problémy. Veľkou výhodou tohto jazyka je, že je Open source s veľkou komunitou, ktorá rada testuje nové vydania, nahlasuje problémy, tým pádom je jazyk rýchlejšie a kvalitnejšie vyvíjaný. Na Python-e vznikli zaujímavé webové frameworky, ako napríklad Django. Každá strana má dve mince, a ani Python nie je stopercentný. Tým, že predstavuje interpretovaný jazyk, neprekypuje rýchlosťou. Veľa ľudí sa zaoberá rýchlosťou jazykov, zisťujú efektívnosť pri rôznych úkonoch, ako napríklad cykly, volania funkcií, aritmetika, prístup k pamäti, vytváranie objektov. V nasledujúcich tabuľkách je možné porovnať rozdiel v rýchlosti jednotlivých testov, uvádzané v sekundách, nižšie hodnota znamená lepší výsledok.

Aj keď sme spomenuli viaceré nedostatky, asi najväčším je rýchlosť. Ďalším nedostatkom je kolísanie syntaxe medzi jednotlivými verziami jazyka, čo má za následok nevyhnutnosť

Jazyk	**Force field benchmark**	**Array reverse benchmark**	**Rolling average benchmark**
C++ (-O2)	1.892	4.367	0.005
Java 7	2.469	3.776	0.463
C# (normal)	10.712	14.071	0.621
JavaScript	16.159	13.162	1.312
Python 2	717.2	1485	71.550
Python 3	880.7	1466	81.143

Tabuľka 3: Porovnanie rýchlostí rôznych jazykov v sekundách[12]

úpravy už preddefinovaných skriptov. Aj keď Python podporuje verzovanie knižníc, na PC je možné mať iba jednu verziu. Výhodnejšie je mať všetky verzie danej knižnice a v hlavičke skriptov definovať verziu, ktorú je potrebné pre daný skript použiť. V takomto prípade by bola zabezpečená spätná kompatibilita, ktorú ale Python nemá.

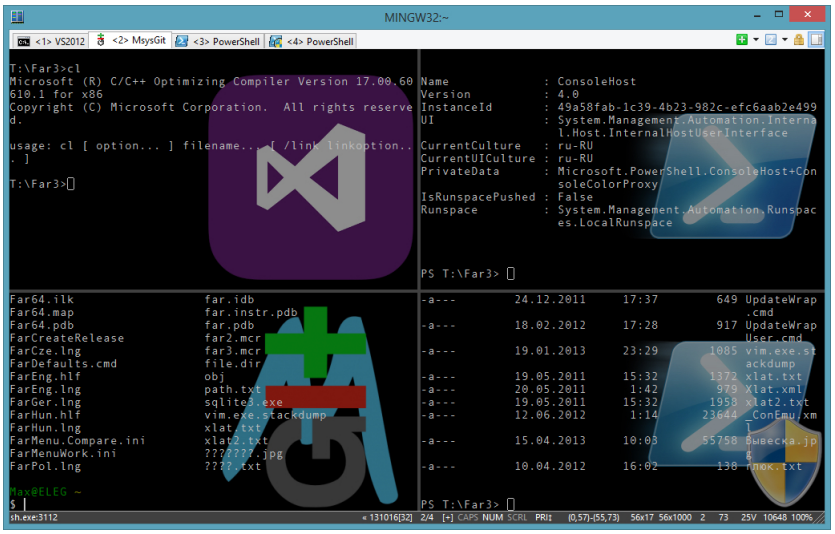
1.3 Existujúce riešenia

Existuje množstvo emulátorov a nástrojov tretích strán, ktoré sprostredkúvajú funkcionality Shell scriptu do Windowsu, niektoré z nich si predstavíme.

1.3.1 ConEmu

ConEmu je konzolový emulátor, ktorý poskytuje jednoduché GUI, do ktorého je možné vložiť viacero konzol. Dokáže spúšťať jednoduché GUI aplikácie, ako napríklad Putty, Cygwin. Obsahuje množstvo nastavení, ako nastavenie kurzora, priehľadnosti, písma a pod. Podporuje Windows 2000 a neskoršie verzie. Neposkytuje verziu pre iné operačné systémy. [13]

1.3.1.1 Skúsenosti ConEmu je vydarený emulátor, ktorý je schopný vykonávať akýkoľvek skript. Používaním sme neprišli na závažné nedostatky, ktoré by neboli popísané v zozname nevyriešených chýb na Github. Tak ako každý softvér, aj ConEmu je náchylný na chyby. Podľa issues logu sa do oficiálnych vydaní dostávajú rôzne problémy, ktoré neboli zahrnuté v predchadzajúcich verziách. V tomto prípade je na zvážení každého používateľa, či aj napriek problémom, ktoré sa môžu dostávať do jednotlivých verzií emulátora, použije ConEmu, resp. či jeho kladné stránky prekonajú tie záporné.

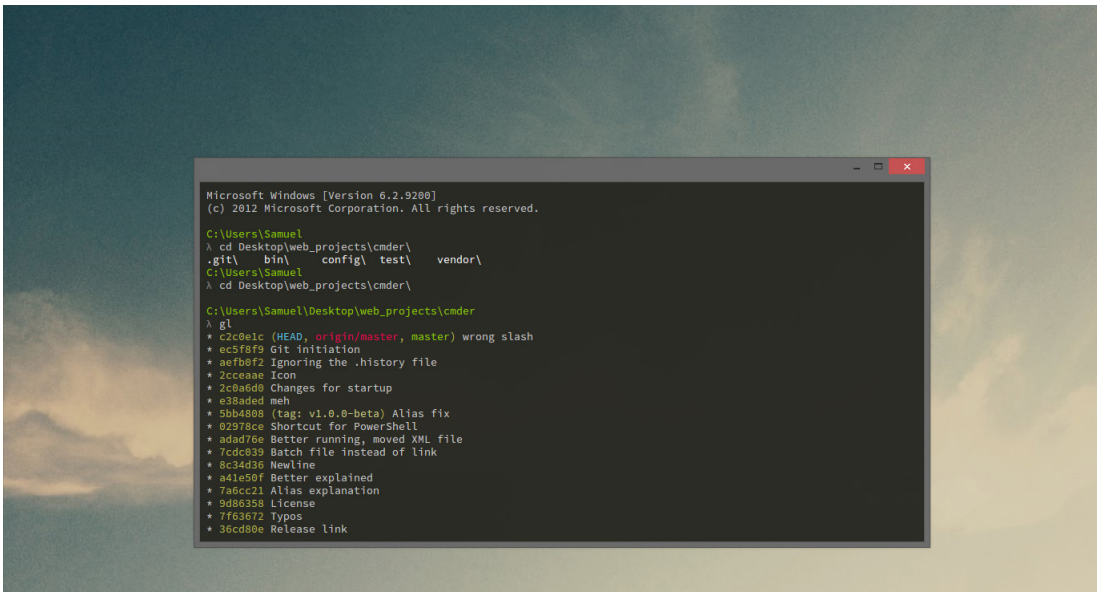


Obrázok 5: Ukážka ConEmu emulátora

1.3.2 cmder

Cmder je ďalším príkladom emulátora Shell terminálu. Vychádza z troch projektov ConEmu, Clink a Git pre Windows - voliteľná súčasť. ConEmu sme si predstavili v predchádzajúcej časti spolu s jeho kladmi a zápormi. Clink, konkrétne Clink-completions je v projekte využívaný na zvýšenie komfortu pri písaní skriptov, nepridáva ďalšiu Shell funkcionalitu. [14]

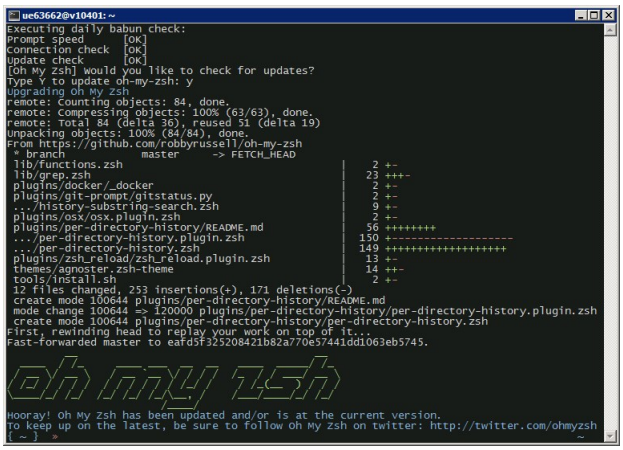
1.3.2.1 Skúsenosti ConEmu je príjemný nástroj, dokáže zjednodušiť prácu, obzvlášť ak je používateľ zvyknutý na programovanie v Shell scripte. Keďže Cmder používa ConEmu ako emulátor Shell terminálu, a je určený pre Windows platformu, nemožno hovoriť o multiplatformovom riešení.



Obrázok 6: Ukážka Cmder emulátora

1.3.3 Babun

Babun je jedným z mnohých emulátorov pre Windows, ktorý je nadstavbou cygwinu. Vo svojom jadre používa zshell a bash, ktoré sme popísali ako populárne medzi komunitou. Prináša vlastné GUI, ktoré dokáže zafarbovať text podľa zdrojového jazyka, čo zvyšuje prehľadnosť. Obsahuje git, svn, python, perl. Tiež má integrované sťahovanie nových balíčkov, ktoré ponúka cygwin pomocou kľúčového slova pact. Prenositeľnosť skriptov z unixových strojov je zabezpečená tým, že používa bash a zsh, avšak je to emulátor výhradne pre Windows distribúcie.[15]



Obrázok 7: Ukážka Babun emulátora

1.3.4 MobaXterm

Poskytuje množstvo funkcionalít, avšak je zaťažený licenciou v hodnote 50 eur. [16]

1.3.4.1 Neplatená verzia

- Plná podpora SSH a X serveru
- Vzdialená plocha (RDP, VNC, Xdmcp)
- Vzdialený terminál (SSH, telnet, rlogin, Mosh)
- X11-Forwarding
- Automatický SFTP prehliadač
- Podpora pluginov
- Možnosť inštalovateľnej alebo prenositeľnej verzie
- Plná dokumentácia
- Maximálne 12 spojení
- Maximálne 2 SSH tunely
- Maximálne 4 makrá
- Maximálne 360 sekúnd pre Tftp, Nfs a Cron

1.3.4.2 Platená verzia

- Všetky vymoženosti z neplatenej verzie Home Edition +
- Možnosť upraviť uvítaciu správu a logo
- Modifikovať profilový skript
- Odstrániť nechcené hry, šetriče obrazovky alebo nástroje
- Nelimitovaný počet spojení
- Nelimitovaný počet tunelov a makier
- Nelimitovaný čas behu pre sieťové daemony
- Podpora centrálneho hesla

- Profesionálna technická podpora
- Doživotné právo používania

1.4 Zhodnotenie analyzovaných technológií

Analýza ukázala, že väčšinový podiel na trhu osobných počítačov, ako aj serverov, tvoria Unix/Linux a Windows operačné systémy. Preto boli objektom analýzy hlavne programovacie jazyky, ktoré sú obľúbené medzi administrátormi daných jazykov. Ďalej sme preskúmali rôzne dostupné riešenia problému univerzálnej konzoly, z čoho usudzujeme, že ani jeden z produktov neposkytoval kompatibilitu na oboch alebo viacerých systémoch, maximálne kopíroval funkcie jedného systému do druhého. Na základe týchto poznatkov, ako aj poznatkov podrobnejšie rozpísaných v predchádzajúcich častiach, sme sa rozhodli pokračovať v analýze prekladu jazykov a získané vedomosti zúročiť do vlastného univerzálneho, platformovo nezávislého konzolového rozhrania.

2 Preklad jazykov

Pri programovacích jazykoch nás zaujímajú ich vyjadrovacie schopnosti ako aj vlastnosti z hľadiska ich rozpoznania. Tieto vlasnosti sa týkajú programovania a prekladu, pričom obe je potrebné zohľadniť pri tvorbe jazyka. V dnešnej dobe sa používajú na programovanie hlavne takzvané vyššie programovacie jazyky, môžeme ich označiť ako zdrojové jazyky. Na to aby vykonávali čo používateľ naprogramoval je potrebné aby boli pretransformované do jazyka daného stroja. Spomínanú transformáciu zabezpečuje prekladač, prekladačom máme na mysli program, ktorý číta zdrojový jazyk a transformuje ho do cieľového jazyka, ktorému rozumie stroj.[17]

2.1 Kompilátor proces prekladu

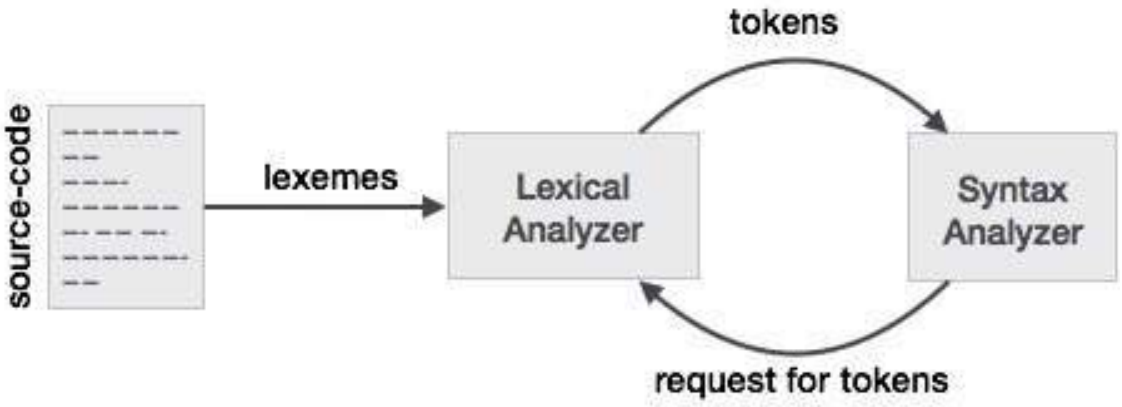
Aby bol preklad možný, musí byť zdrojový kód programu napísaný podľa určitých pravidiel, ktoré vyplývajú z jazyka. Proces prekladu je možné rodeliť na 4 hlavné časti.

- lexikálna analýza
- syntakticka analýza
- spracovanie sémantiky
- generovanie cieľového jazyka

Podrobnejšie si stručne popíšeme všetky štyri časti, ktoré majú pre nás z hľadiska prekladu najväčší zmysel.

2.1.1 Lexikálna analýza

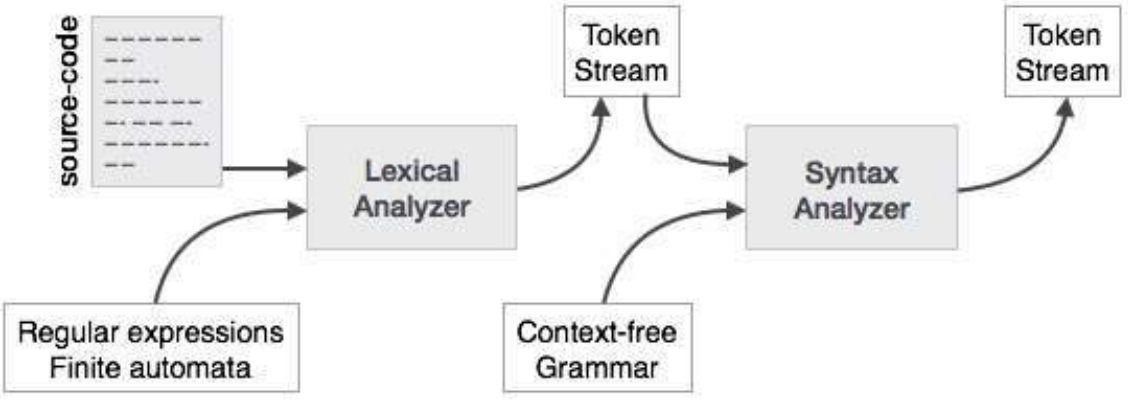
Lexikálna analýza je prvou fázou kompilátora. Dopredu napísaný zdrojový kód je postupne spracovávaný preprocesorom, ktorý vytvára takzvané lexémy.
Lexémou nazývame postupnosť alfanumerických znakov. Tieto postupnosti znakov sú následne vkladané do lexikálneho analyzátora, ktorý ma za úlohu vytvoriť zo vstupných lexém tokeny slúžiace ako vstup pre syntaktický analyzátor.
Tokeny sa vytvárajú na základe preddefinovaných pravidiel, ktoré sa v programovacích jazykoch definujú ako pattern. V prípade, že lexikálny analyátor nieje schopný nájsť pattern pred danú lexému musí vyhlásiť chybu počas tokenizácie.
Výstupom z lexikálnej analýzy sú takzvané tokeny, ktoré tvoria vyššie jednotky jazyka ako kľúčové slová jazyka, konštanty, identifikátory, operátory a iné.[17]



Obrázok 8: Ukážka práce lexikálneho analyzátora

2.1.2 Syntaktická analýza

Ďalšou fázou je syntaktická analýza. Úlohou Syntaktického analyzátora je kontrola správnosti vytvorených tokenov s uchovaním niektorých získaných informácií o štruktúre skúmanej syntaktickej jednotky. Syntaktická analýza sa radí medzi bezkontextové gramatiky. Po skoncení syntaktickej analýzy prichádza na rad sémantická analýza.[17]



Obrázok 9: Ukážka práce syntaktickeho analyzátora

2.1.3 Limitácia syntaktickej analýzy

Syntaktický analyzátor získa vstup z tokenu, ktorý vytvorí lexikálny analyzátor. Lexikálne analyzátory sú zodpovedné za validitu tokenu. Syntaktické analyzátory majú nasledovné limitácie.

- nedokážu zistiť validitu tokenu
- nedokážu zistiť či je token používaný pred tým ako je deklarovaný
- nedokážu zistiť či je token používaný pred tým ako je inicializovaný
- nedokážu zistiť validitu operácie, ktorú token vykonáva

2.1.4 Semantická analýza

Sémantická analýza má za úlohu interpretovať symboly, typy, ich vzťahy. Sémantická analýza rohoduje či má syntax programy význam alebo nie. Ako príklad zisťovania významu môžeme uviesť jednoduchú inicializáciu premennej.[17]

```
int integerVariable = 6

int secondIntegerVariable = "six"
```

Oba príklady by mali prejsť cez lexikálnu a syntaktickú analýzu. Je až na sémantickej analýze aby rozhodla o správnosti zápisu programu a v prípade nesprávneho zápisu informovala o chybe. Hlavné úlohy sémantickej analýzy sú":

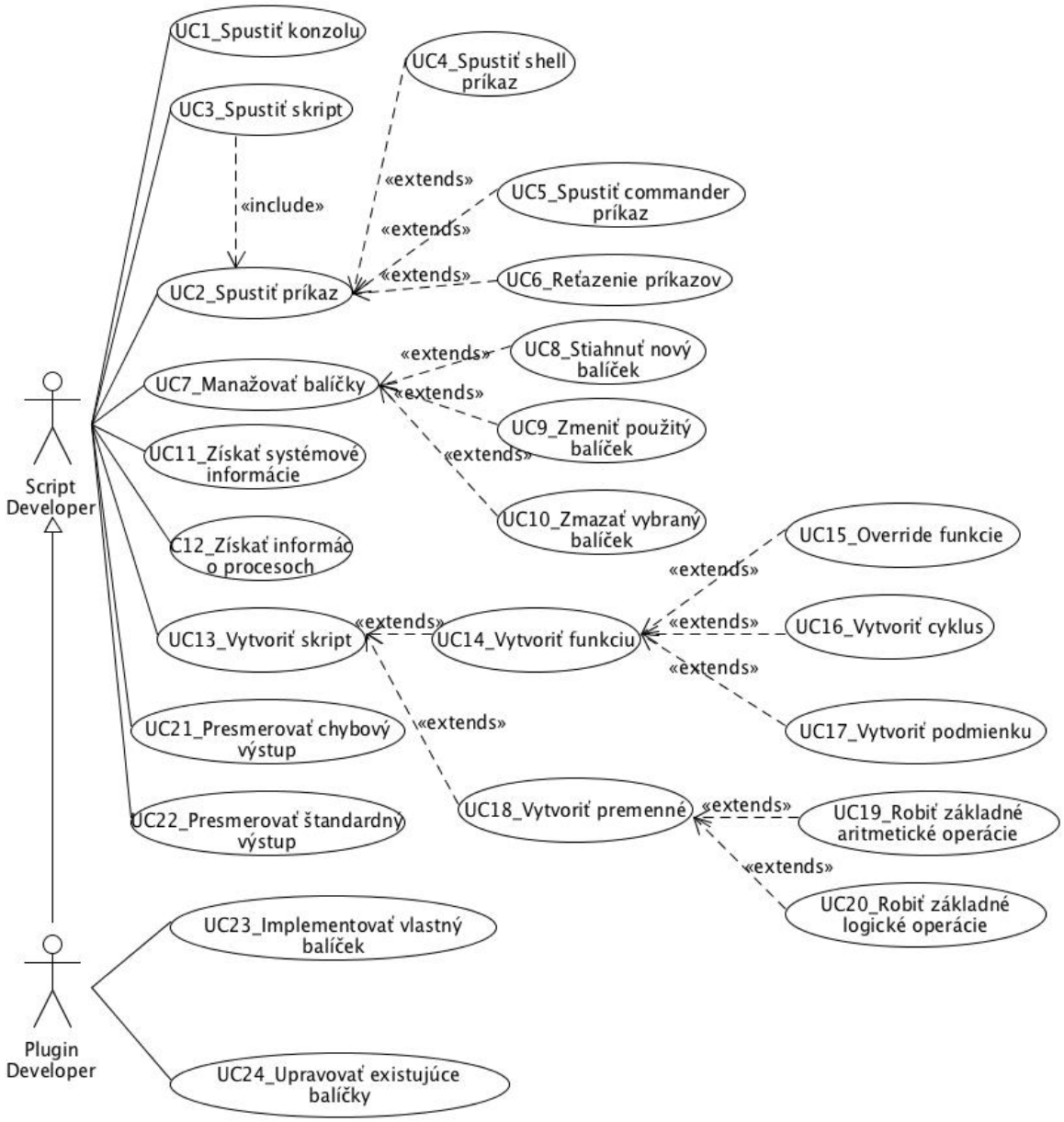
- zisťovanie dosahu definovaných tokenov takzvaný scoping
- kontrola typov
- deklaracia premenných
- definícia premenných
- viacnásobná deklarácia premenných v jednom scope

2.1.5 Generovanie cieľového jazyka

Generovanie cieľového jazyka môžeme považovať za poslednú fázu kompilátora. V tejto fáze sa preklápa jazyk z vyššieho jazyka do strojového jazyka, ktorý úspešne prešiel cez analyzačné časti .[17]

3 Návrh riešenia

3.1 Prípady použitia



Obrázok 10: Prípady použitia pre navrhovanú aplikáciu

3.2 Popis prípadov použitia

V tejto časti sa venujeme popisu jednotlivých prípadov použitia. Diagram prípadov použitia spolu s popisom sú základnými prvkami, na ktorých je možné špecifikovať novovznikajúci softvér. Je dôležité najpodstatnejšie časti systému špecifikovať na začiatku, aby pri navrhovaní aplikácie mohli byť prijaté rozhodnutia zaručujúce dosiahnutie najlepšieho výsledného riešenia vyhovujúceho špecifikácii. Ako je zjavné aj z priloženého diagramu prípadov použitia, pre aplikáciu sme identifikovali dvoch hráčov : Vývojár skriptov a Vývojár balíčkov. Títo hráči majú jednu spoločnú črtu - pre obe platí, že hráč je vývojár. Avšak je rozdiel medzi vývojárom skriptu a vývojárom balíčkov (nových súčastí systému), čo môžeme vyčítať z popisu konkrétnych prípadov použitia.

3.2.1 Vývojár skriptov

Rola sa zameriava hlavne na používanie hotovej aplikácie, prácu s balíčkami, vytváranie skriptov, efektívne využívanie dostupného API.

3.2.1.1 UC1_Spustiť konzolu

Use case	UC1_Spustiť konzolu
Podmienky	Používateľ musí disponovať stiahnutou aplikáciou.
Vstup	Nie je potrebný žiadny vstup od používateľa.
Popis	Konzolové rozhranie sa spustí.
Výstup	Konzola zobrazí základné údaje o konfigurácii.
Chyba	Konzola sa nespustí, musí však poskytnúť informáciu o chybe ktorá pri štarte nastala.

Tabuľka 4: Use case : Spustiť konzolu

3.2.1.2 UC2_Spustiť príkaz

Use case	UC2_Spustiť príkaz
Podmienky	Shell aplikácia musí byť spustená.
Vstup	Textový reťazec obsahujúci príkaz a jeho argumenty.
Popis	Používateľ zadá platný príkaz, následne získa výstup pre zadaný príkaz.

Výstup	Textový reťazec, ktorý sa v závislosti od programu mení v dĺžke a obsahu.
Chyba	V prípade zlyhania je používateľ informovaný o probléme, ktorý nastal.

Tabuľka 5: Use case : Spustiť príkaz

3.2.1.3 UC3_Spustiť skript

Use case	UC3_Spustiť skript
Podmienky	Shell aplikácia musí byť spustená a skript správne napísaný.
Vstup	Vstupom je skript, definujúci v hlavičke balíčky ktoré bude používať. Za nimi môže nasledovať čokoľvek od definície premenných, funkcií. V tele skriptu musí byť zadefinovaná metóda main(String args).
Popis	Vykonajú sa všetky príkazy tak, ako sú napísané v zdrojovom súbore.
Výstup	Výstup je textový reťazec, závislý na logike skriptu.
Chyba	V prípade chyby pri sťahovaní závislostí, exekúcie príkazov alebo iných komplikácií počas behu, program zapisuje na štandardný chybový výstup chybové hlášky spolu so základným popisom problému.

Tabuľka 6: Use case : Spustiť skript

3.2.1.4 UC4_Spustiť Shell príkaz

Use case	UC4_Spustiť Shell príkaz
Podmienky	Shell aplikácia musí byť spustená a skript správne napísaný. Taktiež musí byť v operačnom systéme ktorý podporuje Shell.
Vstup	Textový reťazec obsahujúci príkaz a jeho argumenty.
Popis	Používateľ zadá platný príkaz, následne získa výstup pre zadaný príkaz.
Výstup	Textový reťazec, ktorý sa v závislosti od programu mení v dĺžke a obsahu.

Chyba	V prípade zlyhania je používateľovi vratený chybový kód.

Tabuľka 7: Use case : Spustiť shell príkaz

3.2.1.5 UC5_Spustiť commander príkaz

Use case	UC5_Spustiť commander príkaz
Podmienky	Shell aplikácia musí byť spustená a skript správne napísaný. Systém musí mať nainštalovaný Windows commander.
Vstup	Textový reťazec obsahujúci príkaz a jeho argumenty.
Popis	Používateľ zadá platný príkaz začinajúci win alebo ext, následne získa výstup pre zadaný príkaz.
Výstup	Textový reťazec, ktorý sa v závislosti od programu mení v dĺžke a obsahu.
Chyba	V prípade zlyhania je používateľovi vratený chybový výstup z príkazového riadku.

Tabuľka 8: Use case : Spustiť powershell príkaz

3.2.1.6 UC6_Reťazenie príkazov

Use case	UC6_Reťazenie príkazov
Podmienky	Shell aplikácia musí byť spustená. Vstup musí byť zadaný v správnom formáte.
Vstup	Textový reťazec obsahujúci sekvenciu príkazov, ich argumenty spojené znakom pajpy |.
Popis	Systém rozozná, že ide o zreťazený príkaz a následne začne vykonávať príkazy v poradí v akom boli zadané. Jednotlivé príkazy odovzdajú výstupy nasledovníkovi po úspešnom ukončení. Príkazy sa vykonávajú dovtedy, pokým nepríde na posledný príkaz v sekvencii, alebo ak počas behu nastane chyba. O chybe je používateľ oboznámený a chyba je zapísaná na štandardný chybový výstup.

Výstup	Textový reťazec, ktorý sa v závislosti od programu mení v dĺžke a obsahu, výstup bude vygenerovaný posledným príkazom sekvencie.
Chyba	O chybe je používateľ oboznámený a chyba je zapísaná na štandardný chybový výstup.

Tabuľka 9: Use case : Reťazie príkazov

3.2.1.7 UC7_Manažovať balíčky

Use case	UC7_Manažovať balíčky
Podmienky	Shell aplikácia musí byť spustená.
Vstup	Textový reťazec obsahujúci príkaz pkg a jeho argumenty.
Popis	Používateľ bude schopný nahrať, zmazať, nahradiť vybraný balíček.
Výstup	Textový reťazec, ktorý sa v závislosti od programu mení v dĺžke a obsahu .
Chyba	O chybe je používateľ oboznámený a chyba je zapísaná na štandardný chybový výstup.

Tabuľka 10: Use case : Manažovať balíčky

3.2.1.8 UC8_Stiahnuť nový balíček

Use case	UC8_Stiahnuť nový balíček
Podmienky	Shell aplikácia musí byť spustená. Príkaz na stiahnutie balíčka musí byť správne zadaný.
Vstup	Textový reťazec obsahujúci príkaz "pkg download <názov balička s verziou>"
Popis	Program ako prvé skontroluje adresár balíčkov, či daný balíček nebol stiahnutý, ak nie stiahne nový balíček. V opačnom prípade medzi aktívne balíčky načíta používateľom zvolený balíček.
Výstup	Textový reťazec informujúci o úspešnosti sťahovania. Pre jeho načítanie je potrebný reštart aplikácie.

Chyba	Vypíše chybu na štandardný chybový výstup v prípade, že daný balíček na servery neexistuje, používateľ nemá internetové pripojenie.

Tabuľka 11: Use case : Stiahnuť nový balíček

3.2.1.9 UC9_Zmeniť použitý balíček

Use case	UC9_Zmeniť použitý balíček
Podmienky	Shell aplikácia musí byť spustená. Príkaz na zmenu používaného balíčka musí byť správne zadaný.
Vstup	Textový reťazec obsahujúci príkaz "pkg change <názov nahradzujúceho balíčka s verziou>
Popis	Program zmení používaný balíček z aktuálne používaného na balíček vybratý používateľom. Táto voľba je aplikovateľná iba pre spravovanie verzií existujúcich balíčkov. V prípade, že nahradzujúci balíček nie je dostupný lokálne, používateľ bude vyzvaný stiahnúť daný balíček.
Výstup	Textový reťazec informujúci o úspešnosti výmeny, alebo informujúci o potrebe stiahnutia balíčka.
Chyba	V prípade ak dôjde počas zmeny balíčkov ku chybe, bude zapísaná na štandardný chybový výstup.

Tabuľka 12: Use case : Zmeniť použitý balíček

3.2.1.10 UC10_Zmazať vybraný balíček

Use case	UC10_Zmazať vybraný balíček
Vstup	Textový reťazec obsahujúci príkaz "pkg delete <názov balíčka>.
Podmienky	Shell aplikácia musí byť spustená. Príkaz na zmazanie vybraného balíčka musí byť správne zadaný.
Popis	Program zmaže používateľom vybraný balíček z aktívnych balíčkov a následne ho fyzicky zmaže z disku.

Výstup	Textový reťazec informujúci o úspešnosti zmazania zadaného balíčka
Chyba	V prípade nesprávneho odstránenia balíčka z aktívnych balíčkov alebo pri následnom zmazaní zo súborového systému bude informácia o chybe presmerovaná na štandardný chybový výstup.

Tabuľka 13: Use case : Zmazať vybraný balíček

3.2.1.11 UC11_Získať systémové informácie

Use case	UC11_Získať systémové informácie
Vstup	Vstupom je textový reťazec "sysinfo".
Podmienky	Shell aplikácia musí byť spustená. Používateľ vloží platný príkaz na vyžiadanie systémových informácií.
Popis	Program vypíše na štandardný výstup informácie o využití systémových zdrojov, ako napríklad využitie procesora, využitie pamäte RAM, využitie oddielu swap a podobne.
Výstup	Výstupom je textový reťazec, formátovaný do riadkov. Každému riadku prislúcha jedna informácia, napr. CPU, ďalší riadok RAM atď.. V prípade viac jadrového procesora sa vypíšu informácie o každom z jadier.
Chyba	V prípade, že používateľ nemá právo na získanie informácií, program vypíše dôvod priamo na štandardný výstup. Rovnako program vypíše aj akékoľvek chyby, ku ktorým môže dôjsť počas behu.

Tabuľka 14: Use case : Získať systémové informácie

3.2.1.12 UC12_Získať informácie o procesoch

Use case	UC12_Získať informácie o procesoch
Vstup	Vstupom je textový reťazec "processes".
Podmienky	Shell aplikácia musí byť spustená. Používateľ vloží platný príkaz na vyžiadanie informácií o procesoch.

Popis	Program vypíše na štandardný výstup informácie o spustených procesoch, používateľoch, ktorí tieto procesy spúšťajú, koľko percent procesoru, pamäte RAM používajú.
Výstup	Výstupom je prehľadný výpis v podobe tabuľky, kde každý riadok zodpovedá jednému procesu. Nad jednotlivými hodnotami je hlavný riadok, ktorý popisuje o akú hodnotu ide.
Chyba	V prípade, že nie je možné získať informácie o procesoch, je táto skutočnosť zobrazená na stdout a popis chyby sa presmeruje na štandardný chybový výstup.

Tabuľka 15: Use case : Získať informácie o procesoch

3.2.1.13 UC13_Vytvoriť skript

Use case	UC13_Vytvoriť skript
Podmienky	Používateľ musí mať prístup k akémukoľvek textovému editoru.
Vstup	Vstupom musí byť správne napísaný skript.
Popis	Používateľ napíše skrip, ktorý bude prečítaný programom a vykonaný.
Výstup	Skrip vráti výstup svojho beho buď na štandardný výstup, alebo do súboru, v závislosti od toho ako je naimplementovaný.
Chyba	V prípade, že dôjde k menšej chybe, informácia bude zobrazená používateľovi, resp. presmerovaná do súboru.

Tabuľka 16: Use case : Vytvoriť skript

3.2.1.14 UC14_Vytvoriť funkciu

Use case	UC14_Vytvoriť funkciu
Podmienky	Používateľ musí mať prístup k akémukoľvek textovému editoru.

Vstup	Funkcia musí byť správne zadefinovaná. Syntax pre definovanie funkcie : function <návratový typ> <názov funkcie>(parametre funkcie)telo funkcie.
Popis	Používateľ napíše funkciu, ktorá bude prečítaná programom a vykonaná.
Výstup	Funkcia vracia premennú s definovanou návratovou hodnotou.
Chyba	V prípade, že nastane chyba pri exekúcii funkcie, program skončí a zapíše informácie o chybe na štandardný chybový výstup.

Tabuľka 17: Use case : Vytvoriť funkciu

3.2.1.15 UC15_Override funkcie

Use case	UC15_Override funkcie
Podmienky	Používateľ musí mať prístup k akémukoľvek textovému editoru.
Vstup	Nad funkciou je potrebné zapísať @Override, čo prekladaču povie, že má používať práve túto verziu funkcie.
Popis	Používateľ napíše funkciu, ktorá bude prečítaná programom a vykonaná. Navyše bude nahrádzať funkciu s rovnakým názvom.
Výstup	Premenná, ktorá je uvedená v definícii funkcie.
Chyba	V prípade zle zadefinovanej syntaxe je problém zapísaný na štandardný chybový výstup a vykonávanie skriptu je ukončené.

Tabuľka 18: Use case : Override funkcie

3.2.1.16 UC16_Vytvoriť cyklus

Use case	UC16_Vytvoriť cyklus
Podmienky	Používateľ musí mať prístup k akémukoľvek textovému editoru.

Vstup	Cyklus musí byť správne zadefinovaný. Syntax pre definovanie funkcie : for(<inicializácia premennej>;<podmienka pre spustenie>;<inkrement>)telo cyklu obsahujúce volania funkcií, príkazy, atď...
Popis	Používateľ napíše cyklus, ktorý bude prečítaný programom a vykonaná sa.
Výstup	Cyklus nemá žiadny výstup.
Chyba	V prípade, že nastane chyba pri parsovaní alebo exekúcii cyklu, program skončí a zapíše informácie o chybe na štandardný chybový výstup.

Tabuľka 19: Use case : Vytvoriť cyklus

3.2.1.17 UC17_Vytvoriť podmienku

Use case	UC17_Vytvoriť podmienku
Podmienky	Používateľ musí mať prístup k akémukoľvek textovému editoru.
Vstup	Podmienka musí byť správne zadefinovaná. Syntax pre definovanie podmienky : if(boolean value)telo podmienky obsahujúce volania funkcií, príkazy, atď...
Popis	Používateľ napíše podmienku, ktorá bude prečítaná programom a zohľadnená počas behu skriptu.
Výstup	Podmienka nemá žiadny výstup.
Chyba	V prípade, že nastane chyba pri parsovaní alebo exekúcii podmienky, program skončí a zapíše informácie o chybe na štandardný chybový výstup.

Tabuľka 20: Use case : Vytvoriť podmienku

3.2.1.18 UC18_Vytvoriť premenné

Use case	UC18_Vytvoriť premenné
Podmienky	Používateľ musí mať prístup k akémukoľvek textovému editoru.
Vstup	Premenná musí byť správne zadefinovaná. Syntax pre definovanie premennej : <typ> <názov premennej>; alebo <typ> <názov premennej> = <hodnota>; , kde hodnota môže byť konkrétna hodnota alebo iná premenná rovnakého typu.
Popis	Používateľ napíše inicializáciu alebo definíciu premennej, ktorá bude prečítaná programom a vykonaná.
Výstup	Program si uloží premmenú a jej hodnotu, ak bola definovaná.
Chyba	V prípade, že nastane chyba, používateľ bude informovaný o neúspechu na štandardný chybový výstup.

Tabuľka 21: Use case : Vytvoriť premenné

3.2.1.19 UC19_Vykonať základné aritmetické operácie

Use case	UC19_Vykonať základné aritmetické operácie
Podmienky	Používateľ musí mať prístup k akémukoľvek textovému editoru.
Vstup	Premenná musí byť správne zadefinovaná. Syntax pre definovanie /zmenu hodnoty premennej: <názov premennej> = <výraz>; alebo <typ> <názov premennej> = <výraz>; , kde výraz môže byť operácia nad číselnými hodnotami a číselnými premennými.
Popis	Používateľ napíše príkaz, ktorý bude prečítaný programom a vykonaný.
Výstup	Príkaz nastaví hodnotu premennej s vypočítanou návratovou hodnotou.
Chyba	V prípade, že nastane chyba pri exekúcii príkazu, program skončí a zapíše informácie o chybe na štandardný chybový výstup.

Tabuľka 22: Use case : Vytvoriť funkciu

3.2.1.20 UC20_Vykonať základné logické operácie

Use case	UC20_Vykonať základné logické operácie
Podmienky	Používateľ musí mať prístup k akémukoľvek textovému editoru.
Vstup	Premenná musí byť správne zadefinovaná. Syntax pre definovanie /zmenu hodnoty premennej: <názov premennej> = <výraz>; alebo <typ> <názov premennej> = <výraz>; , kde výraz môže byť operácia nad číselnými hodnotami, číselnými premennými, ako aj nad pravdivostnými. Vzťahy medzi číselnými hodnotami musia byť definované logickými operátormi - <, >, <=, >=, ==, !=. Vzťahy medzi pravdivostnými hodnotami musia byť definované logickými operátormi : ==, !=, ||, &&.
Popis	Používateľ napíše príkaz, ktorý bude prečítaný programom a vykonaný.
Výstup	Príkaz nastaví hodnotu premennej s vypočítanou návratovou hodnotou.
Chyba	V prípade, že nastane chyba pri exekúcii príkazu, program skončí a zapíše informácie o chybe na štandardný chybový výstup.

Tabuľka 23: Use case : Vytvoriť funkciu

3.2.1.21 UC21_Presmerovať chybový výstup

Use case	UC21_Presmerovať chybový výstup
Vstup	Pre presmerovanie na chybový výstup je potrebné dodržať syntax command stderr> file
Podmienky	Shell aplikácia musí byť spustená. Používateľ vloží platný príkaz na presmerovanie chybového výstupu.

Popis	Program presmeruje chybový výstup tam, kam mu používateľ v príkaze zadefinuje.
Výstup	Výstup programu predstavuje textový reťazec s popisom chyby, ktorá nastala.
Chyba	Ak by došlo ku chybe, chyba sa zapíše sa do logu aplikácie.

Tabuľka 24: Use case : Presmerovať chybový výstup

3.2.1.22 UC22_Presmerovať štandardný výstup

Use case	UC22_Presmerovať štandardný výstup
Podmienky	Shell aplikácia musí byť spustená. Používateľ vloží platný príkaz na presmerovanie štandardného výstupu.
Vstup	Pre presmerovanie na štandardný výstup je potrebné dodržať syntax command stdout> file
Popis	Program presmeruje štandardný výstup tam, kam mu používateľ v príkaze zadefinuje.
Výstup	Výstup programu predstavuje textový reťazec s výstupom zo skriptu alebo príkazu.
Chyba	Ak by došlo ku chybe, chyba sa zapíše sa do logu aplikácie.

Tabuľka 25: Use case : Presmerovať štandardný výstup

3.2.2 Vývojár balíčkov

Vychádzajúc z názvu role je zjavné, že tento hráč sa bude starať o vývoj aplikácie a jej funkcionalitu v zmysle rozširovania API, ktoré môže vývojár skriptov používať pre efektívnejšiu prácu.

3.2.2.1 UC23_Implementovať vlastný balíček

Use case	UC23_Implementovať vlastný balíček
Podmienky	Používateľ musí mať nainštalovanú Java SDK vo verzii 8, mať prístup k textovému editoru.

Vstup	Balíček obsahujúci všetky potrebné rozhrania, ktoré musí vývojár balíčka implementovať.
Popis	Používateľ implementuje novú funkcionalitu v Jave, následne všetky zdrojové súbory skompiluje a pridá do jar súboru určeného na ukladanie nových balíčkov.
Výstup	Balíček, ktorý je možné nahrať do aplikácie a používať ako jeden z príkazov.
Chyba	Chyba môže nastať pri vytváraní balíčka, kedy vývojára o chybe informuje prekladač jazyka, v ktorom je balíček implementovaný. V prípade neúspešného načítania je používateľ informovaný priamo v konzole na štandardný výstup.

Tabuľka 26: Use case : Implementovať vlastný balíček

3.2.2.2 UC24_Upravovať existujúce balíčky

Use case	UC24_Upravovať existujúce balíčky
Podmienky	Používateľ musí mať nainštalovanú Java SDK vo verzii 8, mať prístup k textovému editoru.
Vstup	Zdrojové súbory už existujúceho balíčka.
Popis	Používateľ upraví implementáciu alebo pridá novú funkcionalitu v Jave, následne všetky zdrojové súbory skompiluje a pridá do jar súboru určeného na ukladanie nových balíčkov
Výstup	Po úprave je balíček možné nahrať do aplikácie a používať ako jeden z príkazov.
Chyba	Chyba môže nastať pri vytváraní balíčka, kedy vývojára o chybe informuje prekladač jazyka, v ktorom je balíček implementovaný. V prípade neúspešného načítania je používateľ informovaný priamo v konzole na štandardný výstup.

Tabuľka 27: Use case : Upravovať existujúce balíčky

3.3 Výber programovacieho jazyka

Java je programovací jazyk a výpočtová platforma, ktorá bola vydaná spoločnosťou Sun Microsystems v roku 1995. [18] Programy v Jave sú prvotne preložené do tzv. byte-code, ktorý je rovnaký pre všetky PC. Pomocou jednoduchého programu je byte-code preložený do jazyka, ktorému rozumie konrétny PC. Java je objektovo-orientovaný programovací jazyk, čo znamená že rovnako ako v živote, aj v Jave je všetko tvorené objektami. Obsahuje široké spektrum knižníc, ktoré slúžia nielen na vývoj webových, ale aj desktopových aplikácií. Java rovanko podporuje dátové štruktúry, ktoré vo veľkej miere využívame pri vývoji aplikácie ako Collections, List, Annotation, String alebo Optional. Podporuje multitrading, čo umožňuje vytvárať efektívne programy pre počítače s viac-jadrovým procesorom.[18]

3.3.1 Nahrávanie tried v Jave

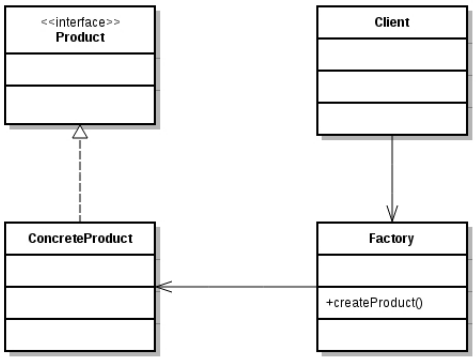
doplniť

3.4 Návrhové vzory

Návrhové vzory sú všeobecne opakovaným riešním pre všeobecne opakujúci sa problém pri dizajnovaní softwéru. Návrhový vzor nie je nemenný dizajn, vždy je potrebné aby si ho programátori prispôsobil podľa vlastných potrieb. Návrhové vzory sa delia do troch základných skupín vytváracie vzory, štrukturálne vzory a vzory správania. [19]

3.4.1 Factory - továreň

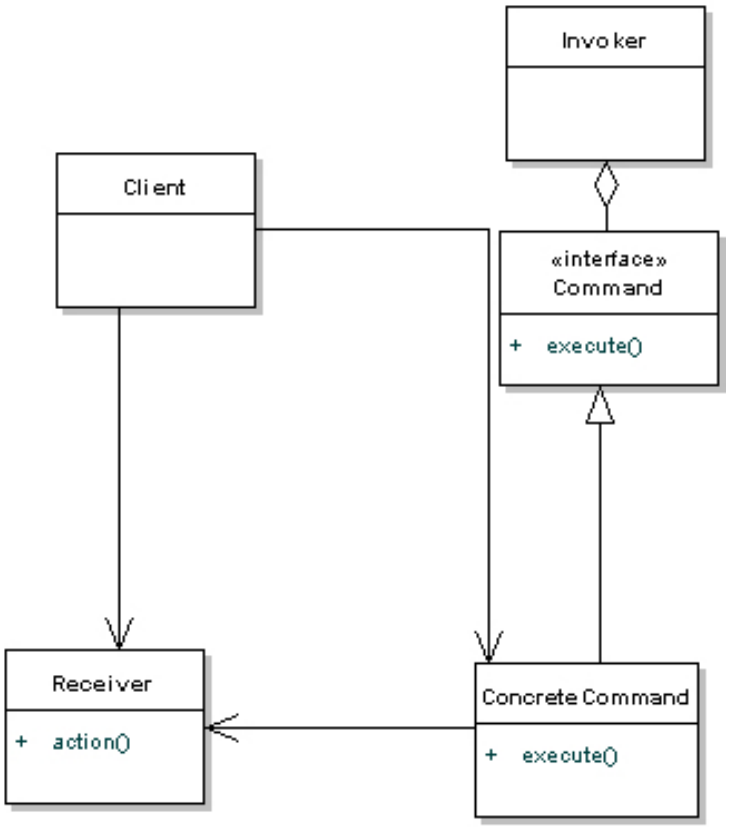
Factory návrhový vzor patrí do sekcie vytváracích vzorov. Pomocou tohoto vzoru budeme schopný vytvárať objekty bez toho aby sme prezradili logiku ich vytvárania klientovi.[19] Diagram návrhového vzoru je mozné vidieť na nasledujúcom obrázku.



Obrázok 11: Class diagram Factory návrhového vzoru

3.4.2 Command - príkaz

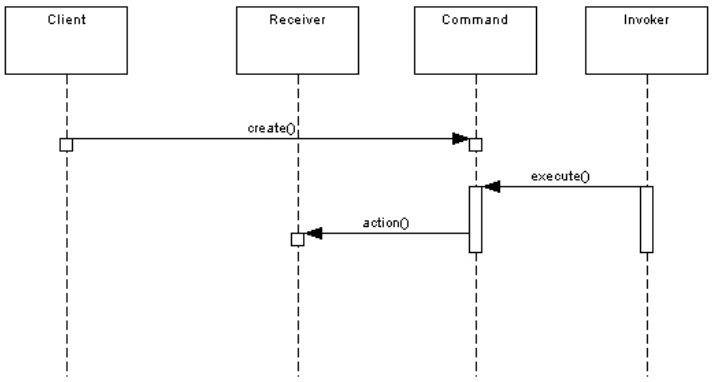
Command pattern je známy behaviorálny návrhový vzor, používa sa najmä na správu algoritmov, vzťahov a zodpovednosti medzi objektami. Cieľom vzoru je zapúzdriť požiadavku (request) ako objekt tým pádom parametrizovať klienta s rôznymi požiadavkami a zabezpečiť operáciu spať.[19]



Obrázok 12: Class diagram Command návrhového vzoru

Command vzor deklaruje rozhranie pre všetky budúce príkazy a zároveň execute() metódu, ktorú s vypýta Receiver commandu aby splnil požadovanú operáciu. Receiver je objekt, ktorý vie ako požadovanú operáciu splniť. Invoker pozná command a pomocou implementovanej execute() metódy dokáže vyvolať požadovanú operáciu. Klient potrebuje implemenotvaž ConcreteCommand a nastavit Receiver pre command. ConcreteCommand definuje spojenie medzi action a receiver. Keď Invoker zavolá execute() metódu na ConcreteCommand spustí tým jednu alebo viac akcií, ktoré budú bežať pomocou Receivera.[19]

Pre lepšie pochopenie je proces zobrazený aj na sekvenčnom diagrame.



Obrázok 13: Sekvenčný diagram Command návrhového vzoru

3.5 Aplikácia

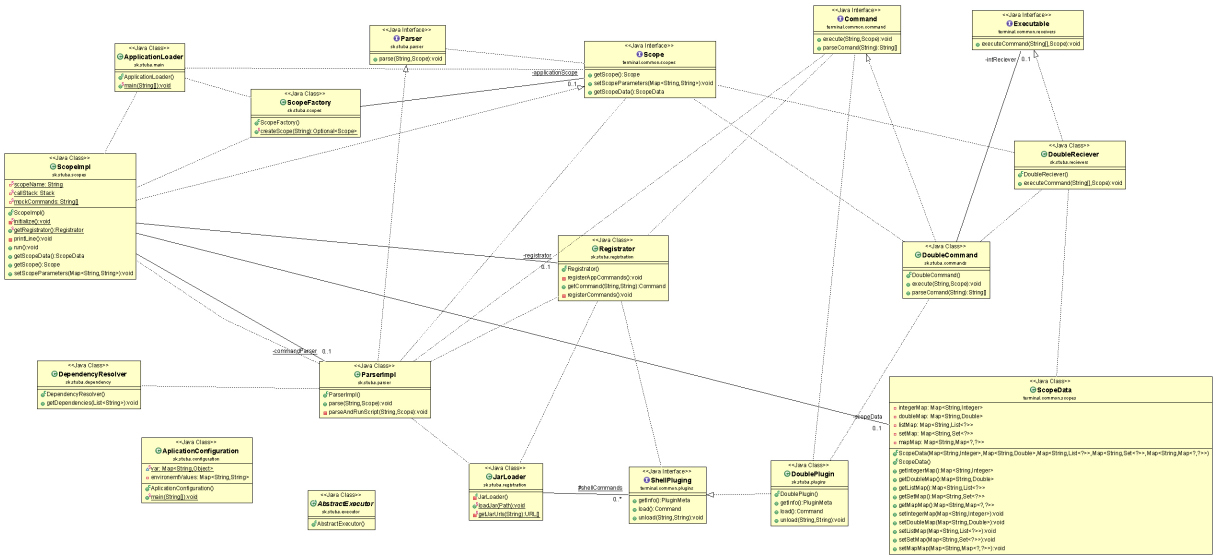
Pre implementáciu vlastného riešenia bolo potrebné na základe prípadov použitia identifikovať časti, z ktorých bude aplikácia pozostávať. Naším zámerom nebolo vytvorenie jednej veľkej aplikácie, ktorá by sa mohla časom stať neudržiavateľnou, ale aplikácia, ktorá umožní používateľom interaktívny aj skriptovací módus. Rozhodli sme sa, že navrhneme jednu hlavnú aplikáciu, ktorá bude mať na starosti interaktívny prístup pre používateľa rovnako aj skriptovací módus, avšak funkcionality, ktoré bude podporovať, zabezpečia menšie externé podprogrami - pluginy, ktoré sa do apikácie nahrajú pri štarte. Pre dosiahnutie požadovaných výsledkov použijeme návrhový vzor Command. Ako ďalšie sme definovali komponenty aplikácie.

3.6 Komponenty aplikácie

Pomocou command dizajnového návrhového vzoru sme prešli na identifikáciu komponentov aplikácie. V prvom návrhu sme identifikovali niekoľko komponentov, ktoré považujeme za podstatné a potrebné pre správny chod programu. Z týchto komponentov sme následne vytvorili malý projekt, kde sme sa pokúsili vytvoriť niekoľko pluginov implementovaných pomocou command dizajnového návrhu na demonštrovanie funkčnosti. Nakoľko bol model funkčný, rozhodli sme sa pokračovať s jeho vývojom. Uvádzame aj komponenty, ktoré sme identifikovali pri vytváraní tejto ukážky funkčnosti :

- Parser - vstupov aj výstupov,
- Loader - na nahrávanie jar súborov,
- Sťahovač závislostí - jar súbory, ktoré momentálne produkt neobsahuje napr. vlastné riešenia,

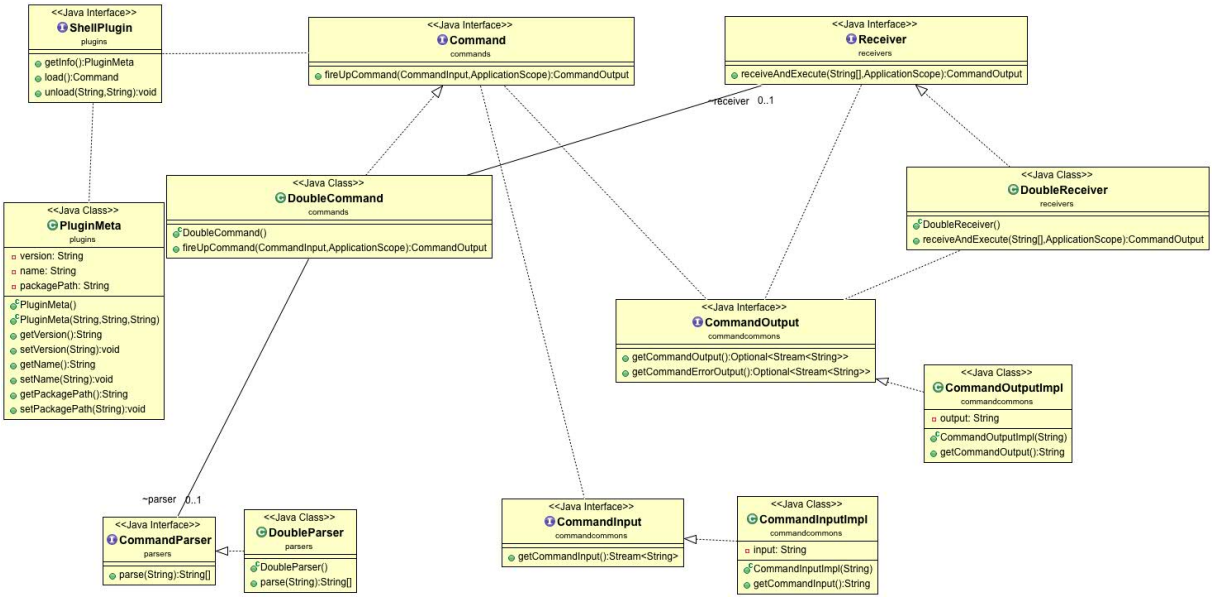
- Scope - je oblasť kde sa definujú premenné, funkcie a pod., tento komponent slúži na vytváranie scopov v rámci aplikácie,
- ScopeData - ktoré majú slúžiť na udržovanie dát v jednotlivých scopoch,
- ShellPlugin - komponent, ktorý nesie implementáciu príkazov.



Obrázok 14: Prvé funkčné riešenie

3.7 Plugin

Z nasledovného diagramu tried nebolo na prvý pohľad zjavné aké komponenty v programe existujú, preto bolo potrebné tieto komponenty rozumne rodeliť. Z prvotného návrhu sme vytiahli plugin, ktorý bude slúžiť na nahrávanie nových funkcionalít do programu. Diagram implementácie rozhraní a konkrétnych tried je viditeľný na nasledovnom obrázku.



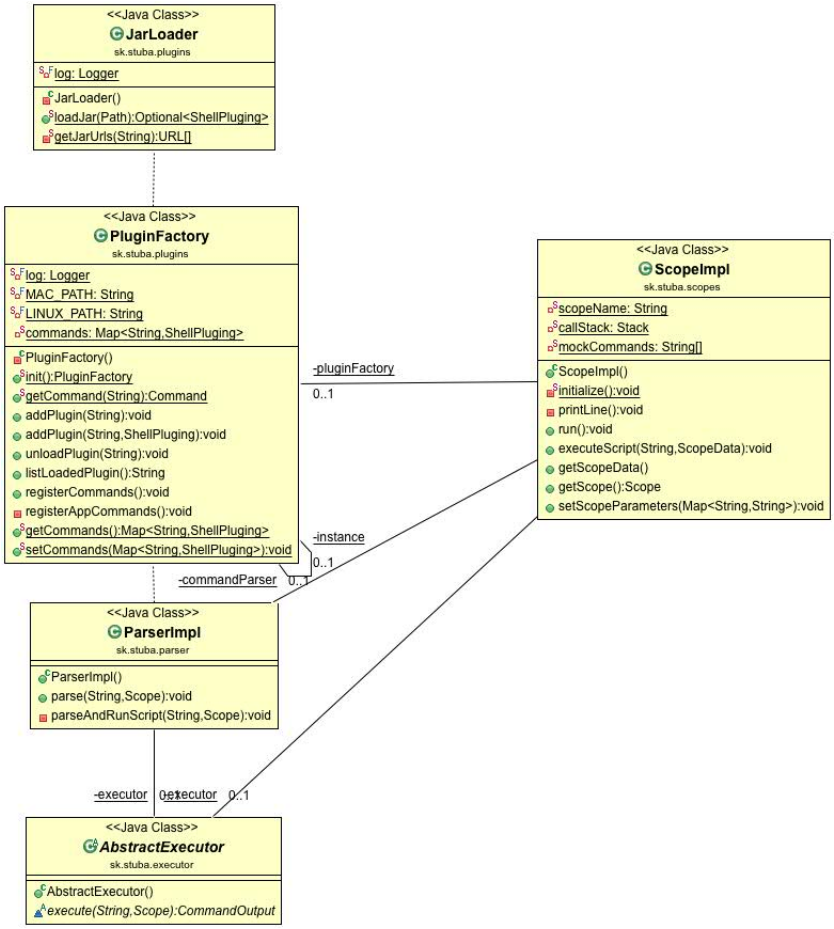
Obrázok 15: Plugin - diagram tried

Ako vidieť z diagramu, Plugin pozostáva z nasledovných častí :

- ShellPlugin - je rozhranie, ktoré slúži na získavanie inštancií commandu, ako aj informáciách o plugine,
- Command - rozhranie, slúžiace na komunikáciu s receivermi,
- Receiver - triedy, ktoré implementujú receiver špecifikujú funkcionalitu pluginu, viď zodpovednosť receiveru v popise command návrhového vzoru,
- CommandParser - obsahuje parser vstupov,
- CommandOutput - rozhranie pre výstupy z commandu,
- CommandInput - rozhranie pre vstupy z commandu,
- PluginMeta - nesie základné informácie o plugine.

Vytvorili sme solídny základ pre vývoj aplikácie, jednoduché rozšérenie funkcionalít, avšak je možné, že bude potrebné aby prišlo k menším zmenám tried a rozhraní, nakoľko nie vždy sa podarí odhadnúť všetky kľúčové časti systímu už počas návrhu.

3.8 Diagram tried aplikácie



Obrázok 16: Diagram tried aplikácie

3.8.1 Stručný popis tried

Obrázok 16 zobrazuje diagram tried aplikácie, ktoré v nasledujúcej časti stručne predstavíme.

3.8.1.1 ScopeImpl Táto trieda je jednou z najhlavnejších celého programu, tvorí základný pilier pre tvorbu akýchkoľvek scopov aplikácie, beh skriptov a pochopiteľne príkazov v interaktívnom móde. Obsahuje informácie o všetkých pluginoch, ktoré sa nahrali pri štarte aplikácie, názov scopu a inštanciu exekútora, pomocou ktorého sa vykonávajú všetky operácie v aplikácii. Implementuje metódy run() pre zabezpečenie interaktívneho módu, kde sa však môžu spúšťať aj skripty. Tiež implementuje metódu executeScript(String function, ScopeData scopeData), ktorá slúži na rekurzívne volanie funkcií v skriptovacom móde a spúšťanie ich príkazov.

3.8.1.2 AbstractExecutor Trieda AbstractExecutor je abstraktná trieda poskytujúca abstraktnú metódu, CommandOutput execute(String command, Scope scope). Vstupom do tejto metódy je príkaz od používateľa a scope v ktorom ho chce vykonať. Definovanie scopu nám slúži na informovanie jednotlivých prijímačov(Reciever), kde sa má daný príkaz vykonať. Ak by scope nebol presne definovaný, aplikácia by sa nevedela rozhodnúť kde sa majú príkazy premietnuť a vyhodila by chybu.

3.8.1.3 ParserImpl RootParser je najpodstatnejšia trieda pre skriptovací mód. Jej úlohou je načítať zdrojové súbory skriptu a následne spustiť exekúciu, ak parsovanie prebehlo v poriadku.

3.8.1.4 PluginFactory PluginFactory, ako z jej názvu vyplýva, je továreň, do ktorej sa pri štarte aplikácie nahrávajú všetky pluginy dostupné z disku. Trieda je vytvorená na základe popisu v kapitole 3.4.1 Návrhove vzory. Na získavanie dostupných pluginov sa používa trieda JarLoader.

3.8.1.5 JarLoader JarLoade je trieda, ktorá pomocou class loadera nahráva nové pluginy do aplikácie.

4 Implementácia

V tejto kapitole sa budeme venovať konkrétnemu riešeniu problematiky jednotnej platformy určenej na vytváranie skrípt. Kapitola postupne definuje jednotlivé kroky implementácie riešenia. Každý krok je popísaný jednou z nasledovných možností alebo ich kombináciou: pomocou diagramu aktivít, diagramu tried, pseudokódom, prípadne útržkami zdrojového kódu z fungujúcej aplikácie. Jednotlivé sekcie práce sme rozdelili na základe toho, akými smermi sa môže aplikácia uberať. Rozhrania, ktoré aplikácia používa, sú uvedené v prílohe C.

4.1 Hlavná trieda aplikácie - AbstractScope

Táto trieda je jednou z najhlavnejších celého programu, tvorí základný pilier pre tvorbu akýchkoľvek scopov aplikácie, beh skriptov, ako aj príkazov v interaktívnom móde. Obsahuje informácie o všetkých pluginoch, ktoré sa nahrali pri štarte aplikácie, názov scopu a inštanciu exekútora, pomocou ktorého sa vykonávajú všetky operácie v aplikácii. Trieda má tiež informácie o vstupných parametroch, výstupných hodnotách a funkciách, ktoré sú v danom scope definované. Spomenuté hodnoty sa využívajú pri vytváraní a behu skriptov pre vstupné parametre funkcií, ako aj ich návratová hodnota. Implementuje rozhrania Scope, Runnable. Implementuje metódy run() pre zabezpečenie interaktívneho módu, kde sa však môžu spúšťať aj skripty. Tiež implementuje metódu executeScript(String function, ScopeVariables scopeVariables), ktorá slúži na rekurzívne volanie funkcií v skriptovacom móde a spúšťanie ich príkazov. Implementuje funkciu runInputCommands(), pomocou ktorej je aplikácia spustená ak má na vstupe akékoľvek vstupné parametre. Túto funkcionalitu sme pridali z dôvodu integrácie s inými aplikáciami.

4.1.1 run

Táto metóda sa volá iba raz a to po štarte aplikácie. V cykle od používateľa získava vstupy, ktoré sa následne pokúša vykonať. Ak aplikácia nedokáže príkaz vykonať, vypíše chybu do konzoly.

4.1.2 executeScript

Je funkcia, pomocou ktorej sa spúšta beh jednotlivých funkcií skriptu. Je definovaná všeobecne pre akýkoľvek scope, samozrejme v prípade, ak novému scopu rozsah metódy nepostačuje je potrebné ju prepísať v triede, ktorá je odvodená od AbstractScope. Základný algoritmus pre beh skriptov je opísaný v nižšie priloženom pseudokóde.

Algoritmus 4 Pseudokód všeobecnej implementácie spúšťania funkcií

```
executeScript(String function, ScopeVariables scopeVariables) {
executeGlobalCommands();
fName = getFunctionName();
Scope parent =  this.getParent();

Scope functionScope = parent.functions.get(fName);
setFunctionscopeVariables(scopeVariables.clone());

for (String command : functionScope.stack) {
if (command.startsWith("$(")) {
out = executor.execute(command, functionScope);
if (command.contains("returnValue = ")) {
setReturnValue();
}else{
writeOutputs(out);
}
} else if (command.startsWith("fcall ")) {
if(functionHasReturnValue(command)){
setCallParameters();
executeScript(function, getVariables());
getReturnValue();
}else{
executeScript(function, getVariables());
}
}
}
```

4.1.3 ForScope

For cyklus má oproti hlavnému scopu niekoľko rozdielov. Ako prvé si môžeme všimúť, že For scope kopíruje referenciu premenných scopu, naopak funkcie vytvárali hlbokú kópiu, aby sa zmeny v rôznych funkciách nepremietli do pôvodných. Zmeny sa vo funkciách dajú aplikovať iba pomocou vracania hodnôt z funkcií. Ďalšia vec navyše je cyklus, v ktorom je vnorený rovnaký For cyklus ako v AbstractScope avšak obohatený o funkciu break, ako je možné vidieť v nasledujúcom kóde :

Algoritmus 5 Kód implementácie spúšťania funkcie For

```
setScopeVariables(scopeVariables);
for (int i = getStart(); evalueateCondition(); i = evalueateIncremet()) {
// for cyklus z abstract scopu obohatený o break
}
```

Cyklus je pridaný preto, aby sme dokázali vyhodnotiť vstupy zadané v skripte. Ako prvé sa do premennej i nainicializuje výsledok exekúcie príkazu zo skriptu. Následne sa rovnakým spôsobom vyhodnotí podmienka, a ak je všetko v poriadku, For cyklus sa začne vykonávať. Ďalej sa rovnako ako v predchádzajúcom prípade zopakuje cez celý stack For cyklu. Momentálnou limitáciou tohto prístupu je možnosť inicializovať a iterovať iba cez premenné numerického typu. Avšak v budúcnosti je tento koncept ľahko rozšíriteľný aj pre premenné iných typov. For cyklus obsahuje aj príkaz break, ktorý je bežnou súčasťou tejto štruktúry.

4.1.4 If Scope

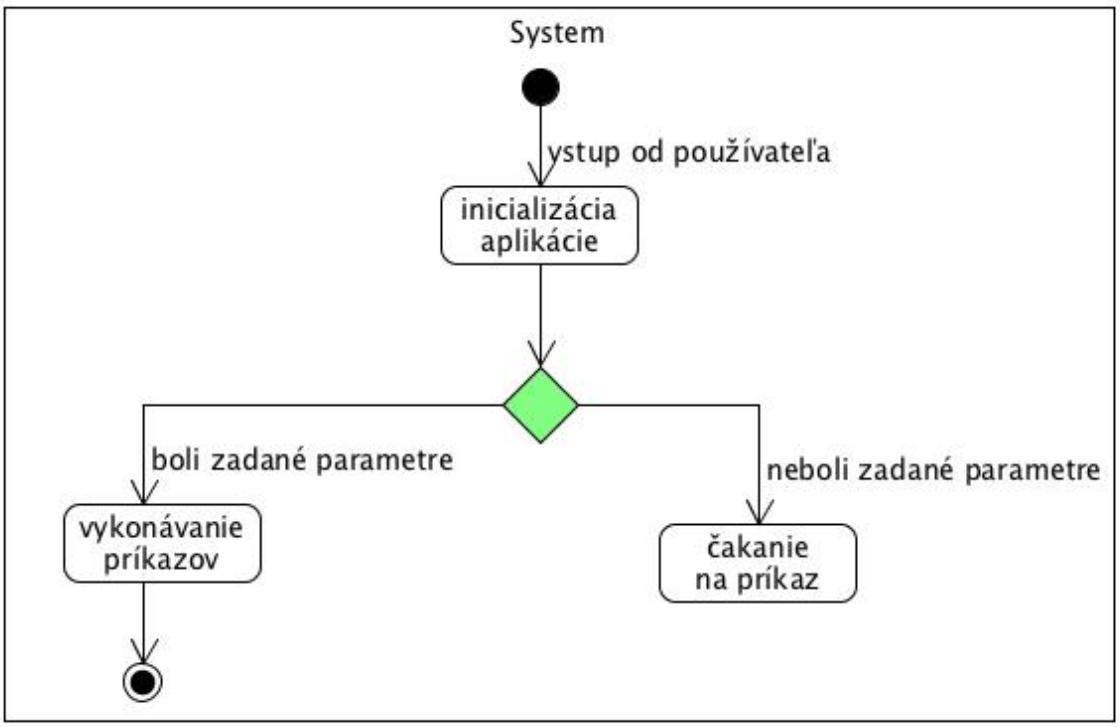
If scope je jednoduchšou implementáciou scopu ako bol For, nakoľko neobsahuje ďalšie cykly iba podmienku, ktorá ak sa vyhodnotí ako pravdivá, vykoná sa aj stack, ktorý if scope obsahuje. Na evaluáciu podmienok, priraďovanie hodnôt premenným ako aj iteráciu boli použité nami vopred naimplementované pluginy, ktoré dokážu pracovať s primitívnymi premennými Javy.

Algoritmus 6 Kód implementácie spúšťania funkcie For

```
if (evalueateCondition()) {
// For cyklus z abstract scopu obohatený o break
}
```

4.2 Štart aplikácie

Na nasledujúcom obrázku je zobrazená aktivita spustenia konzoly. Ako vidieť z diagramu, aplikácia sa najprv nainicializuje, čo vysvetlíme v sekcii 4.3. Po inicializácii premenných sa vytvorí inštancia triedy ScopeImpl, ktorá je implementáciou AbstractScope triedy, charakterizovaná v predchádzajúcej sekcii práce. V ďalšom kroku aplikácia overí, či má na vstupe parametre. V prípade, ak áno vykoná ich, inak spustí konzolu, ktorá čaká na vstup od používateľa. V tejto časti popíšeme čo sa deje v prvom prípade, a teda že máme zadané parametre.



Obrázok 17: Activita spustenia aplikácie

4.2.1 Štart s parametrami

Možnosť s parametrami na vstupe sme implementovali z dôvodu aby sa aplikácia dala integrovať s inými enterprise aplikáciami, ktoré môžu buď čítať výstupy priamo z konzoly, alebo ich môžu presmerovať do súboru. Novovytvorený súbor je následne možné načítať a vykonať nad ním potrebné operácie.

4.3 Inicializácia aplikácie

Inicializácia pozostáva z dvoch krokov:

- Nahranie dostupných pluginov z disku,
- Získanie inštancie exekútora - ExecutorImpl.

Obrázok 18: Activita inicializácie aplikácie

4.3.1 Načítanie pluginov

Načítanie pluginov sa uskutočňuje pomocou triedy PluginFactory a JarLoader, kde tento proces pozostáva z troch krokov.

4.3.1.1 Načítanie dostupých pluginov z disku Trieda PluginFactory zavolá metódu registerAllPlugins(), ktorá najprv získa cesty k uloženým jar súborom. Jar súbory musia byť uložené na presne špecifikovanom mieste. Následne tieto cesty odovzdá triede JarLoader, ktorá sa pokúsi na daných cestách načítať jar súbory, nájsť v nich triedu implementujúcu rozhranie ShellPlugin. Ak takúto triedu nájde, pokúsi sa z nej vytvoriť inštanciu pomocou reflexie, a takto vytvorenú inštanciu následne vráti triede PluginFactory, ktorá si túto triedu uloží na základe informácií, ktoré každý ShellPlugin poskytuje. PluginFactory ukladá inštancie do mapy, kde kľúč je trieda PluginMeta získaná z pluginu, a hodnotou je práve načítaný plugin.

Algoritmus 7 Ukážka načítania balíčka z disku počítača

```
try {
URLClassLoader cl = URLClassLoader.newInstance(urls);
JarInputStream jarFile = new JarInputStream(new FileInputStream(jarPath
.toFile()));
while (true) {
jarEntry = jarFile.getNextJarEntry();
if (jarEntry == null) {
break;
}
if (jarEntry.getName().endsWith(".class")) {
String className = jarEntry.getName()
.substring(0, jarEntry.getName().length() - 6);
className = className.replace('/', '.');
Class c = cl.loadClass(className);
Object obj = c.newInstance();
if (obj instanceof ShellPlugin) {
return Optional.of((ShellPlugin) obj);
}
}
}
```

4.3.1.2 Načítanie dostupných pluginov z aplikácie

Pluginy môžu byť definované priamo v aplikácii. V takom prípade, priamo do rovnakej mapy ako v predchádzajúcom prípade vkladáme inštancie pluginov a informácie o nich.

4.3.1.3 Finálne načítanie pluginov

Používateľ môže počas využívania aplikácie stiahnuť alebo naprogramovať viaceré verzie rovnakého pluginu. Z tohto dôvodu sme navrhli krok, ktorý má za úlohu načítať všetky balíčky s jedinečným menom a najvyššou možnou verziou pre daný balíček. Následne načítanie končí a vráti sa inštancia tejto továrne pre pluginy.

4.4 Vykonávač príkazov

Všetky príkazy, ktoré sa majú pomocou aplikácie vykonať sa spúšťajú pomocou triedy ExecutorImpl, ktorá implementuje metódu rozhrania Executor CommandIO execute(String command, Scope scope). Rozhodovacia logika, ako sa príkazy od používateľa

budú vykonávať, je implementovaná práve v tejto metóde. Pre zjednodušenie opisu si pomôžeme pseudokódom rozhodovacieho algoritmu. Zo pseudokódu je zrejmé, že exekútor najskôr zistí, či príkaz neobsahuje znaky pajpy, presmerovania štandardného vstupu/výstupu. Následne identifikuje čím sa príkaz začína a vyberie vhodnú metódu pre zadaný príkaz. Po vykonaní príkazu zisťuje či sa má výstup z programu zapísať mimo štandardného vstupu alebo výstupu. Ak áno, program očakáva platnú cestu za oddeľovačmi stdout> resp. stderr>, kde sa pokúsi zapísať svoj výstup do súboru. Diagram tried je uvedený v prílohe D.

Algoritmus 8 Ukážka pseudokódu exekútora.

```
isPipe = isPipe(command);
writeStd = shouldWriteStd(command);
writeErr = shouldWriteErr(command);
if (command.startsWith("sh ")
|| command.startsWith("win ")
|| command.startsWith("ext ")){
  out = executeNativeCommand(command);
}else if (command.startsWith("./"){
  out = loadAndExecuteScript(command, scope);
}else if (command.startsWith("${")) {
  out = getVariable(command, scope);
}else if (command.startsWith("$(")){
if(isPipe){
  executeCommandAsPipe(command, scope);
}else{
  executeCommand(command, scope);
}
}
!writeStd ? saveToFile(out.getStdOut()) : out;
!writeErr ? saveToFile(out.getErrOut()) : out;
```

4.4.1 Vykonávač príkazu

Vykonávač príkazu si v prvom kroku vyžiada od PluginFactory plugin, ktorý by dokázal daný príkaz vykonať. Ak factory úspešne vráti plugin, exekútor si z pluginu zoberie command a zavolá na ňom metódu execute(CommandIO commandInput, Optional<Scope> scope). Command následne spracuje vstupy v metóde parseCommand(Stream<String> var). Metóda execute potom vyberie receiver, ktorý daný príkaz dokáže vykonať. Receiver následne vykoná príkaz a vráti rozhranie CommandIO, ktoré nesie informáciu o úspešnosti behu ako aj výstupy.

4.4.2 Vykonávač pajpy

Ak aplikácia rozozná, že v príkaze ide o pajpu, príkaz putuje do triedy PipeExecutor, kde sa rozparsuje na parciálne príkazy. Po rozparsovaní príkazov sa každému príkazu nastaví nasledujúci príkaz, ktorý sa má vykonať. Posledný príkaz neobsahuje žiadny ďalší. Po takomto prednastavení exekútora sa spustí vykonávanie prvého príkazu, jeho výstup je pri úspešnom ukončení presmerovaný ďalšiemu príkazu. Tento kolobeh pokračuje až kým sa nedostane k poslednému príkazu. Ten následne vracia svoj výstup exekútorovi z 4.1.

4.4.3 Vykonávač skriptu

Je popísaný v časti 4.1.2 tejto práce. Pre detailné pochopenie tejto metódy je potrebné prezrieť si priložené zdojové kódy aplikácie.

4.5 Parser skriptov

Parser berie ako parameter cestu k skriptu, ktorý je potrebné sparsovať a vykonať. Parser skriptov pozostáva z dvoch častí:

- Predspracovania
- Parsovanie predspracovaného vstupu

4.5.1 Predspracovanie

Úlohou predspracovania je načítať zdrojový kód skriptu, vyčistiť ho od komentárov, ako aj vyfiltrovať metódy, zistiť či skript obsahuje funkciu main, prípadne zmeniť používané pluginy. Predspracovanie pozostáva z viacerých krokov, ako prvé sa parser pokúsi načítať súbor po riadkoch ako stream zo Stringov. Ďalej zo súboru vyfiltruje všetky prázdne riadky, riadky obsahujúce komentáre, inak povedané začínajúce znakom mriežky. Z už vyfiltrovaných riadkov následne vyfiltrujeme iba tie, ktoré sa začínajú slovom use. Use slúži ako kľúčové slovo pre použitie pluginov v skriptoch. Pomocou týchto riadkov sa pokúsi vyžiadať od PluginFactory pluginy definované v use. Ak sa ich nepodarí nájsť aplikácia použije príkazy, ktoré má aktuálne k dispozícii, ale je pravdepodobné, že v tomto

prípade dôjde ku chybe počas behu skriptu. Tieto riadky sa následne vymažú. Ďalší proces je filtrovanie riadkov, ktoré začínajú na kľúčové slovo include. Tieto riadky špecifikujú, ktoré ďalšie súbory sú použité v skripte. Rovnako ako pri use, riadky načítame do listu a zmažeme ich z hlavného obsahu. Ďalej načítame všetky súbory definované v include a pripojíme ich k tým, ktoré sme mali zo zdrojového skriptu. Načítavanie pomocou include sa deje cyklicky, avšak pamätáme si, ktoré súbory sme už načítali aby sme zabránili zacykleniu aplikácie a jej následnému pádu. Na záver si ešte vyfiltrujeme riadky začínajúce na function a zistíme unikátne mená definovaných funkcií. V prípade, že aplikácia nenájde funkciu main vyhodí výnimku a beh skriptu sa ukončí. Samozrejme používateľ je na túto skutočnosť upozornený. Po predspracovaní nasleduje parsovanie.

4.5.2 Parsovanie

Aby sme si parsovanie skriptu vedeli jednoducho predstaviť, povedzme, že máme jeden for cyklus, ktorý prechádza cez všetky načítané riadky zo súborov. V tomto cykle následne definujeme rozhodovací mechanizmus, ktorý sa na základe prečítaného vstupu rozhodne čo má vykonať. Parser má za úlohu vytvárať scopy a pre vytvorené scopy získať podstatné informácie. Parser očakáva, že ak máme akékoľvek globálne príkazy v našom skripte, budú definované na začiatku skriptu. V momente ako parser narazí na prvú funkciu nenačítava príkazy do globálneho alebo skriptového scopu, ale do scopu konkrétnej funkcie. Parser tiež kontroluje či daná funkcia nemá pred sebou anotáciu @Override, v tomto prípade nastaví pre takúto funkciu príznak, že je prepísaná a mala by sa zobrať funkcia označená s @Override a nie pôvodná. Parser povoľuje aj viacnásobné prepísanie funkcie, do behu sa zoberie vždy tá, ktorá bola načítaná ako posledná.

4.5.3 Parsovanie funkcií

V momente, keď parser narazí na funkciu odovzdá riadenie ďalšej funkcii zodpovednej za správne načítanie funkcie. Parsovanie funkcie je zložitejší proces nakoľko môže dôjsť k rôznym situáciám. Na začiatku sa načíta prvý riadok, z neho sa vyparsuje návratová hodnota funkcie, meno funkcie a tiež parametre funkcie, kontroluje sa aj otváracia zátvorka pre scope, ktorá musí byť na riadku kde bola definovaná funkcia. Ďalšie riadky funkcie sa načítavajú v cykle a kontroluje sa nasledovné:

- kontroluje sa správne ukončenie riadkov pomocou bodkočiarky.
- v prípade, že bol riadok ukončený bodkočiarkou, zisťuje sa či je na riadku definované volanie funkcie. Ak áno tento riadok je označený ako fcall s príslušným volaním funkcie a jej parametrami. V prípade, že nejde o volanie funkcie riadok sa označí ako volanie príkazu a to pomocou $().

- v prípade, že je riadok ukončený s
  parser pripojí k riadku nasledujúce riadky, pokým nenarazí na bodkočiarku.
- v prípade, že narazí na if odovzdá kontrolu parseru pre If scope, no predtým zapíše do aktuálneho scopu fcall s unikátnym identifikátorom podmienky.
- v prípade, že narazí na for odovzdá kontrolu parseru pre For scope, ale predtým zapíše do aktuálneho scopu fcall s unikátnym identifikátorom cyklu.
- v prípade, že narazí na return vytvorí príkaz, ktorý bude poskladaný z typ_vracanej hodnoty returnType = hodnota za returnom;.
- v prípade, že riadok obsahuje ukončovaciu kučeravú zátvorku, ukončí načítanie.

4.5.4 Parsovanie for cyklov

Ak hlavný parser odovzdá kontrolu parseru pre for cyklus, spustí sa preProcess funkcia v tomto scope, kde sa rozdielne načítava iba prvý riadok. Z prvého riadku sa uložia tri parametre pre úspešný beh cyklu - počiatočná hodnota, podmienka, ktorá sa bude kontrolovať a inkrement. Pre vykonanie tejto úlohy je v scope vytvorená nová premenná, s unikátnym menom aby sa predišlo prepisovaniu premenných.

4.5.5 Parsovanie podmienok

Ak hlavný parser odovzdá kontrolu parseru pre If podmienku, spustí sa preProcess funkcia v tomto scope, kde sa rozdielne načítava iba prvý riadok. Z prvého riadku sa uloží iba jeden parameter a to podmienka, ktorá sa bude vždy vyhodnocovať.

4.5.6 Výsledok parsovania

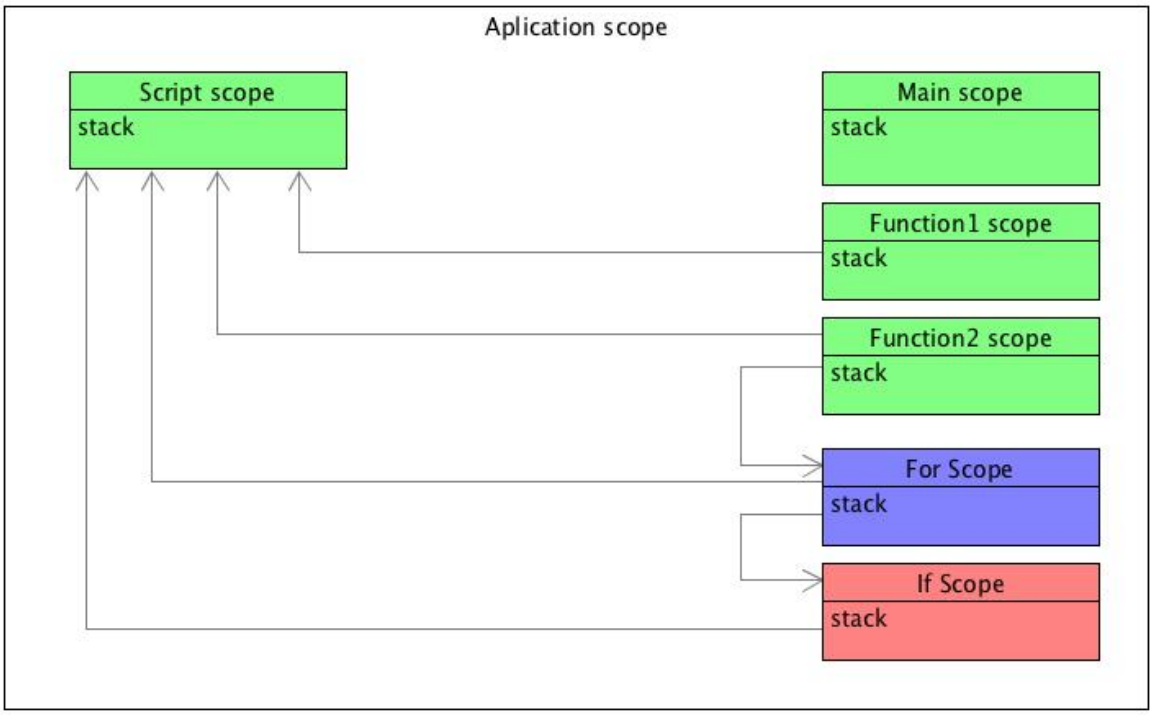
Výsledkom parsovania je štruktúra zobrazená na obrázku nižšie. V aplikačnom scope sa vytvorí scope skriptový, ktorý bude rodičom pre všetky ostatné scopy, ktoré sa vytvoria. Každý scope obsahuje stack príkazov, ktoré má vykonať počas behu. Tento stack môže obsahovať aj funkčné volania, ako je znázornené na obrázku. Po úspešnom ukončení parsovania sa na script scope zavolá funkcia main, ktorá spustí exekúciu skriptu.

4.6 Implementované pluginy

V tejto sekcii si stručne predstavíme naimplementované pluginy.

4.6.1 Variable plugin

Premenné sú nevyhnutné pre každý programovací jazyk, pričom sme sa rozhodli, že budeme mať premenné, pre ktoré vieme určiť typ. Pre prvotnú implementáciu bola naším cieľom realizácia premennej typu int, double, boolean a string. V budúcnosti by sme radi



Obrázok 19: Diagram tried rozhraní aplikácie

implementáciu rozšírili minimálne o list, mapu a set. Variable plugin je jeden z pluginov, ktorý sme neexternalizovali, pričom by to nepredstavovalo zásadný problém. Plugin bol implementovaný pomocou command návrhového vzoru tak, ako všetky ostatné, avšak tento ako jeden z mála obsahuje zložitejší parser vstupov, na základe čoho rozhodne čo sa má vykonať.

Parser pluginu dokáže na základe vstupu rozpoznať a spustiť vykonávanie príkazu na nasledujúcich receiveroch : int, double, boolean, string, change, get, arithmetic, stringPrep, booleanPrep.

Prvé štyri receivre inicializujú premennú, ak nieje definovaná jej hodnota, int sa inicializuje na 0, double na 0D, boolean na false a string na prázdny reťazec.

Receiver change je schopný zmeniť hodnotu akejkoľvek premennej v scope. Rovnako get receiver je schopný vytiahnuť hodnotu z ktorejkoľvek premennej scopu.

Arithmetic je receiver, ktorým sme zabezpečili jednoduché zmeny numerických hodnôt. Podporované operácie sú +, -, *, /, %. ArithmeticPrep receiver ako prvé rozdelí premenné, príkazy a hodnoty do poľa, v ktorom medzi ne vkladá znamienka s operáciou ako ju vyparsoval zo vstupného stringu. V prvom kroku vyhodnotí všetky príkazy, premenné a následne začne vyhodnocovať výraz z ľava do prava. Momenálna implementácia

neumožňuje stanoviť prioritu výpočtu.

StringPrep je receiver, ktorý zo vstupných parametrov vyskladá výsledný string, môžeme mu spájať premenné akéhokoľvek typu ako aj vkladať nové reťazce, ktoré ale musia byť označené úvodzovkami. Operácia pre spájanie stringov je + podobne ako v Jave.

Posledným z implementovaných receiverov je booleanPrep, ktorý funguje v dvoch krokoch. V prvom kroku zadaný vstup rozseká na operácie operátorov && a ||. Následne vyhodnotí premenné, príkazy medzi tymito operátormi a ich pravdivostné hodnoty vráti spať logickým operátorom, ktoré sa vykonávajú z ľava do prava. Nie je možné určovať prioritu vykonávania, teda aspoň v tejto verzii receivera.

4.6.2 Package plugin

Package plugin je, rovnako ako Variable, príkladom kde na základe výstupov z command parsera aplikácia dokáže určiť, ktorý receiver použiť. Package command pozná tri receivery : download, change a delete. Download dokáže stiahnuť nový jar súbor zo servera na základe vstupu od používateľa. Na jeho následné načítanie je potrebný reštart aplikácie. Change sa stará o manažovanie existujúcich pluginov. Ak je vstup zadaný v tvare pkg change <názov balíčka> <verzia> a balíček sa na PC nachádza, spolu s PluginFactory triedou zabezpečia, aby sa počas ďalšieho behu aplikácie používal špecifikovaný balíček. Delete receiver slúži na dočasné vypnutie špecifikovaného pluginu, nemá za úlohu fyzicky zmazať jar z disku PC.

4.6.3 Grep plugin

Plugin je momentálne implementovaný nasledovne : Command očakáva príkaz v tvare grep <filter> <riadky reťazcov>, jeho parser tieto riadky pospája do stringu a pošle Grep receiveru na filtráciu. Aktuálne sa príkaz spolieha, že na vstupe dostane riadky reťazcov, avšak v budúcnosti by bolo dobré aby tento receiver dokázal rozoznať aj príkazy, premenné, prípane cestu k súboru.

4.6.4 Echo plugin

Tento plugin slúži na výpis do konzoly. Jeho syntax je echo (<vstupy>). Pod vstupmi môžeme rozumieť premenné, príkazy alebo textové reťazce, ktoré sú označené úvodzovkami.

4.6.5 Change directory plugin

Tento plugin slúži na jednoduché presúvanie sa v adresárovom strome PC. Presúvanie sa je pre používateľov Windowsu aj Linuxu intuitívne, nakoľko je implementované úplne rovnako ako na spomenutých OS.

4.6.6 Pwd plugin

Tento plugin sa používa na vypísanie cesty aktuálneho adresára, v ktorom sa nachádzame.

4.6.7 List directory plugin

Tento plugin slúži na vypísanie obsahu adresára, v ktorom sa nachádzame, ale ak ako parameter vložíme platnú cestu vypíše obsah špecifikovaného súboru. V prípade, že plugin nedostane inštanciu scopu, ktorý obsahuje informáciu o aktuálnom adresári, je naprogramovaný aby vypísal obsah domovského adresára používateľa.

4.6.8 Copy plugin

Tento plugin slúži na jednoduché kopírovanie súborov z jedného miesta na druhé na disku PC. Momentálna implementácia vyžaduje aby bol príkaz zadaný v tvare copy <zdrojová adresa> <cieľová adresa>. Neumožňuje kopírovanie viacerých súborov alebo adresárov naraz.

4.6.9 Move plugin

Tento plugin slúži na jednoduché presúvanie súborov z jedného miesta na druhé na disku PC. Momentálna implementácia vyžaduje aby bol príkaz zadaný v tvare copy <zdrojová adresa> <cieľová adresa>. Neumožňuje presúvanie viacerých súborov alebo adresárov naraz.

4.6.10 Get processes plugin

Tento plugin je naimplementovaný tak, aby po zadaní príkazu ps dokázal vypísať všetky bežiace procesy na PC.

4.6.11 Get system information plugin

Tento plugin je naimplementovaný tak, aby po zadaní príkazu getinfo dokázal vypísať čo možno najviac informácií o systéme, na ktorom beží. Informácie sa môžu na rôznych počítačoch odlišovať.

5 Zhodnotenie výsledkov

Počas testovania aplikácie sme nenarazili na žiadne závažné problémy, pokiaľ sme dodržiavali zásady vymedzené v práci.

Aplikácia je schopná fungovať v troch rôznych režimoch, a to interaktívnom, skriptovacom a tiež v móde spúštania príkazov a skriptov priamo z konzoly pomocou parametrov. Taktiež je plne funkčná pri práci s premennými, skriptami, vykonávaním funkcií, cyklov a nami definovaných podmienok.

Podpora základných dátových typov je bezproblémová. Aritmetické operácie, ktoré je nad nimi možné vykonávať sú mierne okresané. Je možné definovať operácie nad numerickými dátovými typmi ako +, -, *, /, %, no nie je možné definovať ich prioritu. Rovnako to platí aj pre logické operátory ako &&, ||. Dobrou správou však je, že aj tieto funkcionality boli implementované pomocou pluginov a tým pádom je možné ich kedykoľvek nahradiť sofistikovanejšími riešeniami.

Tým, že je aplikácia postavená na Jave, je plne funkčná na každom PC, ktorý má OS podporovaný touto platformou. Ako je možné vidieť z príručky, na spustenie aplikácie je potrebné iba nainštalovať Java JRE 8 , nastaviť Path aby smerovala na nainštalovanú Javu a spustiť predpripravený skript. Pre úplnosť sme pripravili nasledovné grafy, ktoré zobrazujú efektivitu aritmetických operácií.

Záver

Cieľom práce bolo zanalyzovať populárne konzolové rozhrania rovnako aj skriptovacie jazyky, ktoré sú často využívané pri administrácii počítačových systémov. Taktiež bolo treba nájsť jednotlivé výhody ako aj nedostatky jednotlivých rišení, zhodnotiť ich a nájsť medzi nimi rozumný prienik, ktorý bolo treba dostať do použiteľnej podoby. Kládli sme dôraz hlavne na to aby naše riešenie bolo čím najlepšie upraviteľné aby mohlo vyhovieť požiadavkam rôznych používateľov.

Zoznam použitej literatúry

1. ZARRELLI, Giorgio. *Mastering Bash*. 1. vyd. Birmingham : Packt Publishing, 2017, 2004. ISBN: 9781784396879.
2. CIACCIO, Robert S. *PowerShell vs. the Unix Shell*. 18-12-2010. Dostupné tiež z: `https://superuser.com/questions/223300/powershell-vs-the-unix-shell`.
3. ABRAHAM SILBERSCHATZ Peter B. Galvin, Greg Gagne. *Operating System Concepts - Ninth Edition*. 9. vyd. Wiley, 2012, 2012. ISBN: 978-1118063330.
4. HAAPANEN, Tom. *What is the history of Microsoft Windows?* 18-01-2018. Dostupné tiež z: `https://kb.iu.edu/d/abwa`.
5. MICROSOFT. *Windows and Windows Server Automation with Windows PowerShell*. 2018. Dostupné tiež z: `https://technet.microsoft.com/en-us/library/mt156946.aspx`.
6. STATCOUNTER. *Desktop Operating System Market Share Worldwide | StatCounter Global Stats*. 13-04-2018. Dostupné tiež z: `http://gs.statcounter.com/os-market-share/desktop/worldwide`.
7. STATCOUNTER. *Operating Systems market share*. 13-04-2018. Dostupné tiež z: `https://netmarketshare.com/operating-system-market-share.aspx?options=%7B%22filter%22%3A%7B%22%24and%22%3A%5B%7B%22deviceType%22%3A%7B%22%24in%22%3A%5B%22Desktop%2Flaptop%22%5D%7D%7D%5D%7D%2C%22dateLabel%22%3A%22Trend%22%2C%22attributes%22%3A%22share%22%2C%22group%22%3A%22platform%22%2C%22sort%22%3A%7B%22share%22%3A-1%7D%2C%22id%22%3A%22platformsDesktop%22%2C%22dateInterval%22%3A%22Monthly%22%2C%22dateStart%22%3A%222017-05%22%2C%22dateEnd%22%3A%222018-04%22%2C%22segments%22%3A%22-1000%22%2C%22plotKeys%22%3A%5B%7B%22platform%22%3A%22Windows%22%7D%2C%7B%22platform%22%3A%22Mac%20OS%22%7D%2C%7B%22platform%22%3A%22Linux%22%7D%2C%7B%22platform%22%3A%22Chrome%20OS%22%7D%5D%7D`.
8. W3TECHS. *Unix vs. Linux vs. Windows vs. macOS usage statistics, May 2018*. 13-04-2018. Dostupné tiež z: `https://w3techs.com/technologies/comparison/os-linux,os-windows,os-macos,os-unix`.
9. KOLUGURI, Naveen. *If / Else Statements (Shell Scripting) - Code Wiki*. 11-11-2017. Dostupné tiež z: `http://codewiki.wikidot.com/shell-script:if-else`.

10. BRENTON J.W. BLAWAT, Chris Dent. *Mastering Windows PowerShell Scripting - Second Edition*. 2. vyd. Birmingham : Packt Publishing, 2017, 2004. ISBN: 9781787126305.

11. PAYNE, James. *Beginning Python®: Using Python 2.6 and Python 3.1*. 1. vyd. Wrox, 2010, 2010. ISBN: 9780470414637.

12. NICHOL, Alex. *unixpickle/Benchmarks: Some language performance comparisons.* 12-04-2017. Dostupné tiež z: `https://github.com/unixpickle/Benchmarks`.

13. CONEMU. *ConEmu - Handy Windows Terminal*. 03-01-2018. Dostupné tiež z: `https://conemu.github.io/`.

14. VASKO, Samuel. *Cmder | Console Emulator*. 03-01-2018. Dostupné tiež z: `http://cmder.net/`.

15. TOMEK BUJOK, Lukasz Pielak. *Babun - a windows shell you will love!* 2015. Dostupné tiež z: `http://babun.github.io/`.

16. MOBATEK. *MobaXterm Xserver with SSH, telnet, RDP, VNC and X11 - Features*. 03-01-2018. Dostupné tiež z: `https://mobaxterm.mobatek.net/features.html`.

17. ĽUDOVÍT MOLNÁR Milan Češka, Bořivoj Melichar. *Gramatiky a jazyky*. 1. vyd. Bratislava : Alfa, 1987, 2004. MDT: 519.682(075.8).

18. ORACLE. *What is technology and why do I need it?* 01-2018. Dostupné tiež z: `https://www.java.com/en/download/faq/whatis_java.xml`.

19. WILLIAM C. WAKE, Steven John Metsker. *Design Patterns in Java™*. 2. vyd. Addison-Wesley Professional, 04-2006, 2006. ISBN: 9780321630483.

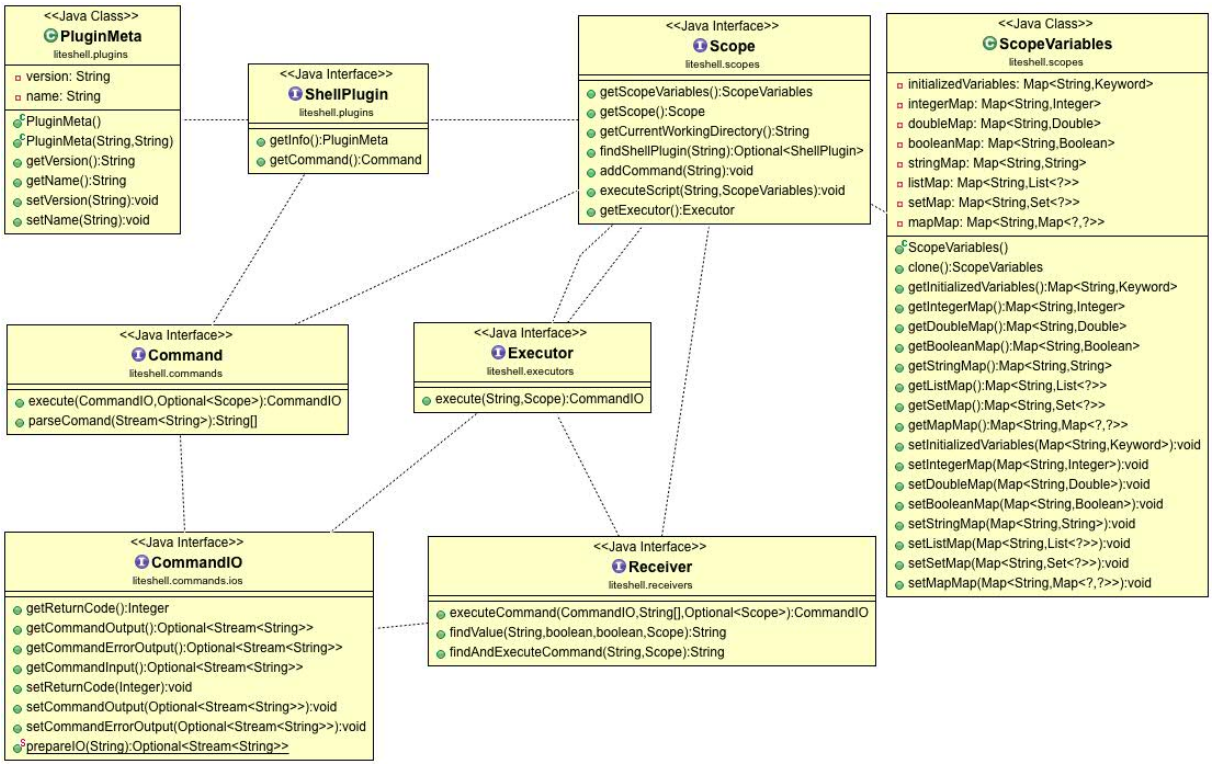
Prílohy

A CD s aplikáciou a prácou

B Návod na spustenie a používanie aplikácie

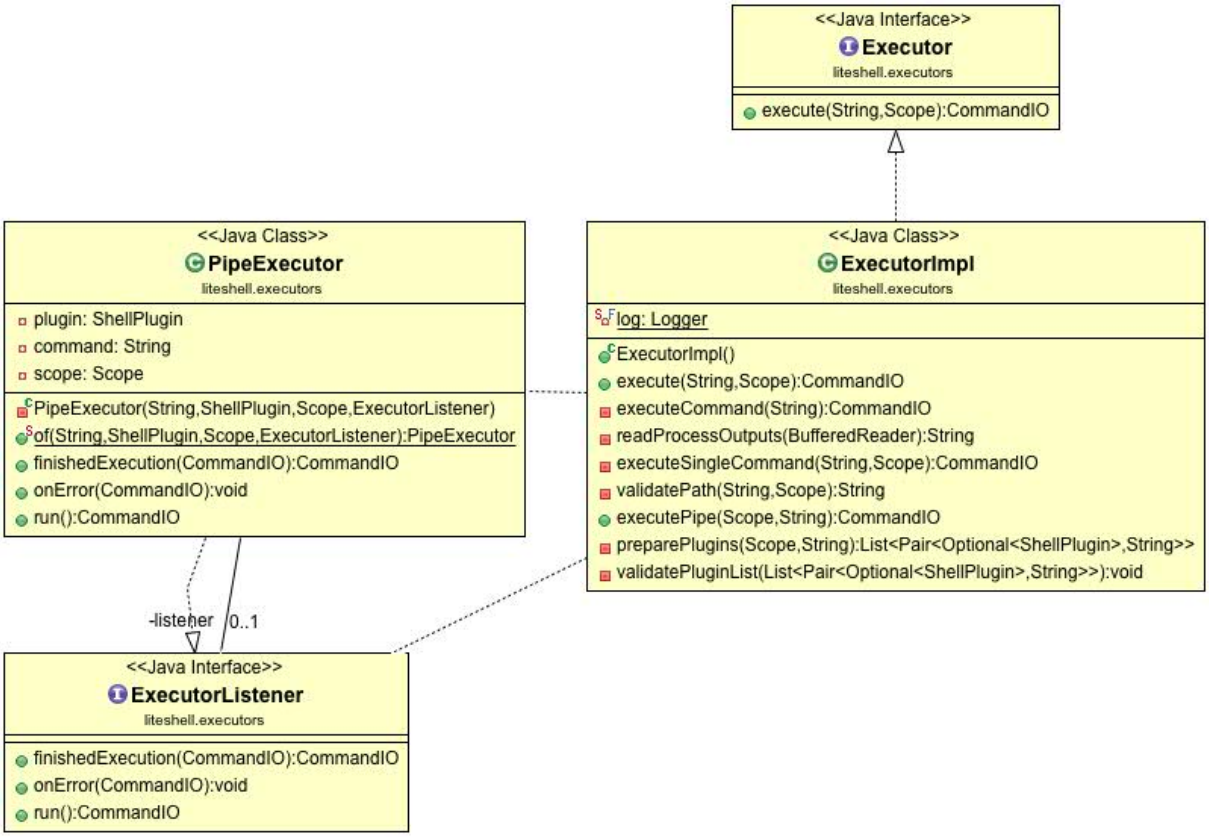
Ako spustit a pouzivat app.

C Diagram tried rozhraní aplikácie



Obrázok C.1: Diagram tried pre rozhrania aplikácie

D Diagram tried rozhraní aplikácie



Obrázok D.1: Diagram tried rozhrní aplikácie