

**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 Kossaczský, 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 |
| Navrhoľ: | doc. Ing. Marek Kukučka, PhD. |
| Abstrakt: | Akupunktúra patrí medzi najstaršie liečebné praktiky sveta a je to jedna z kľúčových častí tradičnej čínskej medicíny. Keďže použitie liečebných metód odvodených od akupunktúry je stále viac rozšírené, z medicínskeho pohľadu je nanejvýš aktuálne venovať sa základnému výskumu v tejto oblasti a pokúsiť sa objasniť základné fyziologické a biofyzikálne mechanizmy stojace za preukázanými klinickými efektami. Z prehľadu publikovaných elektrických vlastností akupunktúrnych bodov a dráh vyplýva potreba dôsledného overenia hypotézy elektrickej rozoznateľnosti akupunktúrnych štruktúr. Očakáva sa, že môžu mať nižšiu impedanciu a vyššiu kapacitu oproti okolitým kontrolným bodom na pokožke. Výstupom mapovania pokožky budú 2D a 3D napäťové/impedančné mapy z povrchu tela. S týmto prístupom bude možné nielen lokalizovať prípadný akupunktúrny bod, ale aj študovať jeho ohraničenie, povrchovú elektrickú štruktúru či jeho veľkosť. Súčasťou výskumu je aj realizovanie meraní závislosti impedancie od frekvencie v akustickom pásme frekvencií 100 Hz – 20 kHz a vplyvu rôznych parametrov na rozoznávanie pozície, tvaru 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 prihlásiť.

| 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, plat- formovo nezávislé konzolové rozhranie |
| Vedúci záverečnej práce: | RnDr. Igor Kossaczky, 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 administrátorské 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 zdrojového kódu na bytecode, 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 pluginou, umožňuje pracovať v interaktívnom, ako aj skriptovacom módu, na rôznych operačných systémoch podporovaných JVM, ako aj rozhrania pomocou, ktorých je možné jednoducho dotvoriť nové pluginy pre rozhranie.

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 desing 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 as well as set of interfaces, which can be implemented to create new plugins for the administrative interface.

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 Kossaczský, CSc.

Bratislava, dňa 14.5.2018

.....
podpis autora

Pod'akovanie

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

Obsah

| | |
|---|----------|
| Úvod | 1 |
| 1 Analýza | 2 |
| 1.1 Operačné systémy | 2 |
| 1.1.1 Windows | 2 |
| 1.1.1.1 Podiel na trhu | 2 |
| 1.1.1.2 Predinštalovaný softvér | 2 |
| 1.1.2 MacOS | 2 |
| 1.1.2.1 Predinštalovaný softvér | 3 |
| 1.1.2.2 Podiel na trhu | 3 |
| 1.1.3 Unix | 3 |
| 1.1.3.1 Predinštalovaný softvér | 3 |
| 1.1.3.2 Podiel na trhu | 3 |
| 1.1.4 Linux | 3 |
| 1.1.4.1 Predinštalovaný softvér | 3 |
| 1.1.4.2 Podiel na trhu | 3 |
| 1.1.5 Porovnanie podielov operačných systémov | 4 |
| 1.2 Programovacie jazyky | 4 |
| 1.2.1 Shell | 6 |
| 1.2.1.1 Výhody | 6 |
| 1.2.1.2 Nevýhody | 6 |
| 1.2.1.3 Popis a zhodnotenie jazyka | 7 |
| 1.2.2 Powershell/Classic command line | 10 |
| 1.2.2.1 Výhody | 10 |
| 1.2.2.2 Nevýhody | 11 |
| 1.2.2.3 Popis a zhodnotenie jazyka | 11 |
| 1.2.3 Python | 12 |
| 1.2.3.1 Výhody | 12 |
| 1.2.3.2 Nevýhody | 13 |
| 1.2.3.3 Popis a zhodnotenie jazyka | 13 |
| 1.3 Existujúce riešenia | 14 |
| 1.3.1 ConEmu | 14 |
| 1.3.1.1 Skúsenosti | 14 |
| 1.3.2 cmdr | 15 |

| | | |
|----------|---|-----------|
| 1.3.2.1 | Skúsenosti | 15 |
| 1.3.3 | Babun | 16 |
| 1.3.4 | MobaXterm | 17 |
| 1.3.4.1 | Neplatená verzia | 17 |
| 1.3.4.2 | Platená verzia | 17 |
| 1.4 | Zhodnotenie analyzovaných technológií | 18 |
| 2 | Preklad jazykov | 19 |
| 2.1 | Kompilátor proces prekladu | 19 |
| 2.1.1 | Lexikálna analýza | 19 |
| 2.1.2 | Syntaktická analýza | 20 |
| 2.1.3 | Limitácia syntaktickej analýzy | 21 |
| 2.1.4 | Semantická analýza | 21 |
| 2.1.5 | Generovanie cieľového jazyka | 21 |
| 3 | Návrh riešenia | 22 |
| 3.1 | Prípady použitia | 22 |
| 3.2 | Popis prípadov použitia | 23 |
| 3.2.1 | Vývojár skriptov | 23 |
| 3.2.1.1 | Spustiť konzolu | 23 |
| 3.2.1.2 | Spustiť príkaz | 23 |
| 3.2.1.3 | Spustiť skript | 24 |
| 3.2.1.4 | Spustiť Shell príkaz | 24 |
| 3.2.1.5 | Spustiť commander príkaz | 25 |
| 3.2.1.6 | Reťazenie príkazov | 25 |
| 3.2.1.7 | Manažovať balíčky | 26 |
| 3.2.1.8 | Stiahnuť nový balíček | 26 |
| 3.2.1.9 | Zmeniť použitý balíček | 27 |
| 3.2.1.10 | Zmazať vybraný balíček | 27 |
| 3.2.1.11 | Získať systémové informácie | 28 |
| 3.2.1.12 | Získať informácie o procesoch | 28 |
| 3.2.1.13 | Vytvoriť skript | 29 |
| 3.2.1.14 | Vytvoriť funkciu | 29 |
| 3.2.1.15 | Override funkcie | 30 |
| 3.2.1.16 | Vytvoriť cyklus | 30 |
| 3.2.1.17 | Vytvoriť podmienku | 31 |

| | | |
|----------|--|-----------|
| 3.2.1.18 | Vytvoriť premenné | 31 |
| 3.2.1.19 | Vykonať základné aritmetické operácie | 32 |
| 3.2.1.20 | Vykonať základné logické operácie | 33 |
| 3.2.1.21 | Presmerovať chybový výstup | 33 |
| 3.2.1.22 | Presmerovať štandardný výstup | 34 |
| 3.2.2 | Vývojár balíčkov | 34 |
| 3.2.2.1 | Implementovať vlastný balíček | 34 |
| 3.2.2.2 | Upravovať existujúce balíčky | 35 |
| 3.3 | Výber programovacieho jazyka | 35 |
| 3.4 | Návrhové vzory | 36 |
| 3.4.1 | Factory - továreň | 36 |
| 3.4.2 | Command - príkaz | 36 |
| 3.5 | Aplikácia | 38 |
| 3.6 | Komponenty aplikácie | 38 |
| 3.7 | Plugin | 39 |
| 3.8 | Diagram tried aplikácie | 41 |
| 3.8.1 | Stručný popis tried | 41 |
| 3.8.1.1 | ScopeImpl | 41 |
| 3.8.1.2 | AbstractExecutor | 42 |
| 3.8.1.3 | ParserImpl | 42 |
| 3.8.1.4 | PluginFactory | 42 |
| 3.8.1.5 | JarLoader | 42 |
| 4 | Implementácia | 43 |
| 4.1 | Hlavná trieda aplikácie - AbstractScope | 43 |
| 4.2 | Štart aplikácie | 43 |
| 4.3 | Inicializácia aplikácie | 44 |
| 4.3.1 | Nahrávanie pluginov | 45 |
| 4.3.1.1 | Načítanie dostupých pluginov z disku | 45 |
| 4.3.1.2 | Načítanie dostupých pluginov z aplikácie | 46 |
| 4.3.1.3 | Finálne načítanie pluginov | 46 |
| 4.4 | Vykonávač príkazov | 46 |
| 4.4.1 | Vykonávač príkazu | 48 |
| 4.4.2 | Vykonávač pajpy | 48 |
| 4.4.3 | Vykonávač skriptu | 48 |

| | |
|---|-----|
| 5 Zhodnotenie výsledkov | 49 |
| Záver | 50 |
| Zoznam použitej literatúry | 51 |
| Prílohy | I |
| A CD s aplikáciou a prácou | II |
| B Návod na spustenie a používanie aplikácie | III |
| C Diagram tried rozhraní aplikácie | IV |
| D Diagram tried rozhraní aplikácie | V |

Zoznam obrázkov a tabuliek

| | | |
|-------------|--|----|
| Obrázok 1 | OS - podiel na trhu podľa statcounter[6] | 4 |
| Obrázok 2 | OS - podiel na trhu podľa netmarketshare[7] | 5 |
| Obrázok 3 | Serverové OS - podiel na trhu podľa w3techs[8] | 5 |
| Obrázok 4 | Serverové OS - podiel na trhu a rozdelenie podľa hodnotenia stránok podľa w3techs[8] | 5 |
| Obrázok 5 | Ukážka ConEmu emulátora | 15 |
| Obrázok 6 | Ukážka Cmder emulátora | 16 |
| Obrázok 7 | Ukážka Babun emulátora | 16 |
| Obrázok 8 | Ukážka práce lexikálneho analyzátora | 20 |
| Obrázok 9 | Ukážka práce syntaktického analyzátora | 20 |
| Obrázok 10 | Prípady použitia pre navrhovanú aplikáciu | 22 |
| Obrázok 11 | Class diagram Factory návrhového vzoru | 36 |
| Obrázok 12 | Class diagram Command návrhového vzoru | 37 |
| Obrázok 13 | Sekvenčný diagram Command návrhového vzoru | 38 |
| Obrázok 14 | Prvé funkčné riešenie | 39 |
| Obrázok 15 | Plugin - diagram tried | 40 |
| Obrázok 16 | Diagram tried aplikácie | 41 |
| Obrázok 17 | Activita spustenia aplikácie | 44 |
| Obrázok 18 | Activita inicializácie aplikácie | 44 |
| Obrázok C.1 | Diagram tried pre rozhrania aplikácie | IV |
| Obrázok D.1 | Activita spustenia aplikácie | V |
| Tabuľka 1 | Ukážka reťazcových prepínačov v podmienkovom výraze if [9] . . . | 10 |
| Tabuľka 2 | Ukážka numerických prepínačov v podmienkovom výraze if [9] . . | 10 |
| Tabuľka 3 | Porovnanie rýchlostí rôznych jazykov[12] | 14 |
| Tabuľka 4 | Use case : Spustiť konzolu | 23 |
| Tabuľka 5 | Use case : Spustiť príkaz | 24 |
| Tabuľka 6 | Use case : Spustiť skript | 24 |
| Tabuľka 7 | Use case : Spustiť shell príkaz | 25 |
| Tabuľka 8 | Use case : Spustiť powershell príkaz | 25 |
| Tabuľka 9 | Use case : Reťazie príkazov | 26 |
| Tabuľka 10 | Use case : Manažovať balíčky | 26 |

| | | |
|------------|--|----|
| Tabuľka 11 | Use case : Stiahnuť nový balíček | 27 |
| Tabuľka 12 | Use case : Zmeniť použitý balíček | 27 |
| Tabuľka 13 | Use case : Zmazať vybraný balíček | 28 |
| Tabuľka 14 | Use case : Získať systémové informácie | 28 |
| Tabuľka 15 | Use case : Získať informácie o procesoch | 29 |
| Tabuľka 16 | Use case : Vytvoriť skript | 29 |
| Tabuľka 17 | Use case : Vytvoriť funkciu | 30 |
| Tabuľka 18 | Use case : Override funkcie | 30 |
| Tabuľka 19 | Use case : Vytvoriť cyklus | 31 |
| Tabuľka 20 | Use case : Vytvoriť podmienku | 31 |
| Tabuľka 21 | Use case : Vytvoriť premenné | 32 |
| Tabuľka 22 | Use case : Vytvoriť funkciu | 32 |
| Tabuľka 23 | Use case : Vytvoriť funkciu | 33 |
| Tabuľka 24 | Use case : Presmerovať chybový výstup | 34 |
| Tabuľka 25 | Use case : Presmerovať štandardný výstup | 34 |
| Tabuľka 26 | Use case : Implementovať vlastný balíček | 35 |
| Tabuľka 27 | Use case : Upravovať existujúce balíčky | 35 |

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

| | | |
|---|---|----|
| 1 | Bash ukážka rôznych volaní for cyklu. [1] | 8 |
| 2 | Bash ukážka volania skriptu s for cyklom priamo z konzoly . [1] | 9 |
| 3 | Ukážka použitia pipe v Powershell. [2] | 12 |
| 4 | Ukážka loadovania balíčka z disku počítača | 45 |
| 5 | Ukážka pseudokódu exekútora. | 47 |

Úvod

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

V prvej časti práce sme sa zamerali na rôznorodosť operačných systémov a absencia jednotnej platformy na vytváranie skriptov vo väčšine prípadov vyžadujú ich duplikovanie alebo viacnásobnú implementáciu. Čiastočným riešením tohto problému je použitie skriptovacieho jazyka s podporou cieľových platform. Zásadným problémom skriptovacích jazykov pri riešení tohto problému je absencia syntaktických a funkčných konštrukcií, ktoré sú už overené a široko používané, ako napríklad paja alebo presmerovanie štandardného a chybového vstupu a výstupu. Okrem skriptovacích jazykov sme si vyskúšali emulátory, ktoré zabezpečujú preklad platformovo špecifického jazyka do jazyka spustiteľného na konkrétnej platforme. Ďalšia časť teoretickej analýzy sa venujeme prekladu jazykov, kde sú popísané, postupy ako fungujú kompilátory programovacích jazykov. Následne sa venujeme návrhu celej práce, kde sme špecifikovali prípady použitia aplikácie, vybrali programovací jazyk pre úspešné vypracovanie zadania, popísali návrhové vzory, ktoré nám umožnili sprehľadniť zdrojový kód a na záver špecifikovali návrh aplikácie spolu s prvotným návrhom tried.

Ako bolo spomenuté výsledkom práce je plne funkčné administrátorské rozhranie, ktoré je jednoducho rozširiteľné pomocou pluginov, umožňuje pracovať v interaktívnom, ako aj skriptovacom módu, na rôznych operačných systémoch podporovaných JVM. Taktiež poskytuje možnosti integrácie s ďalšími enterprise nástrojmi a v neposlednom rade poskytuje knižnicu obsahujúcu rozhrania pomocou, ktorých sa dá funkcionálnosť 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 systému, predstavuje softvér, ktorý umožňuje počítačom pracovať. V poslednej dobe, oblasť operačných systémov prechádza rapidnými zmenami, pretože počítače sa stali súčasťou každodenného života, a to od malých zariadení, napríklad v automobiloch, až po najsofisticovanejšie 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 funkcionálnu 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 MacOS je Python, AppleScript, Ruby, Bash, BOjective-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 server, 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.

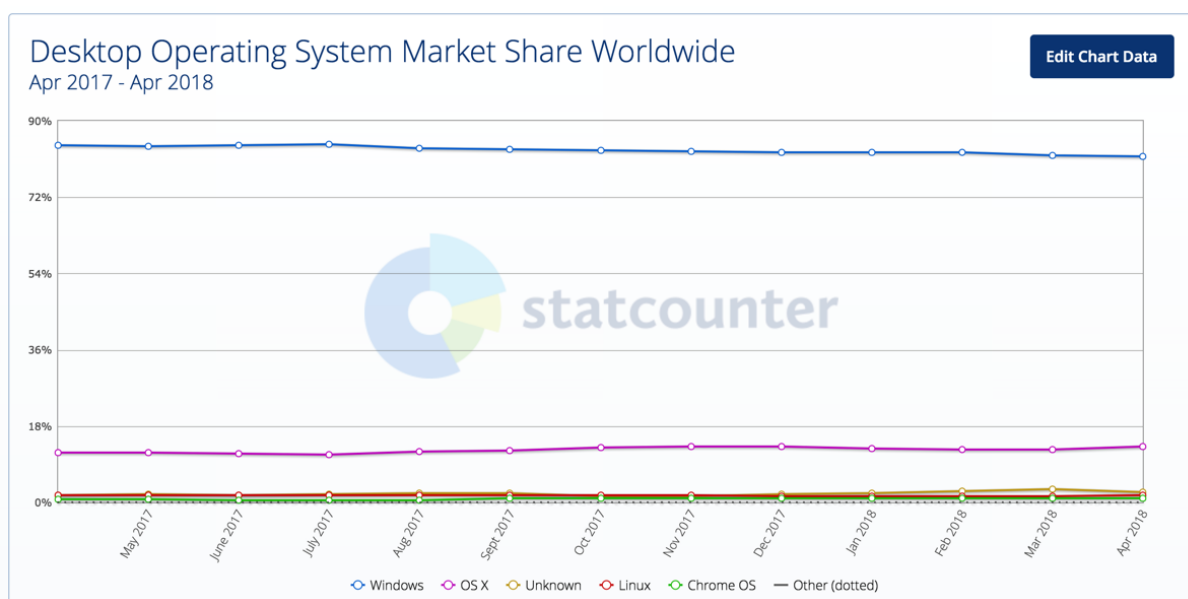
1.1.4.1 Predinštalovaný softvér Predinštalovaný softvér vo väčšine distribúciách Linuxu sú Bash, Open JDK - Java, niektoré distribúcie ponúkajú Python. RedHat začína s podporou .NET frameworku.

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

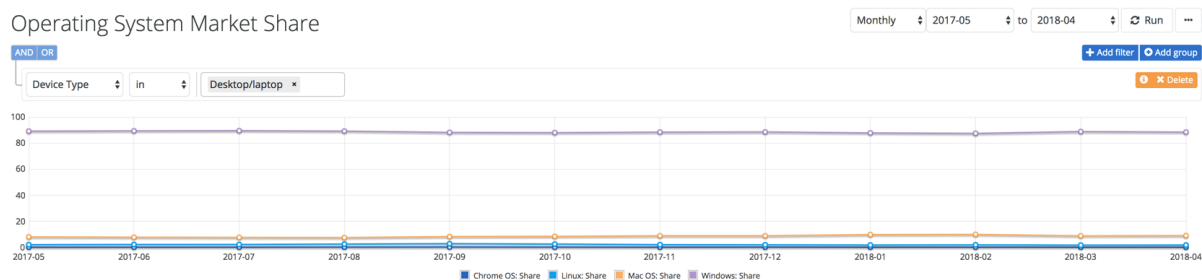
Nasledujúce grafy zobrazujú podiel operačných systémov na trhu v segmente osobných počítačov, ako aj v segmente serverov. Zaujímavý graf je vývoj trendu využívania serverových operačných systémov, z ktorého vidíme, že unixové a linuxové servery zvyšujú svoje podiely na trhu. Naopak, Windows v posledných 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.



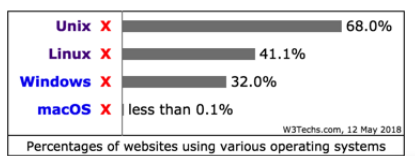
Obrázok 1: OS - podiel na trhu podľa statcounter[6]

1.2 Programovacie jazyky

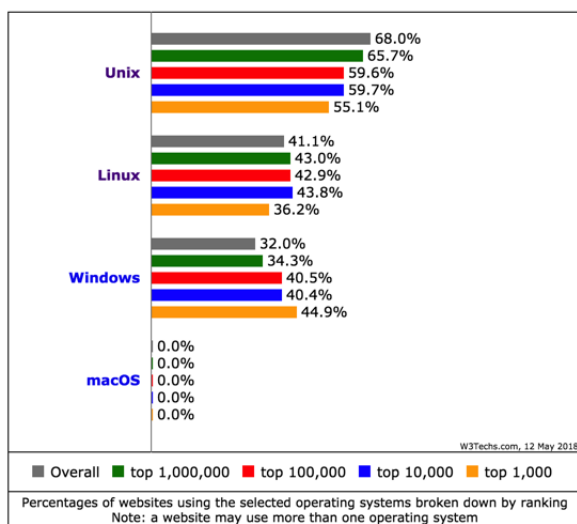
S príchodom osobných počítačov, no najmä serverov, sa programátori zaujímali o automatizáciu procesov, ktoré na danom stroji bolo spočiatku potrebné spúšťať manuálne. Keďže tieto úlohy neboli natoľko komplexné ako samotné programy, ktoré spúšťali, bolo vhodné na úlohy využiť a vytvoriť skriptovacie jazyky. 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 developerami. V nasledujúcich častiach všeobecne zhodnotíme jednotlivé výhody a nevýhody skriptovacieho jazyka Shell.

1.2.1.1 Výhody

- automatizácia často opakujúcich sa úloh,
- možnosť zbíhať 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, 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,
- neprenosný medzi platformami,
- 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 najpoužívanějších 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 | zbíha 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 nasledovnú 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 môže 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 cyklom 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. 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 premennej expr1 väčšia než hodnota premennej expr2. |
| expr1 -ge expr2 | Vráti true, ak je hodnota premennej expr1 väčšia alebo rovná hodnote premennej expr2. |
| expr1 -lt expr2 | Vráti true, ak je hodnota premennej expr1 menšia než hodnota premennej expr2. |
| expr1 -le expr2 | Vráti true, ak je hodnota premennej 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 Powershell/Classic command line

Command line je základným skriptovacím jazykom pre Windows distribúcie, ktorý poskytuje malé API pre svojich používateľov. Aj kvôli uvedenej skutočnosti Microsoft 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 funkcionality 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 framework 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, keďže je jedným z najpopulárnejších programovacích jazykov súčasnosti. Je viacúčelový, patrí medzi 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 píšú 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.

Aj keď sme spomenuli viaceré nedostatky, asi najväčším je rýchlosť. Preto má zmysel nájsť lepšie riešenie daného problému.

| 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[12]

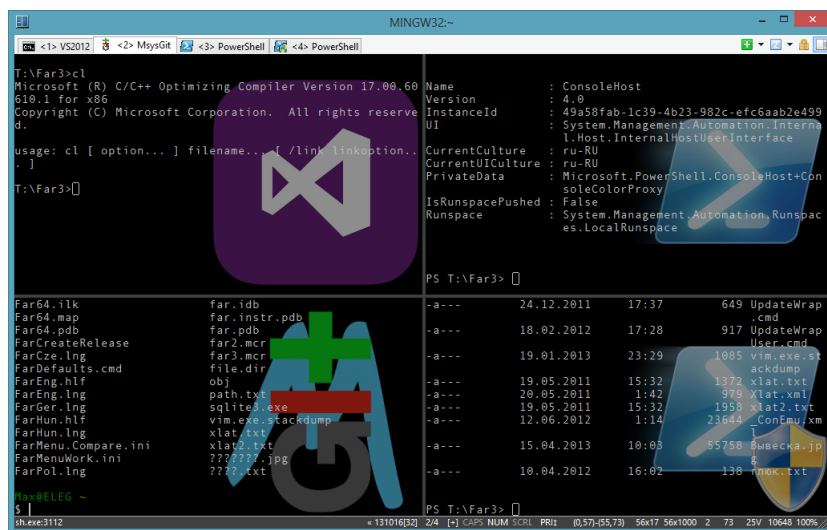
1.3 Existujúce riešenia

Existuje množstvo emulátorov a nástrojov tretích strán, ktoré sprostredkujú 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 issues logu na Githube. Tak ako každý softvér, aj ConEmu je náchylný na chyby. Podľa issues logu sa do oficiálnych vydaní dostávajú rôzne problémy, ktoré neboli zahrnuté v predchádzajúcich 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.

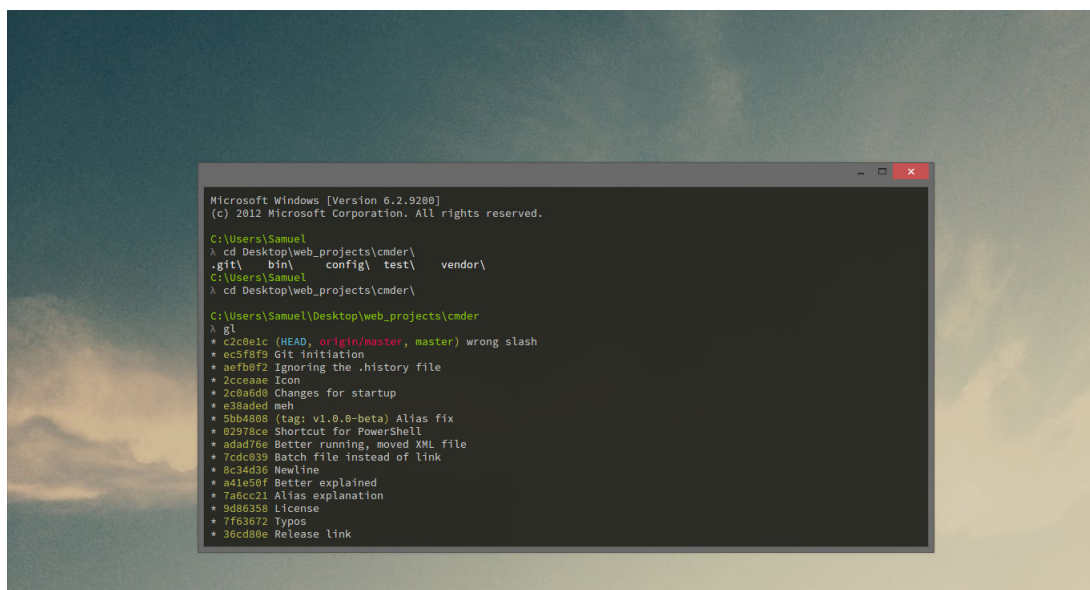


Obrázok 5: Ukážka ConEmu emulátora

1.3.2 cmdr

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 funkcionálnu. [14]

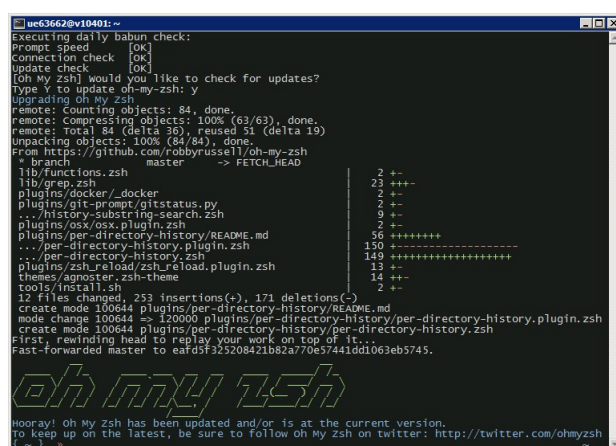
1.3.2.1 Skúsenosti ConEmu je príjemný nástroj, dokáže zjednodušiť prácu, obzvlášť ak je používateľ zvyknutý na programovanie v Shell scripte. Keďže Cmder používa ConEmu ako emulátor Shell terminálu, a je určený pre Windows platformu, nemožno hovoriť o multiplatformovom riešení.



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. Prenositelnosť 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 zaťaženy 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 centrálného 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 vlastnosti 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
- syntaktická analýza
- spracovanie sémantiky
- generovanie cieľového jazyka

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

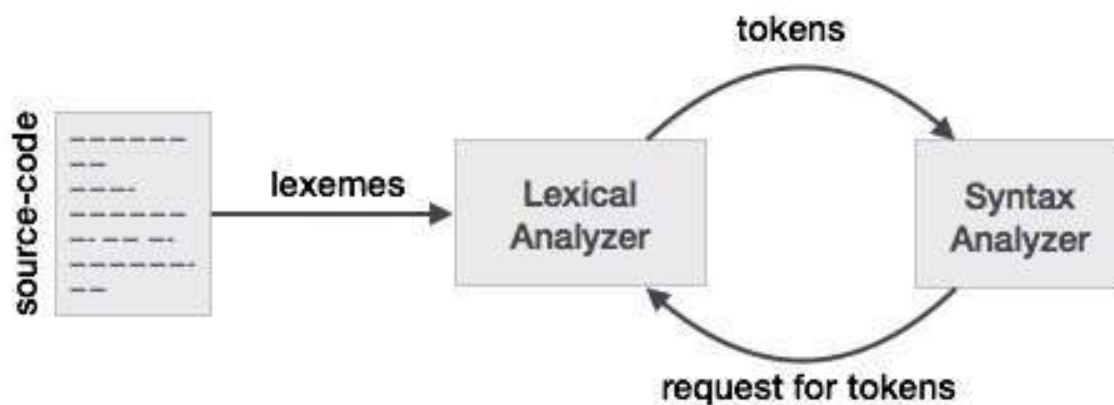
2.1.1 Lexikálna analýza

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 analyzátor nieje schopný nájsť pattern pred danú lexému musí vyhlásiť chybu počas tokenizácie.

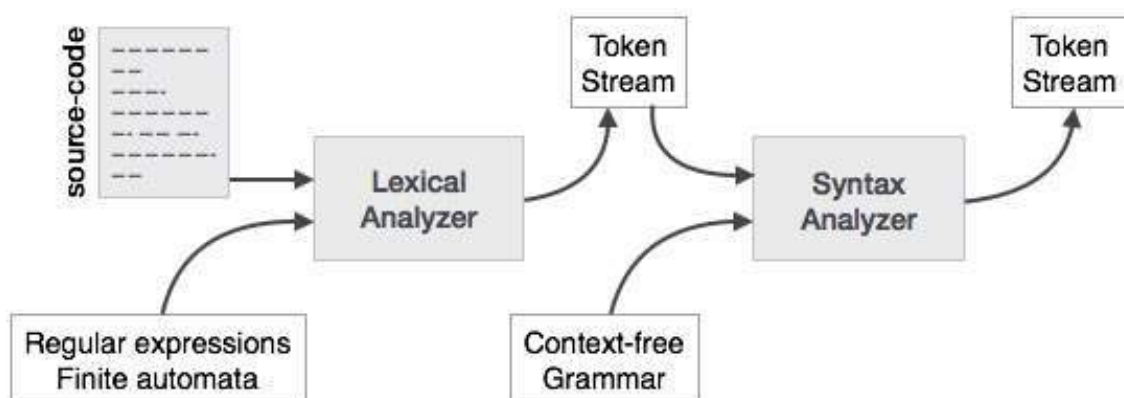
Výstupom z lexikálnej analýzy sú takzvané tokeny, ktoré tvoria vyššie jednotky jazyka ako kľúčové slová jazyka, konštanty, identifikátory, operátory a iné.[17]



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 syntactickej jednotky. Syntaktická analýza sa radí medzi bezkontextové gramatiky. Po skončení syntactickej analýzy prichádza na rad sémantická analýza.[17]



Obrázok 9: Ukážka práce syntactickeho 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 rozhoduje 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:

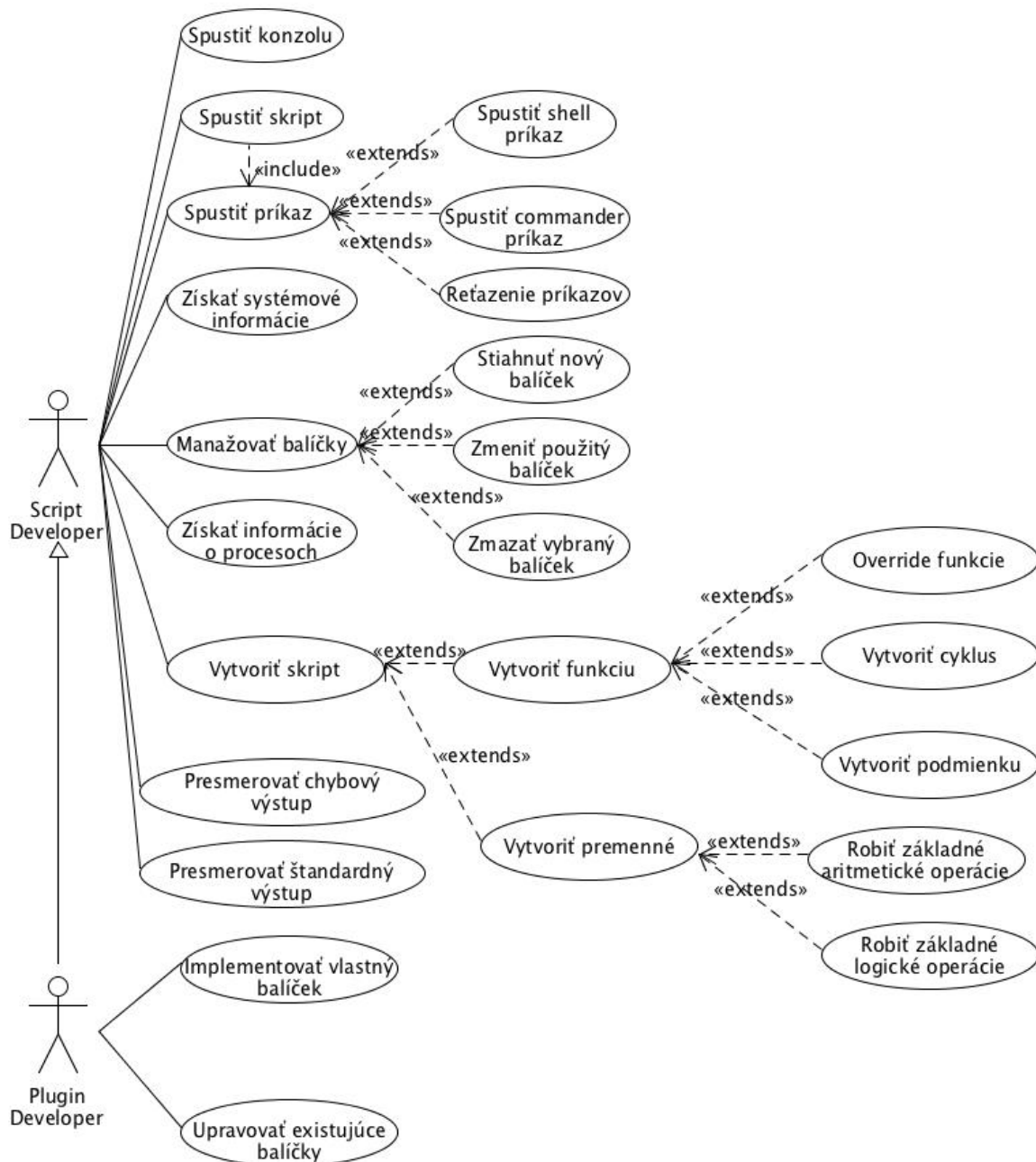
- zisťovanie dosahu definovaných tokenov takzvaný scoping
- kontrola typov
- deklarácia premenných
- definícia premenných
- viacnásobná deklarácia premenných v jedno scope

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 Spustiť konzolu

| | |
|------------------|---|
| Use case | 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 Spustiť príkaz

| | |
|------------------|---|
| Use case | 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 Spustiť skript

| | |
|------------------|---|
| Use case | 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 zadaná 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, tracom. |

Tabuľka 6: Use case : Spustiť skript

3.2.1.4 Spustiť Shell príkaz

| | |
|------------------|--|
| Use case | 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 Spustiť commander príkaz

| | |
|------------------|---|
| Use case | 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 začínajúci 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 Reťazenie príkazov

| | |
|------------------|---|
| Use case | 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 odovzdávajú výstupy nasledovníkovi po úspešnom ukončení. Príkazy sa vykonávajú dovtedy, pokiaľ 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 Manažovať balíčky

| | |
|------------------|--|
| Use case | Manažovať balíčky |
| Podmienky | Shell aplikácia musí byť spustená. |
| Vstup | Textový reťazec obsahujúci príkaz pkg a jeho argumenty. |
| Popis | Používateľ bude schopný nahradiť, 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 Stiahnuť nový balíček

| | |
|------------------|---|
| Use case | Stiahnuť nový balíček |
| Podmienky | Shell aplikácia musí byť spustená. Príkaz na stiahnutie balíčka musí byť správne zadáný. |
| Vstup | Textový reťazec obsahujúci príkaz "pkg download <názov balíčka s verzou>" |
| 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 Zmeniť použitý balíček

| | |
|------------------|--|
| Use case | Zmeniť použitý balíček |
| Podmienky | Shell aplikácia musí byť spustená. Príkaz na zmenu používaného balíčka musí byť správne zadáný. |
| Vstup | Textový reťazec obsahujúci príkaz "pkg change <názov nahradzujúceho balíčka s verziou> |
| Popis | Program zmení používaný balíček z aktuálne používaného na balíček vybraný používateľom. Táto voľba je aplikovateľná iba pre spravovanie verzií existujúcich balíčkov. V prípade, že nahradzujúci balíček nie je dostupný lokálne, používateľ bude vyzvaný stiahnuť 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 Zmazať vybraný balíček

| | |
|------------------|--|
| Use case | 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 zadáný. |
| 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 Získať systémové informácie

| | |
|------------------|--|
| Use case | 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 Získať informácie o procesoch

| | |
|------------------|---|
| Use case | 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 Vytvoriť skript

| | |
|------------------|---|
| Use case | 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 behu 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 Vytvoriť funkciu

| | |
|------------------|--|
| Use case | Vytvoriť funkciu |
| Podmienky | Používateľ musí mať prístup k akémukoľvek textovému editoru. |

| | |
|---------------|---|
| Vstup | Funkcia musí byť správne zadaná. 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 Override funkcie

| | |
|------------------|--|
| Use case | 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 zadaného syntaxu je problém zapísaný na štandardný chybový výstup a vykonávanie skriptu je ukončené. |

Tabuľka 18: Use case : Override funkcie

3.2.1.16 Vytvoriť cyklus

| | |
|------------------|--|
| Use case | Vytvoriť cyklus |
| Podmienky | Používateľ musí mať prístup k akémukoľvek textovému editoru. |

| | |
|---------------|--|
| Vstup | Cyklus musí byť správne zadaný. 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 Vytvoriť podmienku

| | |
|------------------|--|
| Use case | Vytvoriť podmienku |
| Podmienky | Používateľ musí mať prístup k akémukoľvek textovému editoru. |
| Vstup | Podmienka musí byť správne zadaná. 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 Vytvoriť premenné

| | |
|------------------|--|
| Use case | 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 premennú 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 Vykonať základné aritmetické operácie

| | |
|------------------|--|
| Use case | 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 Vykonať základné logické operácie

| | |
|------------------|--|
| Use case | Vykonať základné logické operácie |
| Podmienky | Používateľ musí mať prístup k akémukoľvek textovému editoru. |
| Vstup | <p>Premenná musí byť správne zadefinovaná.</p> <p>Syntax pre definovanie /zmenu hodnoty premennej:</p> <p><názov premennej> = <výraz>; alebo</p> <p><typ> <názov premennej> = <výraz>;</p> <p>, kde výraz môže byť operácia nad číselnými hodnotami, číselnými premennými, ako aj nad pravdivostnými.</p> <p>Vzťahy medzi číselnými hodnotami musia byť definované logickými operátormi - <, >, <=, >=, ==, !=.</p> <p>Vzťahy medzi pravdivostnými hodnotami musia byť definované logickými operátormi : ==, !=, , &&.</p> |
| 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 Presmerovať chybový výstup

| | |
|------------------|---|
| Use case | Presmerovať chybový výstup |
| Vstup | Pre presmerovanie na chybový výstup je potrebné dodržať syntax <code>command stderr> file</code> |
| 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 Presmerovať štandardný výstup

| | |
|------------------|--|
| Use case | 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 <code>command stdout> file</code> |
| Popis | Program presmeruje štandardný výstup tam, kam mu používateľ v príkaze zdefinuje. |
| 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 funkcionality v zmysle rozširovania API, ktoré môže vývojár skriptov používať pre efektívnejšiu prácu.

3.2.2.1 Implementovať vlastný balíček

| | |
|------------------|--|
| Use case | 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 funkcionálnosť 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 Upravovať existujúce balíčky

| | |
|------------------|---|
| Use case | 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 funkcionálnosť 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-

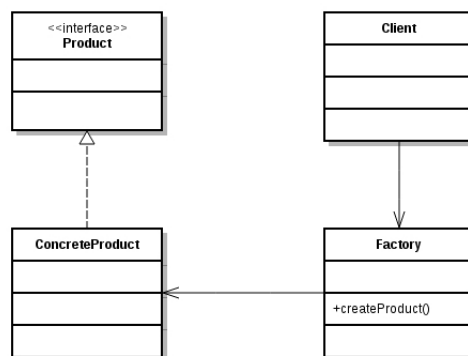
code, ktorý je rovnaký pre všetky PC. Pomocou jednoduchého programu je byte-code preložený do jazyka, ktorému rozumie konkrétny PC. Java je objektovo-orientovaný programovací jazyk, čo znamená že rovnako ako v živote, aj v Jave je všetko tvorené objektami. Obsahuje široké spektrum knižníc, ktoré slúžia nielen na vývoj webových, ale aj desktopových aplikácií. Podporuje multitreading, čo umožňuje vytvárať efektívne programy pre počítače s viacjadrovým procesorom.[19]

3.4 Návrhové vzory

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

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. Diagram návrhového vzoru je možné vidieť na nasledujúcom obrázku.

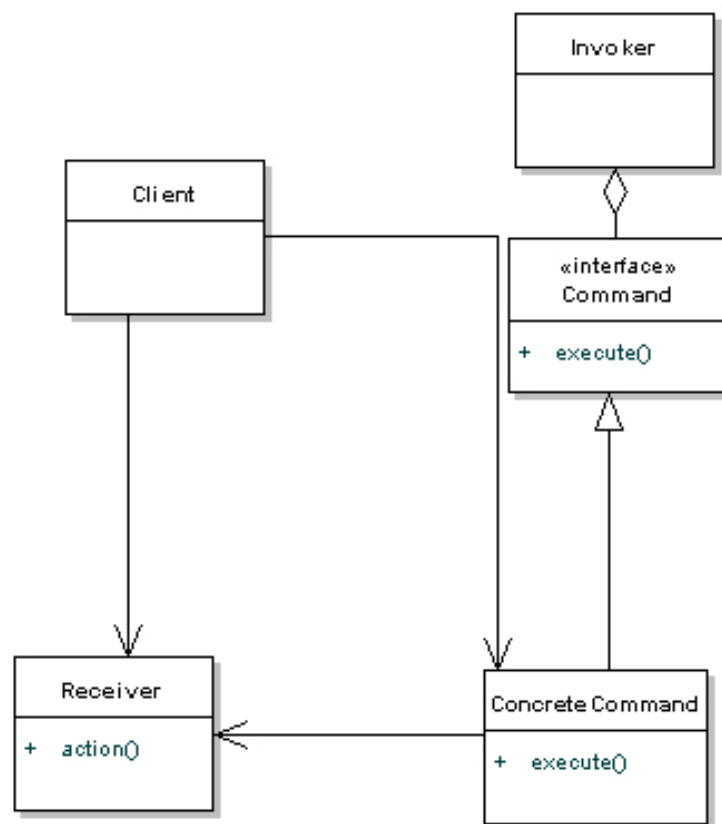


Obrázok 11: Class diagram Factory návrhového vzoru

3.4.2 Command - príkaz

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

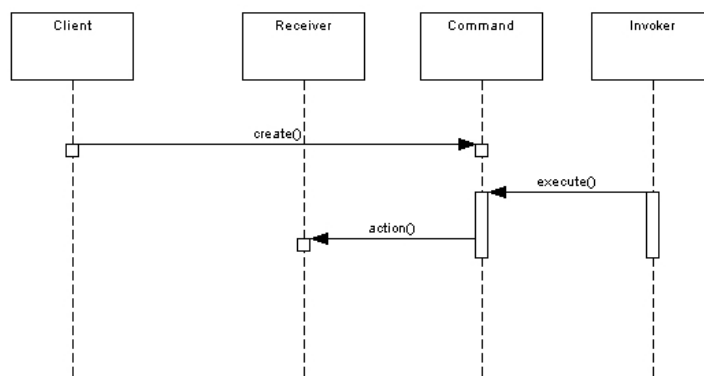
Command vzor deklaruje rozhranie pre všetky budúce commandy a zároveň execute() metódu, ktorú s vypýta Receiver commandu aby splnil požadovanú operáciu. Receiver je



Obrázok 12: Class diagram Command návrhového vzoru

objekt, ktorý vie ako požadovanú operáciu splniť. Invoker pozná command a pomocou implementovanej `execute()` metódy dokáže vyvolať požadovanú operáciu. Klient potrebuje implementovať `ConcreteCommand` a nastaviť `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 `Receiver`.

Pre lepšie pochopenie je proces zobrazený aj na sekvenčnom diagrame.



Obrázok 13: Sekvenčný diagram Command návrhového vzoru

3.5 Aplikácia

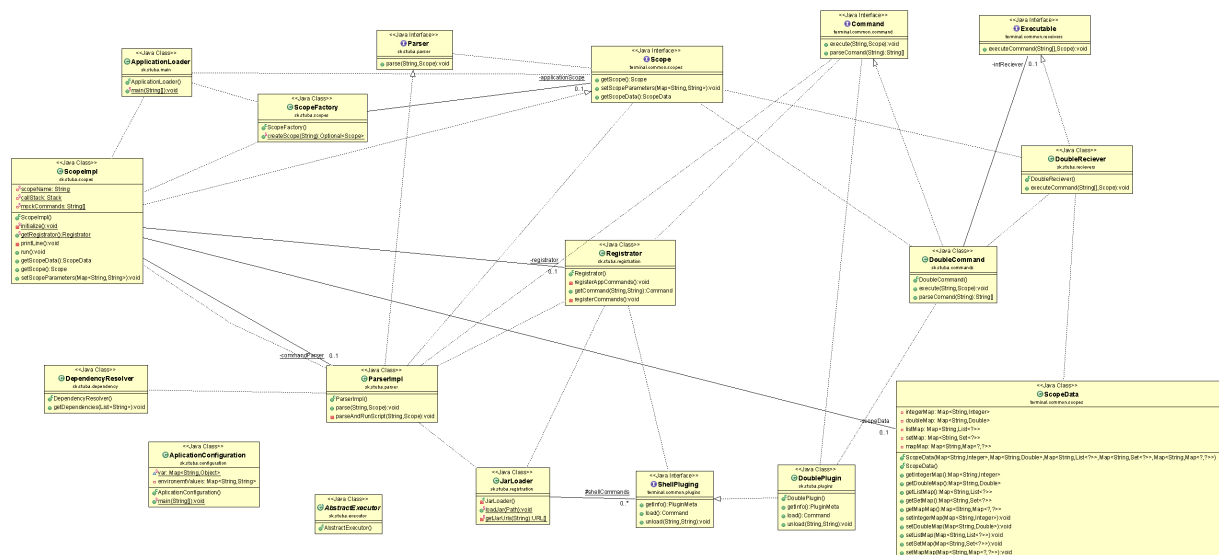
Pre implementáciu vlastného riešenia bolo potrebné na základe prípadov použitia identifikovať časti, z ktorých bude aplikácia pozostávať. Naším zámerom nebolo vytvorenie jednej veľkej aplikácie, ktorá by sa mohla časom stať neudržiavateľnou, ale aplikácia, ktorá umožní používateľom interaktívny aj skriptovací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

3.6 Komponenty aplikácie

Po vybratí 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 udržiavanie dát v jednotlivých scopoch,
- ShellPlugin - komponent, ktorý nesie implementáciu príkazov.

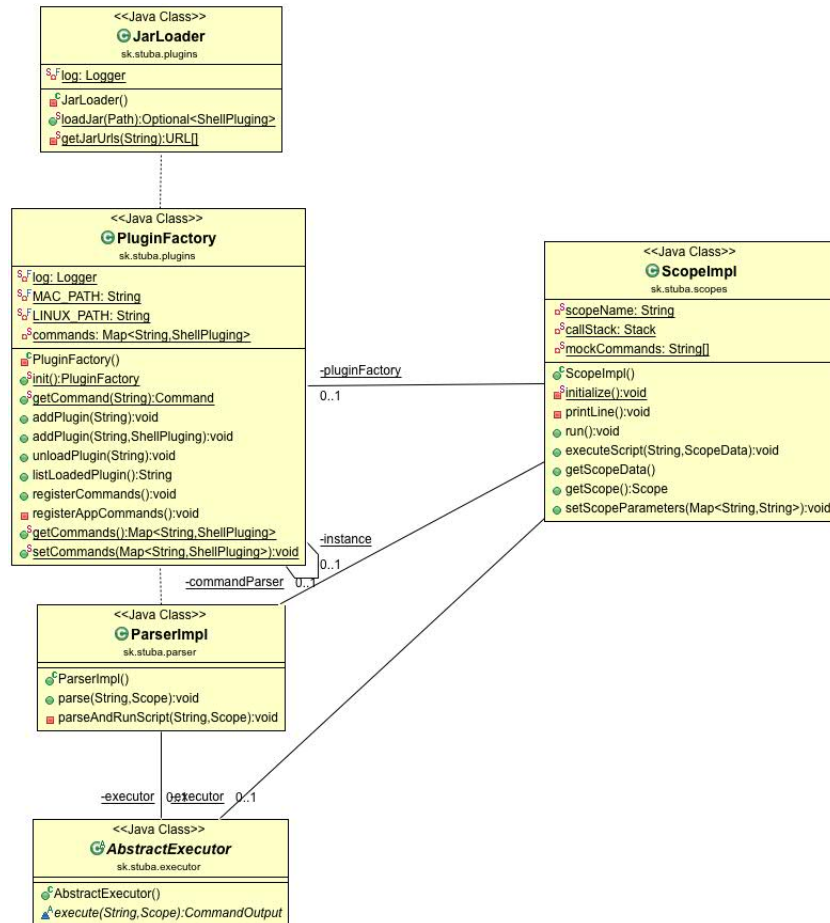


Obrázok 14: Prvé funkčné riešenie

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 funkcionálít do programu. Diagram implementácie rozhraní a konkrétnych tried je viditeľný na nasledovnom obrázku.

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 scopeu 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 scope nám slúži na informovanie jednotlivých prijímačov(Receiver), 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. Na získavanie dostupných pluginov sa používa trieda JarLoader.

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

4 Implementácia

V tejto kapitole sa budeme venovať konkrétnemu riešeniu daného problému. Kapitola nás postupne prevedie cez jednotlivé kroky implementácie riešenia. Každý krok budeme popisovať jednou z nasledovných možností alebo kombináciou : pomocou diagramu aktivít, diagramu tried, pseudokódom prípadne útržkami zdrojového kódu z fungujúcej aplikácie. Jednotlivé sekcie práce sme rodelili na základe toho akými smermi sa môže aplikácia uberať.

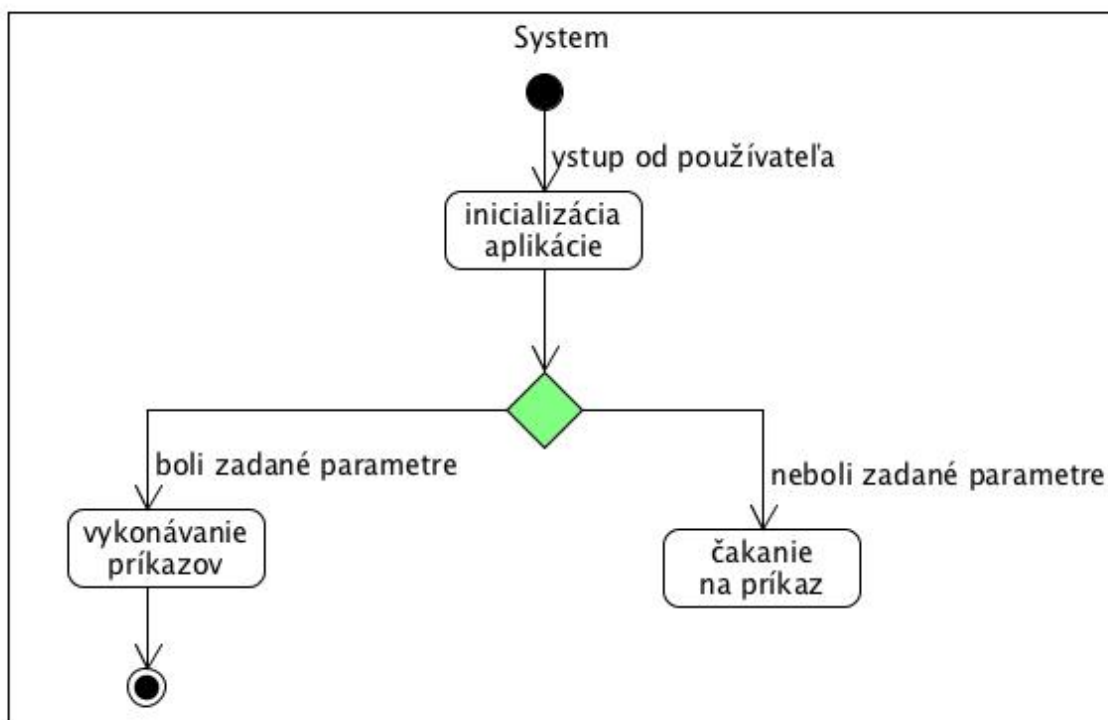
4.1 Hlavná trieda aplikácie - AbstractScope

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 scopeu 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 na vstupe akékoľvek vstupné parametre. Túto funkcionality sme pridali z dôvodu integrácie s inými aplikáciami.

4.2 Štart aplikácie

Na nasledovnom obrázku je zobrazená aktivita spustenia konzoly. Ako vidieť z diagramu aplikácia sa najprv nainicializuje, čo to znamená si povieme v sekcii 4.1. Po inicializácii premenných sa nám vytvorí inštancia triedy ScopeImpl, ktorá je implementáciou AbstractScope tried, ktorú sme si predstavili v predchádzajúcej sekcii práce. V ďalšom kroku zistí či má na vstupe parametre. V prípade, že áno vykoná ich, inak spustí konzolu, ktorá čaká na vstup od používateľa. V tejto časti si popíšeme čo sa deje v prvom prípade a teda, že máme zadané parametre.

Možnosť s parametrami na vstupe sme potrebovali implementovať kvôli tomu aby sa dala aplikácia integrovať s inými enterprise aplikáciami. Tie môžu buď čítať výstupy priamo z konzoly alebo si ich môžu presmerovať do súboru. Novo vytvorený súbor si následne môžu načítať a vykonať nad ním potrebné operácie.



Obrázok 17: Activita spustenia aplikácie

4.3 Inicializácia aplikácie

Inicializácia pozostáva z dvoch krokov :

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



Obrázok 18: Activita inicializácie aplikácie

4.3.1 Nahrávanie pluginov

Nahrávanie pluginov má na starosti trieda `PluginFactory` a `JarLoader`, proces nahrátia 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, ktoré 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 vracia 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 4 Ukážka loadovania 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 dostupých pluginov z aplikácie Pluginy môžu byť definované priamov a plikácii. V tomto prípade priamo do rovnakej mapy ako v predchádzajúcom prípade vkladáme inštanacie pluginov a informáciách o nich.

4.3.1.3 Finálne načítanie pluginov Používateľ môže počas využívania aplikácie, stiahnuť alebo naprogramovať viaceré verzie rovnakého pluginu. Z tohoto dôvodu sme naimplementovali 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 pomôžem pseudokódом rozhodovacieho algoritmu. Zo pseudokódu jasne vidieť, že exekútor si 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 sa zisťuje či sa má výstup z programu zapísať mimo štandardného vstupu alebo výstupu. Ak áno program očakáva validnú cestu za oddeľovačmi `stdout>` resp. `stderr>` kde sa pokúsi zapísať svoj výstup do súboru. Diagram tried je uvedený v prílohe D.

Algoritmus 5 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 zobere `command` a zavolá na ňom metódu `execute(CommandIO commandInput, Optional<Scope> scope)`. `Command` následne predspracuje vstupy v metóde `parseCommand(Stream<String> var)`. Metóda `execute` následne 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 rozná, že v príkaze ide o pajpu, príkaz putuje do triedy `PipeExecutor`, kde sa rozparsuje na parciálne príkazy. Po rozprarovaní príkazov sa každému príkazu nastaví nasledujúci príkaz, ktorý sa má vykonať. Posledný príkaz neobsahuje žiadnen 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 exekútoru z 4.1.

4.4.3 Vykonávač skriptu

5 Zhodnotenie výsledkov

Zatiaľ sa toho nespravilo hodne ale verím, že sa to tu cele zaplní.

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 riešení, 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%22Flaptop%22%5D%7D%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%20S%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. LUDOVÍT MOLNÁR Milan Česka, Bořivoj Melichar. *Gramatiky a jazyky*. 1. vyd. Bratislava : Alfa, 1987, 2004. MDT: 519.682(075.8).
18. ORACLE. *What is Java technology and why do I need it?* 01-2018. Dostupné tiež z: https://www.java.com/en/download/faq/whatis_java.xml.
19. WALTER SAVITCH, Kenrick Mock. *Absolute Java*. 6. vyd. Pearson, 15-04-2015, 2015. ISBN: 978-0134041674.

Prílohy

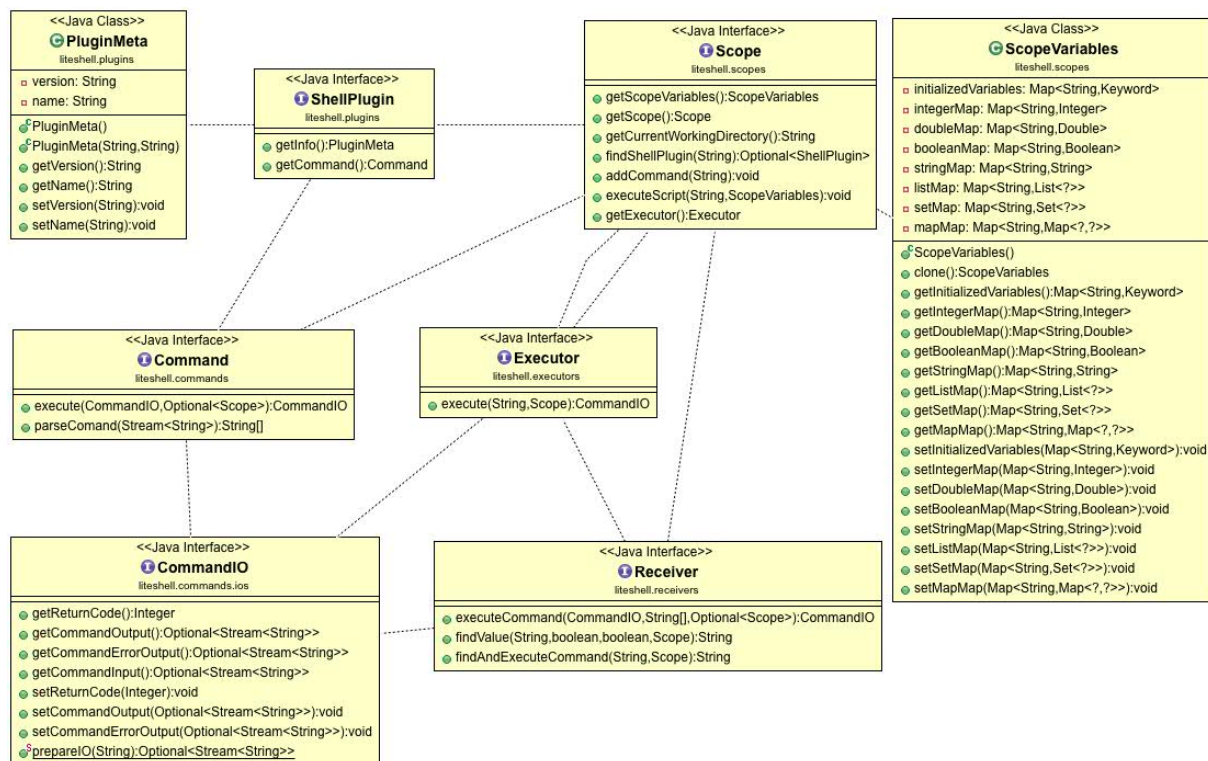
| | | |
|---|---|-----|
| A | CD s aplikáciou a prácou | II |
| B | Návod na spustenie a používanie aplikácie | III |
| C | Diagram tried rozhraní aplikácie | IV |
| D | Diagram tried rozhraní aplikácie | V |

A CD s aplikáciou a prácou

B Návod na spustenie a používanie aplikácie

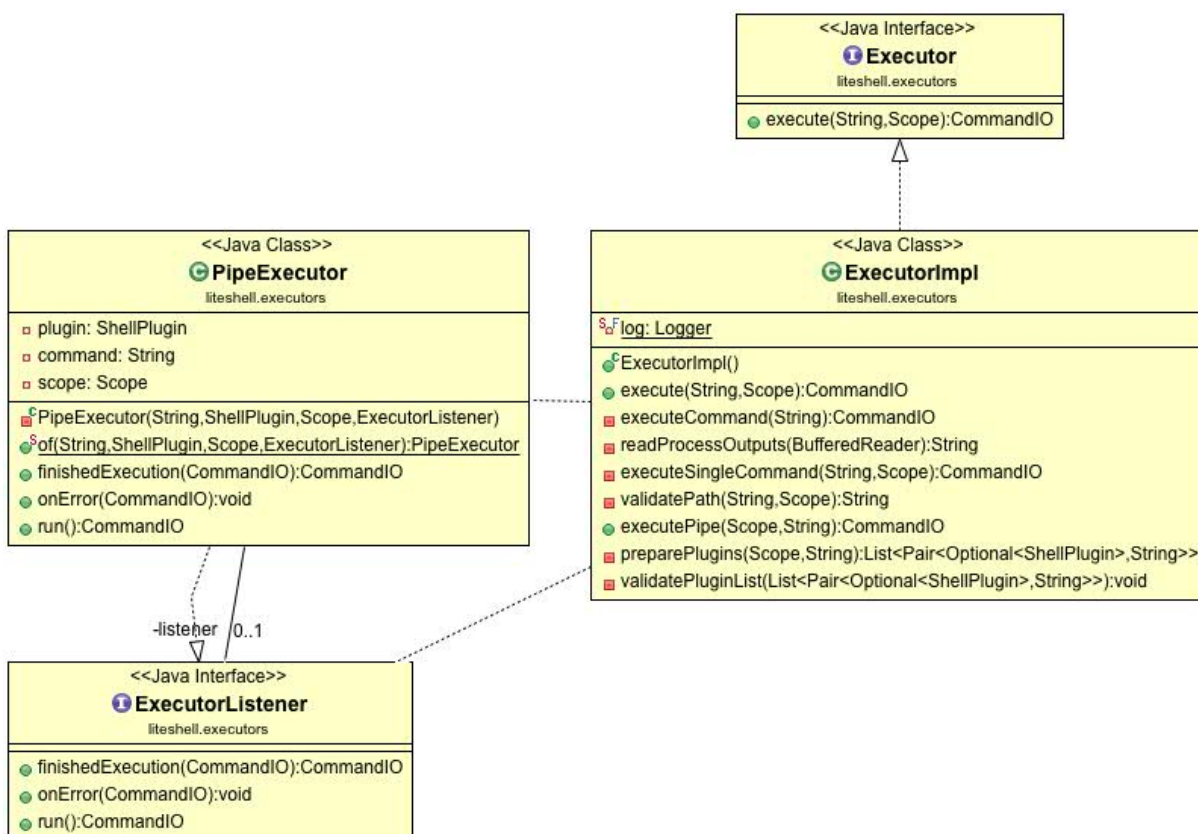
Ako spustiť a používať app.

C Diagram tried rozhraní aplikácie



Obrázok C.1: Diagram tried pre rozhrania aplikácie

D Diagram tried rozhraní aplikácie



Obrázok D.1: Aktivita spustenia aplikácie