Sem vložte zadání Vaší práce.

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra softwarového inženýrství

Diplomová práce

Grafická reprezentace příkazové řádky unixového typu

Bc. Ondřej Kmoch

Vedoucí práce: Ing. Viktor Černý

5. května 2015

Poděkování

Děkuji zejména vedoucímu diplomové práce Ing. Viktoru Černému za vstřícnost, trpělivost a optimismus, které mi pomohly při její tvorbě. Dále bych rád poděkoval své rodině, jejíž nezlomná podpora během mnoha let studií mi pomohla dostat se až k okamžiku, kdy je touto prací mohu formálně završit.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 5. května 2015 .....

České vysoké učení technické v Praze
Fakulta informačních technologií

Odkaz na tuto práci

Kmoch, Ondřej. *Grafická reprezentace příkazové řádky unixového typu.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2015.

Abstrakt

I na začátku 21. století má textová příkazová řádka v operačních systémech stále pevné místo. Obvyklé je to u systémů založených na Unixu, kde její schopnosti často zcela překonávají možnosti grafického uživatelského rozhraní. Platí to zejména pro jednu její často používanou schopnost - pracovat efektivně s filtračními nástroji. Používání příkazové řádky, řetězů a filtrů ovšem není jednoduché a klade velké mnemotechnické nároky na uživatele. Tato práce se zabývá myšlenkou, jestli je možné tuto podmnožinu funkčnosti příkazové řádky uživateli usnadnit pomocí interaktivního grafického rozhraní. Smyslem této práce je na něj definovat uživatelské požadavky, zjistit problémy spojené s jeho tvorbou a pokusit se je vyřešit. Cílem je pak využít těchto závěrů a vytvořit funkční prototyp programu, nazvaný SPsim (Shell Pipe Simulator).

Klíčová slova SPsim, CLI, GUI, debugging roury, unix, grafický simulátor, extrakce příkazů, extrakce manuálových stránek

Abstract

At the beginning of 21st. century, the text command line still has its own place in operating systems. This is often the case with those based on Unix, where its CLI capabilities often completely surpass abilities of graphical user

interface. This is especially true for one of its commonly used functionalities - to work effectively by using the filter commands. Using of the command line, pipes and filters, is, though, not an easy task, and is putting a a lot of mnenonic requirements on user. This thesis is concerned about whether it is possible for such subset of functionalities to be simplified for user by use of graphical representation. Goal of this thesis is to define requirements for such environment, find out possible obstacles connected with achieving this task, and try to solve them. Final product of this work is to utilize such findings and creation of a working prototype of such program, called SPsim (Shell Pipe Simulator).

Keywords SPsim, CLI, GUI, pipe debugger, unix, graphical simulator, command extraction, man pages extraction

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Příkazová řádka (CLI) je fenomén, který doprovází počítače téměř od jejich vzniku, a rozhodně od dob, kdy se díky jejímu zavedení staly počítače rozšířené a relativně uživatelsky přívětivé. V dnešní době jsou příkazové řádky využívány pro svou rychlost a eleganci neustále, a to nejenom jako systémový relikt tam, kde se nedá využít grafické rozhraní (GUI), ale i v běžném použití. Mnoho IT profesionálů dává stále přednost textovým editorům jako je *vim* před kancelářskými editory či grafickými vývojovými rozhraními. Někteří dokonce využívají hybridní grafickou příkazovou řádku integrovanou do pracovního prostředí namísto klasické pracovní plochy a jako zvláštní windows manager (například Xmonad). Její oblíbenost přetrvává právě kvůli její rychlosti, možnosti plného ovládání bez nutnosti myši a podpoře velkého množství funkcí. Výhoda příkazové řádky tak spočívá i ve využití klávesových zkratek a mnoha programů a příkazů, které nemusí mít komplikované a na vývoj náročné GUI, ale dají se spouštět a kombinovat právě její pomocí. Příkazová řádka si tak od dob svého vzniku uchovala charakteristickou filozofii používání, a týká se spíše programování, administrace, vývoje a dávkového zpracování, které může uživatel průběžně ovlivňovat. Do dnešních dob zůstala v Unixových systémech (zejména Linuxu, ale využití má snad ve všech existujících OS) příkazová řádka jako naprosto nenahraditelný nástroj ovládání a práce se systémem, a tyto operační systémy často dokonce ani grafické rozhraní nemají či mít nemusí. S tím jak se interprety příkazové řádky a programy pro ní s postupem času vyvíjely, rozšiřovaly se i jejich schopnosti. Mnoho nástrojů a programů vzniklých v éře příkazové řádky si zachovalo svojí jedinečnost do dnešních dob, a stále za ně fakticky neexistuje vhodná náhrada. Mezi takové nástroje, komunikující s uživatelem skrze příkazovou řádku, patří například filtry. Filtry jsou dobře napsané, rychlé programy, které berou nějaký vstup, nad ním provedou požadovanou operaci, a vrátí jej upravený výstup. Tyto programy se v unixové příkazové řádce navíc dají řetězit do tzv. „Rour“ [1],

[1] anglicky Pipe

při kterých dochází k předávání vstupů a výstupů mezi příkazy v řadě za sebou. Díky rourám se tak se tak dá sestavit skript, který umožňuje rychle, bez nutnosti meziukládání výsledků na disk, i velice komplikované a přesně customizované operace a úpravy. Výstupu z takovýchto filtrů pak může navíc v tom samém řetězci příkazů použít další program, který tyto zpracované informace využije jinak.

Příkazová řádka má ovšem zásadní nevýhodu. Je ve své definici vlastně jednorozměrná. Silná konfigurovatelnost nástrojů (zde opět například filtračních) tak klade na velké mnemotechnické nároky na tvůrce takového skriptu a jeho paměť. Je vyžadována podrobná znalost přesné syntaxe příkazů a jejich přepínačů, společně s jasnou představou jak vypadá vstup a jak má vypadat výstup každého prvku roury. Tato náročnost je přímo úměrná složitosti zpracování a nároků na konečný výstup. Tvorba takové roury a posloupnosti příkazů je pak často otázkou praxe a metody pokus/omyl, která částečně smazává její výhody pro méně zdatné a zkušené uživatele.

Příkazová řádka je tedy v kombinaci s vhodnými programy a řetězením velice mocný a rychlý sluha, který je ovšem dosti náročný na zvládnutí a naučení. Smyslem této práce je prozkoumat a vytvořit zjednodušené řešení, které by tuto problematickou fázi tvorby a ladění rour, zejména s filtry, dovedl uživateli usnadnit. Pomoci má tomu právě grafické uživatelské prostředí, které vhodným způsobem reprezentuje a ladí příkazovou řádku s programy a přepínači tak, aby jí uživatel mohl interaktivně budovat a testovat. Tento program by měl i uživateli pokud možno napovídat, například prostřednictvím zobrazení momentálně relevantních manuálů a jejich vhodných pasáží.

Obsah této práce

Tato diplomová práce se skládá ze sedmi kapitol, včetně úvodu a závěru.

V První kapitole (Úvod) je nastíněno co to příkazová řádka v počítači je, jak ovlivňuje používání počítačů v dnešní době a zhodnocení její užitečnosti. Je jsou zde nastíněna úskalí spojena s jejím používáním, a ukázání směru, kterému se věnuje zbytek diplomové práce - zpracovávání příkazů do rour v uživatelsky přívětivějším grafickém rozhraní. Krátká sekce je věnována historii příkazové řádky, od prvních počítačů až po nástup grafických rozhraní, a význam pro dnešního uživatele.

Druhá kapitola se věnuje stanovení Cílů této diplomové práce. Je zde více rozepsáno, jaké překážky se vyskytují při tvorbě rour s příkazy (hlavní důraz je opět kladen na příkazy filtrů), a snaží se ospravedlnit potřebu pro nástroj, který by tuto tvorbu usnadňoval. Je zde popsána cílová skupina uživatelů. Jsou zde zjištěny překážky, kterým se nejspíše bude čelit při jeho tvorbě. Dvě sekce této kapitoly se zvlášť zabývají požadavky na samostatný výsledný program

SPsim, tedy jeho definovanou funkčnost a použitelnost, co by daný program měl konkrétně umět. Druhá z těchto sekcí se věnuje spíše obecným požadavkům na nástroje, prostředky a překážky, které vyplývají z analýzy problému, jak jich využít a dosáhnout, a jak čelit případným problémům.

Třetí kapitola, Analýza, se věnuje detailnějšímu prozkoumání příkazové řádky a její podmnožiny dané zadáním (důraz na stavění a zpracovávání filtračních rour). Je zde analýza podmnožiny funkcí a schopností, které by měl výsledný program umět. Věnuje se diskuzi nad vhodnými technologiemi, programovacímu jazyku, omezením vyplývajícím z požadavků a následným kompromisům. Je diskutován vhodný grafický vzhled a rozvržení aplikace včetně alternativ. Důraz je kladen i na popis problémů, na které se během vývoje a analýzy narazilo, a jejich řešení. Obsahem kapitoly je u jednotlivých problémů i diskuze a porovnání s jinými možnými přístupy, než těmi zvolenými. Součástí kapitoly je i rešerše, která popisuje existující alternativy k SPsim (či jejich absenci), včetně analýzy jejich přístupu k debuggingu a zda plní cíle zadané touto prací. Analýza je rozdělena do několika sekcí, a každá z nich je ukončena závěrem a jeho přehledným shrnutím v několika bodech.

Čtvrtá kapitola, Implementace, se věnuje architektuře a tvorbě samotného programu. Je zde popsáno, jak přesně program funguje a co dělá, je rozdělen a analyzován z pohledu logických celků. K lepšímu pochopení je kapitola doplněna o UML modely a vysvětlující diagramy. Okrajově se zde věnuje použitému vývojovému prostředí a funkcím, které tvorbu usnadnily.

Pátá kapitola se věnuje testování, jak byl program testován, a proč.

Šestá kapitola je zhodnocení, jak se podařilo splnit cíle této Diplomové práce. Pokud ano, tak jaké úpravy, problémy či kompromisy byly zjištěny a provedeny. Souhrnně reflektuje původní zamýšlené cíle s konečným výsledkem. Část této kapitoly se věnuje i možnému budoucímu pokračování vývoje programu, funkcím, které by se daly do programu přidat, a obecně možnostem jak jej dále rozvíjet.

Poslední kapitola je Závěr, kde je zhodnocen přínos této diplomové práce pro autora, chyby, kterých se dopustil, jak se poučil, čeho by se příště vyvaroval. Shrnutím je zda práce měla smysl a zda se dá výsledek považovat za úspěšný.

Příkazová řádka a její historie

Příkazová řádka je fenomén, pod kterým si představíme něco co provází počítače již od jejich vzniku. Příkaz (ve svém klasickém smyslu, tedy povel vykonat nějakou činnost) jako takový je pevně spjatý se samotnou funkčností počítače

a vlastně jakéhokoliv automatu. Příkazy ve formátu „Udělej X“ jsou v takových případech doplněny booleovskou logikou, která umožňuje rozhodování. Soubor příkazů má pak možnost, specifikovanou programátorem, volat na základě svých výstupů podmnožinu sebe samých. Výsledkem může být téměř libovolně komplikovaný a složitý software.

Zatímco toto byla spíše obecná definice příkazu jako takového, pod příkazovou řádkou si většina lidí představí něco jiného. Příkazová řádka je textové rozhraní, které umožňuje zapisovat příkazy, spouštět je, případně ovlivňovat jejich nastavení, průběh, vstup a výstup, a to takřka v reálném čase. V éře počítačů se chápe často jako původní způsob interakce s počítačem, ačkoliv to není pravda.

Samotná „Příkazová řádka CLI“ [2] vznikla postupným vývojem z *dávkových monitorů* [3], které byly připojeny k systémovou konzoli a toto řešení mělo řadu omezení, která příkazová řádka překonala. Nový model interakce s uživatelem pomocí příkazových řádek spočíval v transakční sérii požadavků a odpovědí na tyto požadavky. Požadavky pak byly vyjádřeny pomocí textových příkazů, dodržujících nějaký speciální slovník a pravidla zápisu. Výhoda tohoto řešení proti dávkovým monitorům byla zejména v rychlosti interakce s uživatelem, latence takové komunikace se počítala ve vteřinách namísto dřívějších hodin až dní. Díky tomu umožnily systémy s příkazovou řádkou uživatelům měnit stavy v pozdější fázi nějaké zpracovávané transakce, a to prakticky v reálném čase (nebo blízké reálnému času) a získávat ihned zpětnou vazbu. Tato změna umožnila že programy mohly být interaktivní tak, jak by dříve nikoho nenapadlo.

Nicméně ani příkazová řádka není zcela bez nevýhod. Tento interface sice umožňuje uživateli interagovat s během programu či nějakou transakcí téměř v reálném čase, ukládá ale na něj zároveň i velké požadavky na paměť a orientaci v textu. Uživatel tak, aby byl dobrý v práci s příkazovou řádkou, musí pro to investovat mnoho času a snahy, a to jak studiem tak praxí.

Rozhraní pomocí příkazových řádek se začala rozvíjet s nástupem „počítačů se sdílením pracovního času“, tedy rozvrhnutím úkolů které počítač postupně zpracovával mezi více programů a uživatelů. Koncept takového sdílení času se datuje do 50. let 20. století, nicméně zcela nejvýznamnějším systémem poskytujícím rozhraní skrze příkazovou řádku se stal v roce 1969 UNIX. Význam systémů založených na UNIXu, včetně jeho pozdějších derivací, trvá do dnešních dob a velmi ovlivnil prakticky všechny pozdější systémy.

Nejstarší systémy s příkazovou řádkou kombinovaly Teletyp (Dálnopis)0.1, který byl spojením tiskárny a klávesnice, s počítačem. Původní využití tele-

[2] anglicky CLI - command line interface

[3] anglicky batch monitors, byly to programy napevno přítomné v počítači které spouštěný program mohl volat a využívat, používaly se například pro lepší informování o chybách

typu jako nástroj komunikace mezi dvěma lidmi na vzdálenost (Teletyp vznikl jako evoluce telegrafu již na začátku 20. století) - se přirozeně využil jako vhodný nástroj komunikace člověka s počítačem. Využití teletypů mělo dvě výhody. Z uživatelského hlediska byli lidé pracující s počítačem zvyklí na jeho rozhraní a uměli jej používat. Z ekonomického hlediska umožnil využití stávající technologie, a tím i finanční úsporu.

V polovině 70. let 20. století došlo k velkému rozšíření videoterminálů (VDT)0.1, a tím začala další éra a rozvoj příkazové řádky. Zobrazení řádků na obrazovce (fungující na principu zářících bodů na fosforové vrstvě) vykreslovaly text v porovnání s tiskárnou teletypu mnohem rychleji (kde bylo nutné přesouvat tiskovou hlavu po papíře). To vedlo k dalšímu snížení času odezvy mezi uživatelem a systémem a také snížení nákladů, které v takovém případě interaktivní vedení programu mělo. Na dlouhou dobu se tak běžnou představou počítače se z velké krabice stala televizní obrazovka s textovými řádky, přinášející člověku pracujícímu s terminálem mnohem větší komfort. Uvedení těchto obrazovek do ovládání počítačů skrze příkazovou řádku s okamžitou odezvou a možností rychlé modifikace umožnilo tvůrcům programů také dále rozvíjet její schopnosti. Tím začala vznikat rozhraní, která se díky různým prvkům tvořeným vhodnými znaky mohla považovat za předchůdce dnešních grafických rozhraní. CLI tak dalo vzniknout jednoduchým počítačovým hrám a textovým editorům - mezi jejich nástupce se dá považovat například stále populární editor *vim*.

Obrázek 0.1: ukázka teletypu ASR33 s rozhraním pro zpracování děrných štítků a videoterminálu televideo 925 ze 70. let (zdroj: wikipedia.org, Arnold Reinhold, CC)

Kapitola 1

Definice problému a cílů práce

Původním požadavkem ze zadání je „vytvořit grafickou reprezentaci unixové příkazové řádky, která umožní provádět definovanou podmnožinu funkcionality běžné příkazové řádky.“ Cílem je tedy nástroj, konkrétně program, který by umožnil nějakou úroveň simulace příkazové řádky. Důvody pro to jsou stanoveny částí „Zaměřte se především na příkazy typu filtr“. Každý kdo se kdy setkal s pokročilejším využíváním příkazové řádky v Unixu (Linuxu) ví, že obsahuje mnoho mocných filtračních nástrojů. Nástrojů je množství, a většinu z nich jde ještě bohatě parametrizovat tak, aby dávaly přesně požadované výsledky. K tomu všemu navíc lze tyto příkazy řetězit, dokonce i vícekrát za sebou. Běžná činnost očekávaného budoucího uživatele programu tedy bude transformace nějakých textových vstupů (které mohou, ale nemusí být reprezentovány souborem se zdrojovým textem, může jít například i o výstup nějakého jiného příkazu) na základě konkrétních požadavků, jejich zpracování a výstup v požadovaném formátu.

Uživatel s tímto požadavkem na unixovou příkazovou řádku přichází do styku hned s několika problémy. Hlavním problémem, který dokáže většinu uživatelů hned z kraje odradit je pohled na samotnou černou linku s blikajícím kurzorem a představou, že následující hodinu stráví zkoumáním dokumentací, návodů na internetu a hledáním jak napsat zpětnou jednoduchou uvozovku ‘. Toto „odrazování“ by šlo redukovat nějakým vhodným grafickým nástrojem, který by byl většině uživatelů zvyklých na grafické rozhraní příjemnější. Výsledkem by tedy měla být nějaká abstrakce textových příkazů do grafického a nejlépe klikacího rozhraní rozhraní podporující myš. Dalším velmi častým problémem při přípravě a tvorbě takového skriptu (a to je obvyklá bolest i mnohem zkušenějších uživatelů), že si jednoduše nepamatují, či ani netuší, jaké nástroje pro daný problém je vlastně vhodné použít. Nebo mají na výběr z více příkazů a neví pro jaký je lépe se rozhodnout. A i když tuší, tak už neví, jaké použít operátory. Tudíž, dobrý a vhodný grafický reprezentant a ladič CLI by měl uživateli i v tomto ohledu nějak napovědět.

Problém s takovou inteligentní nápovědou je v tom, že rozpoznávání po-

třeby uživatele na základě nějakého vágního uživatelova zadání daného problému by bylo velice obtížné. Byla by fakticky nutná pokročilá, učící se umělá inteligence. Takový systém, pro zobrazení vhodné nápovědy, by musel nejen z nedokonalého popisu pochopit co uživatel po něm chce, ale ještě najít vhodné řešení, nebo jeho části. Tuto schopnost prozatím v reálném použití suplují internetové vyhledávače, které dovedou na základě klíčových slov sice ne přímo uživateli napovědět, ale díky obrovskému množství indexovaných dat které mají k dispozici, odkázat na co nejrelevantnější téma, které se zabývá něčím podobným. Uživatel je v takovém případě obvykle odkázán na diskusi, kde měl někdo podobný dotaz, a ostatní lidé se mu snažili poradit. Tuto funkčnost tedy necháme budoucnosti. Nabízí se využít zbývající, už existující a dobře zavedený prostředek, a tím jsou manuálové stránky.

Manuálové stránky jsou v unixových systémech prakticky od začátku a těší se velké popularitě.[4] Poskytují podobné rozhraní (a tudíž by měly dodržovat nějaký stanovený formát) ideálně pro každý příkaz a program pro příkazovou řádku. Je v nich obvykle několik „kapitol“ které popisují zadaný program či funkci. Manuálové stránky mají několik výhod. Jsou ustálené jako de-facto standard [5], dodržují ve většině případů podobné principy a požadavky na formu jsou jasné a dostatečně stručné [6](právě pro to aby s v nich daly rychleji najít požadované informace). Manuálové stránky ovšem nejsou bez nevýhod. Jsou vždy pouze v anglickém jazyce (s tím se nedá příliš nic moc dělat, možnost automatického překladu není součástí této práce), což již dnes není tak velký problém, ale pořád se může najít dost uživatelů kteří mají s technickou angličtinou manuálových stránek potíže. Druhá a asi ta největší nevýhoda (a to zejména pro začátečníky) je ta, že se nedovedou v manuálových stránkách orientovat. Není příliš uživatelsky přívětivé hledat v částečně formátovaném textu nějakou konkrétní pomoc. Manuálové stránky jsou tedy dobré na získání informací, špatné jako výuková učebnice (na internetu je lépe najít množství ukázek a příkladů). Nejlépe slouží jako zdroj informací těm uživatelům, kteří už přesně vědí co chtějí hledat (a zajímá je například jen syntax příkazu). S pokročilostí uživatele roste i užitná hodnota manuálových stránek. Grafický program reprezentující příkazovou řádku se schopností debuggingu by tedy ideálně měl využívat těchto manuálových stránek (je to navíc dáno tím že na různých platformách mohou být různé verze stejných příkazů), a poskytovat uživateli už co nejužší množství informací, relevantní k tomu co právě uživatel

[4] první dva roky byl UNIX prakticky bez dokumentace, první skutečné manuálové stránky vytvořili Denis Ritchie a Ken Thompson na naléhání svého manažera Douga McIlroye v roce 1971

[5] některé grafické programy, zejména ty postavené na grafických manažerech Gnome a KDE se snaží poskytovat manuály i v HTML formátu

[6] `http://babbage.cs.qc.edu/courses/cs701/Handouts/man_pages.html`, `http://www.fnal.gov/docs/products/ups/ReferenceManual/html/manpages.html`, `http://archive09.linux.com/feature/34212`

dělá. Cílem této práce by tedy celkově mělo být usnadnit uživatelům tvorbu skriptů s rourami a nástroji, zejména filtračními. Vzhledem k uvedeným obtížím při jejich klasické tvorbě přímo v příkazové řádce je nasnadě, že takový program má smysl. Bude - li navíc proveden dobře, jeho použití umožní (díky univerzálnosti unixové příkazové řádky a manuálových stránek) použití i na na poměrně širokém spektru různých odnoží a distribucí, bez ohledu na konkrétní verze využívaných příkazů.

Výsledný program SPsim tedy má potenciál oslovit díky své schopnosti nabízet automaticky aktuální a užší nápovědu (konkrétní relevantní pasáže manuálových stránek) jak odborníky, tak méně zkušené uživatele. Oboum navíc usnadní tvorbu tím, že pro složitější příkazy bude díky grafickému interaktivnímu provedení lépe vidět a odlaďovat jednotlivé části rour bez ohledu na její celkovou délku a složitost. K tomu všemu by měl být program schopen fungovat napříč různým unixovým prostředím.

1.1 Cíle programu SPsim

Smyslem této části je definovat funkčnost samostatného programu a výsledného produktu SPsim. Vzhledem k výše uvedenému rozboru se došlo k závěru, že SPsim by měl mít přehledné uživatelské prostředí, náhled na data a příkazy v rouře, je vhodné aby zobrazoval nějakou „personalizovanou“ nápovědu relevantní vůči momentálnímu stavu příkazů v aplikaci. Nutná podmínka je aby také umožňoval ladění jednotlivých funkčních částí roury bez ohledu na okolí. Praktická funkčnost by měla obsahovat i možnost přesouvání už vytvořených příkazů a průběžné přizpůsobování roury. To vše v univerzální aplikaci, která půjde ideálně spustit i na více platformách. Uživatel by měl mít také možnost zpětného převedení grafického formátu na textový - aby jej mohl zpětně použít přímo v příkazové řádce.

Očekávání od výsledného programu se tedy dají shrnout do těchto bodů:

- Program reprezentuje rouru s příkazy podobně jako příkazová řádka.
- GUI by mělo být interaktivní.
- Roura by měla jít průběžně vytvářet, rozšiřovat či zmenšovat bez ohledu na její zbytek
- Program by měl poskytovat nápovědu relevantní k používaným příkazům a jejich operátorům.
- Uživatel by měl mít možnost náhledu na skutečnou podobu simulované roury, jako je v CLI

- Program by měl umět automaticky získávat informace o podporovaných příkazech a zejména operátorech
- Program musí umět spouštět a testovat jednotlivé části roury bez ohledu či ovlivnění na jejich okolí
- Program by měl být co nejvíce multiplatformní
- Program by měl mít podobu desktopové aplikace a případně webového appletu

1.2 Cíle písemné práce

Výše uvedené požadavky a očekávání se týkají výsledného produktu - spustitelného softwaru. Tato práce se ovšem musí detailněji zabývat i překážkami které k daným funkcím vedou. V souladu s jednotlivými body předchozího odstavce se při tvorbě vyskytnou obtíže a problémy vyžadující nepřímočará řešení, které je nutné hlouběji analyzovat a překonat, anebo odůvodnit, proč bylo něco uděláno jinak nebo proč se požadavky musely pozměnit. V průběhu práce na programu se navíc stalo, že některá počáteční očekávání byla pozměněna (jmenovitě verze v podobě appletu, detailněji viz kapitola 2.) Cílem písemné části je tedy zaměřit se i na tyto nalezené problémy. Opomeňme nyní na chvíli formu programu, a proveďme krátkou analýzu požadavků na software, se kterými se nejspíše budou vyskytovat problémy, a jejich řešení není přímočaré a vyžaduje hlubší analýzu: *(Součástí kapitoly s analýzou je v diplomové práci ve všech hlavních případech i úvaha, proč se dané problémy řešily zrovna takto a jaké by byly další možné přístupy).*

- Roura by měla jít průběžně vytvářet, rozšiřovat či zmenšovat bez ohledu na její zbytek

Při tvorbě klasické roury se jedná o jeden skript. Jeho ladění přichází tradičním způsobem v úvahu tak, že se roura buduje postupně, z leva do prava. Uživatel si testuje samostatné příkazy s připraveným vstupem, a ladí jejich výstup. Má-li nějakou rudimentární představu o tom, že výstup daného příkazu je v požadované formě, připojí k rouře další příkaz a postup opakuje.

Program by však měl zvládat jednotlivé ladění komponent přímo v rouře. Cílem je tedy zodpovědět otázku jak najít vhodný způsob spouštění roury a jejich příkazů tak, aby zůstaly na sebe navázány *a zároveň* přitom bylo možné testovat jednotlivé komponenty i nezávisle (a ideálně jim modifikovat vstup a výstup).

- Program by měl poskytovat nápovědu relevantní k používaným příkazům a jejich operátorům.

V předchozím textu bylo řečeno, že manuálové stránky, navzdory jistým pravidlům kterých se obvykle drží, nejsou příliš přehledné. Ideální stav je aby v programu byl nabízen obsah relevantní ke konkrétní činnosti (příkazu), se kterým uživatel pracuje. Je tedy nutné nějak v manuálových stránkách daného příkazu najít automaticky relevantní pasáž (a navíc počítat s tím že na různých systémech mohou být různé verze). Tento problém je problémem inteligentního parsování manuálových stránek, cílem je tedy prozkoumat možnosti takové extrakce, spojení dané informace s grafickými bloky programu a zvolení nejlepšího řešení, které k tomuto kroku vede.

- Program by měl umět automaticky získávat informace o podporovaných příkazech a zejména operátorech

Tímto bodem se myslí, aby program uměl sám ze systému zjistit podporované příkazy (podporovaným se zde myslí nějaká zadaná podmnožina, viz zadání práce), poradit si s tím že mohou být různých verzí, mít různá množství (nebo žádné) parametry a že to samé platí i o jejich operátorech - ty také mohou mít různou syntaxi, různá množství vlastních parametrů. Cílem je prozkoumat možnosti inteligentní automatizace, která toto zvládne. Program pak umožní uživateli v grafickém prostředí pracovat s příkazy v takové podobě, jakou daný operační systém skutečně podporuje.

Cílem písemné práce je tedy kromě popisu analýzy programu a jeho tvorby s testováním ještě teoreticky rozebrat a vyřešit několik složitějších problémů týkajících se data-miningu ze systémových manuálových stránek a simulace se sebou spojených příkazů, které vyžadují detailnější analýzu a hlubší znalost problematiky. Součástí je i diskuze proč byly zvoleny vybrané přístupy a řešení, a současně s tím i návrh jiných možných postupů.

Kapitola 2

Analýza a návrh

Tato kapitola se zabývá rozsáhlejším rozborem problémů, stanovení požadavků na výsledný program a ujasnění si vlastností před samotnou implementací. Dělí se na sedm podkapitol, které mají samy několik sekcí. V nich se práce věnuje jednotlivým fázím analýzy a tvorbě podkladů pro implementaci. Současně u všech důležitějších částí probíhá i diskuze a zamyšlení nad tím, zda je nakonec zvolené řešení vhodné, a jak by se případně dala daná věc řešit jinak.

Na začátku se rozebere unixový shell, a pokusí se srovnat textové řešení CLI s dalšími a ospravedlnit požadavek, aby byl simulátor plně grafický. Je diskutována multiplatformnost a omezen rozsah podporovaných systémů. Dále se udělá analýza příkazů, stanovení jejich množiny a rozbor samotné Roury, jak vypadá a jak souvisí s příkazy a programy, které se v ní spouštějí. Je navrženo jak tyto části reprezentovat ve výsledném simulátoru, v čem se inspirovat a co upravit podle vlastních požadavků. Další sekci je věnována rešerše a zjištění, zda již existují podobné programy. Rešerše je zároveň inspirací pro vhodnou formu debuggingu a ladění. Rozebrán bude způsob, jak vhodně ladit a testovat příkazy v rouře.

Další části pojednávají o využití konkrétního interpretu příkazů v samotném programu, proč je lepší na pozadí GUI programu využívat "naživo"skutečný shell se skutečnými příkazy, a nesnažit se nahradit úplně vše vlastním typem sandboxu. Předchozí poznatky jsou potom využity ve stanovení požadavků a návrhu na vzhled GUI samotného programu. Část se věnuje rozboru a výběru možných rozvržení (layoutů) a debatuje se jejich vhodnost a nevhodnost pro toto použití. Jsou zde uvažovány možnosti nasazení aplikace jako applet, desktopový program nebo webová aplikace a volba vhodného jazyka z hlediska vývoje i multiplatformnosti. Velký důraz je zde kladen především na způsob spouštění individuálních příkazů a částí rour. Poslední částí je analýza možností strojového získávání informací o podporovaných příkazech hostujícího operačního systému, jejich operátorů a nápověd.

2.1 Shell, Pipe, Příkazy, Filtry

V této sekci jsou rozebrány různé formy CLI v komunikaci s uživatelem (v našem případě pro vytváření rour), a je následně odůvodněno, proč je nejvhodnější právě grafická reprezentace zpracovávaných úkonů. Určí se omezení splňující požadavek multiplatformnosti. Dále je věnován prostor příkazům a datovým proudům, jak je bude vhodné zachytávat a reprezentovat, a jakou množinu příkazů vůbec podporovat a proč. Analyzuje se použití příkazů v samotné rouře v CLI a jak bude rouru vhodné využívat pro potřeby SPsim.

2.1.1 Proč grafické rozhraní pro textové CLI

Ačkoliv byl příkazové řádce obecně věnován již nějaký prostor v úvodu práce, je potřeba přesně stanovit co jsou to základní prvky a komponenty, se kterými má výsledný program SPsim pracovat. Je také vhodné zjistit nevýhody textových CLI v porovnání s jejími modernějšími a grafickými revizemi a na základě toho se snažit určit vhodnou pokročilejší formu, ve které má program SPsim toto klasické CLI do jisté míry simulovat.

Dnes běžně využívané příkazové řádky jsou vlastně jiným slovem pro samotný Shell. Shell je program který v počítači vyváří rozhraní pro uživatele. [7] Naproti obecnému přesvědčení a chápání tohoto slova, shell není navenek vždy jen textová příkazová řádka (CLI), které se týká tato práce, ale shell může být i grafický nástroj (například windows explorer anebo obecně všechny windows managery linuxu, například Gnome nebo KDE), nebo hybridní, textové rozhraní. Shell je tedy program který umožňuje uživateli spouštět programy a komunikovat s jádrem systému (volat jeho funkce). Pro přehled uveďme obvyklá rozhraní shellů : 2.1

Příkazový řádek (kterého se týká tato práce) je jednorozměrný předchůdce novějších rozhraní. Má plně textový vstup i výstup, navigace v adresářových strukturách, vstup zadává uživatel plně textově. Shellů je vícero druhů a v unixu jsou nejznámějšími například SH, BASH, KSH, ASH.[8]. Ve MS Windows je to třeba například Powershell.

Plně grafických shellů je mnoho a jelikož jsou všeobecně známé a tato práce se jim nevěnuje, nebudeme je zde rozebírat. Příkladem je například Unity pro systémy linux.

Za zmínku stojí ještě další dva typy prostředí. Jeden se vyvinul přímo z textové příkazové řádky, druhý se k ní zase vrací od plnohodnotných GUI. Příkladem této první verze přechodu příkazové řádky k primitivní grafice je Textové rozhraní (text-based UI). Tato rozhraní částečně kombinovala výhodu textového ovládání, ale dala řádce druhý rozměr, udržovala tedy více-

[7] Etymologie slova Shell (anglicky skořápka) naznačuje jeho funkci. Je to přirovnání k jeho roli v operačním systému, jako „skořápka obalující jádro“. Jádrem se zde myslí kernel operačního systému, a shell je prostředek, který jej obaluje a umožňuje k němu přístup.

[8] bourne shell, bourne again shell, korn shell, almquist shell

	Input	**Output**
Příkazová řádka	Rychlé zadávání povelů	Pomalý příjem informací
Grafické rozhraní	Pomalé zadávání povelů	Rychlý příjem informací
Hybridní rozhraní	Různé	Různé

Tabulka 2.1: Porovnání jednotlivých typů shellu a rozhraní z pohledu interakce s uživatelem. U hybridního záleží na konkrétním provedení.

méně funkci tabulky. Zástupci těchto rozhraní je například dříve populární Norton Commander, ale i v dnešní době se stále využívá jako konfigurační rozhraní při různých systémových instalacích (např. kernelu). Druhým, opačným případem, je moderní hybrid rozhraní sice s podporou ovládání myší, ale je zaměřeno na přijímání textových vstupů a klávesových zkratek od uživatele (Například Xmonad). Výhody a nevýhody těchto rozhraní z pohledu uživatele a jeho interakce může ukázat tabulka 2.1. Je vidět, že textové rozhraní tedy umožňuje rychlé zadávání informací, ale je na něj nutná dobrá znalost slovníku a syntaxe. Následné získávání informací pro uživatele už, v závislosti na výstupu, může být horší. Vzhledem k zadání tedy opravdu vychází grafické rozhraní jako ideální způsob jak reprezentovat data, se kterými jinak pracuje rychlé a robustní, ale na přehlednost nedostačující textové CLI. Tím je ospravedlněn úvodní požadavek na „grafickou reprezentaci“.

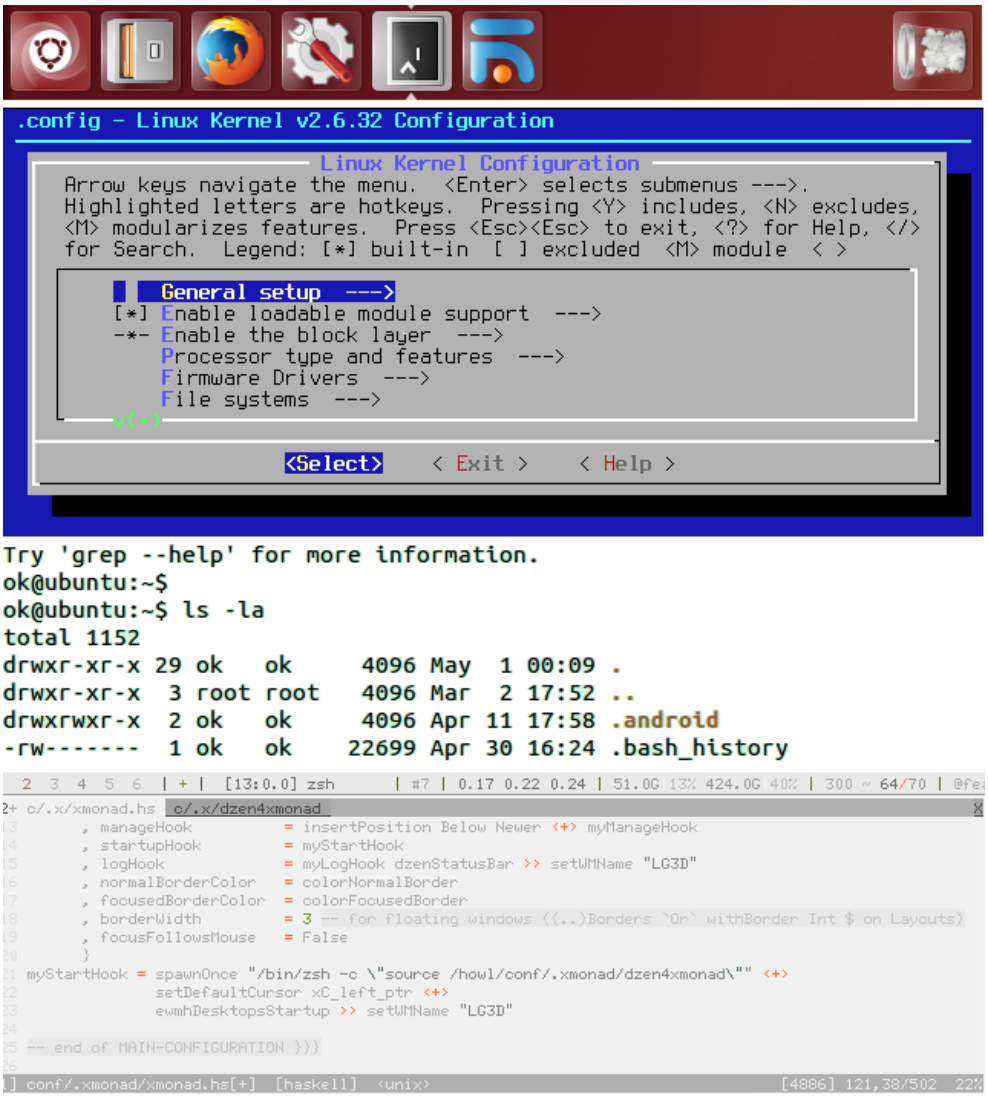
Závěrem této sekce tedy je konstatování, že Grafický simulátor skutečně má smysl v grafické a nikoliv textové či hybridní formě (například rozšířením existující CLI). Nicméně by měl mít nějakou vazbu na formát v kterém je zobrazen v klasické CLI, aby měl uživatel náhled v na rouru a příkazy v původním formátu.

Shrnutím těchto zjištění pro přehlednost do následujících bodů a požadavků na rozhraní SPsim:

- mělo by být plně grafické a využívat obou rozměrů pro přehledné zobrazení více informací najednou
- mělo by umožňovat i náhled na tvar příkazů v původní formě, jako je v CLI

2.1.2 Multiplatformnost

Zadáním práce je „umožněte provádění definované podmnožiny funkcionality běžné příkazové řádky, zaměřte se na příkazy typu filtr“. Toto omezení nepřímo omezuje volbu na operační systémy UNIXového typu. Běžnou příkazovou řádkou zde budeme uvažovat CLI těchto systémů (neboť je nejvíce



Obrázek 2.1: Srovnání čtyř typů shellu - interakce uživatele se systémem. Klasické GUI (Unity), Textové rozhraní (konfigurace kernelu), CLI BASH, Moderní hybrid (Xmodnad)

rozšířená a uživatelům nejvíce povědomá). Část zadání která tvrdí že je třeba zaměřit se na filtry, tento okruh ještě více omezí, neboť taková příkazová řádka musí tyto nástroje samozřejmě podporovat. Řetězení do rour je další funkčnost takové příkazové řádky. Shelly v UNIXových operačních systémech toto podporují. Požadavek na multiplatformnost se tedy sám nabízí, pokud budeme uvažovat UNIXovou rodinu systémů.

Univerzálnost a obecné vlastnosti shellu, příkazových řádek a podporovaných příkazů (mimo mnoha dalšího) napříč UNIXovým světem stanovuje především standard POSIX[9] a SUS[10]. POSIX (a SUS) je okruh standardů, specifikovaných organizací IEEE[11] [1] a vyvíjený a udržovaný především díky Austin Group[12]. Je to kolektivní specifikace nějaké rodiny standardů pro počítačové operační systémy, tedy souhrn předpokladů a funkčnosti, které by měl operační systém plnit, aby se mohl řadit do rodiny UNIXových systémů, a umožňující například přenositelnost programů. To je v tomto případě důležité (zejména sekce POSIX ohledně systémových interface a headerů + Příkazů a utilit), neboť příkazová řádka a unixové shelly vycházejí i z této specifikace. Tento standard by tedy měl umožnit, aby byl program, který bude tuto příkazovou řádku simulovat, do jisté míry multiplatformní. Alespoň dokud bude spouštěn na nějakém operačním systému z UNIXové rodiny.[13] Závěry pro multiplatformnost jsou tedy:

- Grafický simulátor by měl reprezentovat nějaký rozšířený shell kompatibilní s UNIX systémy splňujícími standard POSIX.
- Jazyk a rozhraní, ve kterém bude simulátor vytvořen, musí umožňovat chod programu na těchto systémech

2.1.3 Co sledovat a zachytávat

Uživatel, co do spouštění příkazů v grafické simulaci by měl mít možnosti minimálně stejné, jako má při použití příkazové řádky. Definujeme co spouštění takového příkazu obnáší a co je jeho výsledkem. Použitím nějakého Unix kompatibilního shellu a jeho příkazové řádky by mělo v zadávání příkazů přinést kompatibilitu napříč vícero možnými systémy. Standard dokonce definuje základní podporované příkazy. Ty specifikované SUS standardem jsou dány IEEE specifikací 1003.1-2008 [2] a je jich 161. Příkazy mohou být zabudované už v systému anebo externí (v první případě jsou součástí samotného shellu,

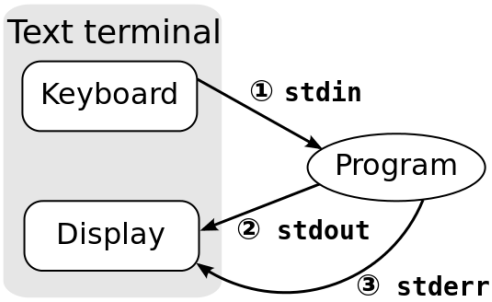
[9] akronym pro „Portable Operating System Interface“

[10] Single Unix Specification

[11] Institute of Electrical and Electronics Engineers (česky „Institut pro elektrotechnické a elektronické inženýrství“)

[12] The Austin Common Standards Revision Group, „a joint working group of IEEE, ISO JTC 1 SC22 and The Open Group“

[13] POSIX-kompatibilní je i MacOS firmy apple a překvapivě dokonce i některé verze Microsoft Windows, zejména v kombinaci s utilitou Cigwin



Obrázek 2.2: 3 datové proudy programu v terminálu (wikipedia.org, volné dílo)

v opačném jsou to zkompilované programy které shell jen využívá). Příkaz je tedy nějaký program nebo povel, který vykonává zadanou funkci. Aby mohl příkaz zpracovávat nějaká data, musí mít nějak definován vstup a výstup. Tento tok dat se v terminologii příkazů shellu nazývá standard stream, neboli standardní datový proud.

Streamy jsou textové (plain text), obsahují lidsky čitelné alfanumerické znaky, a v systému se chápou jako speciální případ textového souboru. Tento „speciální" soubor defaultně asociován s konzolou (tedy klávesnicí a displayem podporujícím zobrazování textu). SPsim tedy může tyto data nějak ukládat a přímo bez dalšího parsování zobrazit uživateli - není tedy nutné je nějak překládat.

Standard stream je souhrnné označení pro kanály datového vstupu a výstupu mezi programem a jeho prostředím ve chvíli spuštění. Jsou to 3 I/O kanály:

1. Standard input (stdin) - standardní vstup
2. Standard output (stdout) - standardní výstup
3. Standard error (stderr) - chybový výstup

To, jak program (v našem uvažovaném příkladě to bude příkaz filtru) zpracovává data do těchto tří kanálů, naznačuje obrázek 2.2.

Pokud uživatel, bez ohledu na roury, používá a snaží se třeba i odladit nějaký program, bude muset využívat tyto tři proudy. „STDIN" a „STDOUT" je standardní vstup a výstup programu, a uživatel o nich musí mít jasný přehled. Chybový výstup „STDERR" nastává teprve ve chvíli, kdy program vykoná nějakou chybu. Uživatel buď zadává vstupní hodnoty příkazu ručně klávesnicí, nebo je směruje z jiného zdroje. Výstupní proudy se mu zobrazí ve chvíli, kdy program svou činnost ukončil. Tyto údaje jsou navíc (pro uživatele shellu nepovinně) doplněny o „návratový kód". Když shell, který v počítači běží jako

program, dostane od uživatele nějaký příkaz (zavolat jiný program), spustí jej jako předek. Jakmile tento vyžádaný program ukončí svou činnost (systémovým voláním exit()), předek je informován a je mu doručen exit status a jeho hodnotu může získat zavoláním systémového požadavku wait(). Shell takto získá informaci o průběhu nebo stavu příkazu, který spouštěl. Návratový kód je číselná hodnota mezi 0 a 255. Nejčastěji se dá setkat s hodnotami 0, která indikuje úspěšné provedení, 127 nenalezení spouštěného příkazu, 2 špatné použití parametrů shellu a příkazu, a 130, kdy bylo vykonávání příkazu násilně ukončeno zkratkou Ctrl+C. Tyto návratové hodnoty ale uživateli shell nedává automaticky, jsou buď součástí chybového výstupu, anebo je uživatel může zjistit voláním speciálního příkazu ($ v kombinaci s ?). Uživatel grafického simulátoru tedy musí mít možnost toto zobrazit.

Příkazová řádka má v tomto ještě jednu nevýhodu. Ve chvíli kdy zpracovává zadaný příkaz neposkytuje uživateli obvykle příliš zpětné vazby. Spouštěný program samozřejmě může mít podporu vypisování průběžných informací (i vyžadování dalších vstupů) od uživatele. Nicméně v případě víceméně jednoúčelových programů a filtrů, které má tato práce využívat, to tak není. Bude - li příkaz muset zpracovávat nějaké větší množství dat, CLI přestane reagovat a „zamrzne“ po dobu jeho běhu. Uživatel má možnost v příkazové řádce násilně provádění takového příkazu ukončit zkratkou Ctrl + C, výsledkem čehož je návratový kód 130. Toto je standardní chování CLI a nemá tedy smysl se ho snažit nějak v grafickém simulátoru obcházet.

Nevýhodou příkazové řádky je, že při vykonávání příkazu uživatel nemá informaci o tom, kolik času jeho provedení zabralo. Grafické rozhraní by mohlo tuto informaci uživateli nabídnout tím, že nebude vycházet jen z dat které mu poskytuje shell, ale bude samo měřit dobu trvání běhu, od spuštění do ukončení.

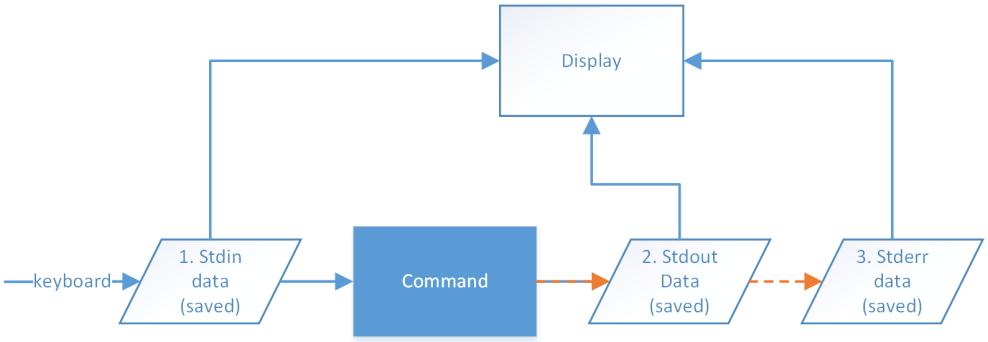
Závěrem tedy můžeme stanovit zjištěné požadavky na grafický simulátoru z pohledu informování uživatele ohledně běhu individuálního příkazu. Tím je podpora stávajících funkcí:

- Zobrazení a ovlivnění standardního vstupu
- Zobrazení standardního výstupu
- Zobrazení chybového výstupu
- Vyvolání a zobrazení návratového kódu

+ funkce navíc

- Měření času trvání průběhu příkazu

Pokud tyto požadavky zobrazíme do grafického náčrtu, dostáváme podobný obrázek jako 2.2, s tím rozdílem že streamy v simulátoru musíme ukládat, aby na ně uživatel mohl mít kdykoliv náhled.2.3.



Obrázek 2.3: Požadavek na datové proudy z pohledu vytvářeného programu

2.1.4 Jaké příkazy podporovat, proč filtry

Jak bylo řečeno v předchozí sekci, standard definovaný POSIX a SUS zajišťuje kompatibilitu mezi podporovanými systémy ohledně řady vlastností a funkcí. Patří mezi ně i základní sada příkazů pro příkazovou řádku [**?**]. Těchto základních příkazů je 161 a patří do sady příkazů GNU coreutils. Slouží k základní manipulaci se soubory, textem, interakci se systémem a shellem samotným.

Zadání této práce volá po zaměření se na filtrační příkazy. Filtrační příkazy (Filtry) umožňují manipulaci s textem (dají se do toho ovšem započítat i ty co hledají, manipulují a porovnávají soubory) a mohou být i velmi komplexní. Některé z nich, jmenovitě třeba AWK, umožňuje svými schopnosti i zastoupení mnoha dalších dostupných příkazů. Cenou za to je samozřejmě o něco větší složitost správného použití a ovládání takového příkazu.

Nabízí se otázka, proč podporovat a simulovat zrovna filtrační příkazy. Odpověď na to je ta, že filtry jsou příkazy, které v mnoha situacích udržují příkazovou řádku naživu v běžném používání. Příkazová řádka se dnes využívá zejména k administraci systémů a počítačů a tam, kde je potřeba vyhledávat, zpracovávat a vyhodnocovat větší množství dat a informací. A právě filtry jsou jedním z důvodů, proč je pro mnohé lidi v takových případech dodnes populární mít otevřených několik textových terminálů namísto jednoho grafického okna, jak se mylně domnívají mnohé filmy. Pomocí několika málo filtrů v příkazové řádce jde rychle a snadno (pokud víte jak) na základě mnoha vlastností a parametrů zpracovat i velké množství informací, a dá se dobře specifikovat jejich výsledná podoba a požadovaný tvar. Proto je to neocenitelný nástroj mnoha administrátorů. Filtry jim umožňují, v kombinaci s příkazy přímo interagujícími se systémem, pomocí minimální námahy a času spravovat a „troubleshootovat“ i velké systémy a sítě. To, že jejich správné použití může být nepříliš snadná věc i pro zkušené experty, je vůbec důvodem pro existenci této práce a výsledného simulačního programu.

1. Sada podporovaných filtrů by tedy měla kvůli univerzálnosti a multiplatformitě vycházet pokud možno z oné množiny 161 základních příkazů.
2. Je vhodné do simulátoru přidat i další příkazy, které se s filtry nejčastěji kombinují

Dle zkušenosti autora a po poradě s vedoucím této práce byl výsledný seznam filtrů a podporovaných příkazů omezen na následující množinu **14.** nejdůležitějších filtrů:

- grep - výpis řádek splňujících omezení
- split - rozdělení vstupu na menší části
- head - výpis prvních částí vstupu
- tail - výpis posledních částí vstupu
- cut - odebrání části z každého řádku
- paste - spojení řádků
- sort - řazení řádků
- uniq - zvýraznění či odstranění opakujících se řádků
- diff - porovnání dvou vstupů po řádcích
- cmp - porovnání dvou vstupů po bytech
- comm - porovnávání dvou seřazených vstupů po řádcích
- tr - přepis nebo mazání znaků
- cat - spojování a označování vstupů
- tee - rozvětvení vstupu do samostatného souboru a zároveň na standardní výstup

Nebyly vybrány příkazy jako je „sed“ a „awk“, zejména díky jejich komplikovanější syntaxi a zápisu. Tyto nástroje mají širší použití, nicméně se vymykají z podobného formátu a syntaxe použití jako mají ostatní filtry. Simulátor samotný a zejména jeho grafické prostředí by pak muselo být zvlášť navíc uzpůsobeno pro práci s těmito nástroji, což by jednak implementačně o hodně složitější, a za druhé by to narušovalo jednotný vhled rozhraní. Není ovšem vyloučeno že tyto nástroje nemohou být zakomponovány v případném dalším vývoji programu.

Program potom musí samozřejmě podporovat i příkaz Pipe „|“ aby se daly příkazy řetězit (viz další sekce).

Simulátor, aby byly uvedené filtry využitelné, by měl poskytovat i sadu dalších příkazů, které se s nimi často používají. Těžko by mělo smysl ladit řadu filtračních příkazů, pokud by neměly data se kterými pracovat. Nicméně příkazů je velké množství, mají různou syntaxi zápisu a nemusí být na všech systémech stejně podporované.

Přesto je nutné pro použitelnost filtrů přidat další funkce z CLI. Ideálním řešením pro to je virtuální, „vlastní“ CUSTOM příkaz 2.4. Custom příkaz je abstraktní vizualizace vlastního uživatelova vstupu. Jinými slovy, uživatel si bude moci v simulátoru spouštět jakýkoliv vlastní příkaz či skript, který se bude vůči zvoleným filtračním nástrojům tvářit, jako by to byl jen další příkaz se vstupem a výstupem. Je na uživateli, aby takovýto vstup či jiný volaný příkaz byl na daném systému funkční. Simulátor tedy bude přímo odladovat a skládat zadanou podmnožinu filtrů, ale zároveň neochudí uživatele a nesníží použitelnost takového simulátoru, neboť si tam na vlastní nebezpečí bude moci zadávat zcela vlastní příkazy stejnou formou, jako by to dělal v příkazové řádce. Simulátor je pak jen abstraktně zobrazí jako „virtuální příkaz“, pro který bude navíc podporovat všechny ostatní funkce simulátoru a ladění (tedy kontrolu vstupu a výstupu, selektivní spouštění, měření času provedení a podobně).

Z tohoto důvodu je ještě vhodné přímo do simulátoru přidat ještě jeden příkaz:

Xargs

Existují totiž příkazy v CLI, které nepřijímají standardní vstupní stream, ale pouze vlastní parametry. Xargs je nástroj, který toto pomáhá obejít. Xargs totiž bere svůj standardní vstup, a ve svém těle onen zvolený příkaz, kterému jako parametr předává právě svůj standardní vstup. Funguje tedy jako wrapper, a umožňuje aby se do roury zapojovaly i příkazy které k tomu nemají předpoklady. Požadavky na program z pohledu podporovaných příkazů na výsledný simulátor je tedy shrnut do následujících bodů:

- Detailní podpora dostatečně široké a pevně zadané kompatibilní sady filtrů
- Umožnění testování vlastních příkazů a skriptů pomoci virtuálního příkazu
- Umožnění zapojení do roury i vlastním příkazům, které nepřijímají stdin z roury (Xargs)

Kombinace těchto tří možností ve výsledném programu velmi rozšíří využitelnost takového simulátoru. Nabídne požadované usnadnění tvorby problematických filtrů, a přitom bez nároků na velký nárůst složitosti programu umožní přímo v simulátoru jejich interakci s velkou škálou vlastních vstupů a příkazů.

Obrázek 2.4: virtuální vlastní příkaz v pipeline, simulující libovolný jiný příkaz

2.1.5 Roury, fronty a redirect

Samostatné příkazy samy o sobě umožňují mnohé, ale je nutné je často používat v kombinaci s ostatními. Jedno možné řešení je ukládat si výstupy příkazů do souboru, a z nich poté číst vstup pro další zpracování. Toto řešení, jakkoliv zcela validní a funkční, je velmi nepraktické. Uživatel musí ztrácet zbytečné množství času a pozornosti jen meziukládáním dat. Ladění potom takové sekvence samostatně spouštěných příkazů je časově náročná záležitost. Proto byl vynalezen příkaz zvaný Pipe (Roura). Roura se zadává jako jediný znak „|“ a umožňuje svázat více procesů (příkazů) za sebou skrze jejich standardní streamy, stdin a stdout. V takovém případě je výstup příkazu (stdout) rovnou nasměrován přímo do vstupu (stdin) následujícího příkazu napravo od něj. Příkazy o tomto přesměrování neví, vstup a výstup je z jejich pohledu stejný. Pokud některý příkaz v rouře zahlásí chybu a vypíše něco na chybový stream (stderr), roura tuto informaci dál nepřenáší. Toto chování je defaultní, ale dá se ovlivnit. Způsob jakým je toho docíleno se liší podle typu shellu, například CSH využívá pro to verzi Pipe ve tvaru "|&", Bourne shell a jeho dnes nejoblíbenější nástupce BASH zase "2>&1".

Z pohledu systému je Roura implementována jako fronta (datová struktura). Jsou s ní spojeny dva popisovače otevřených souborů (jeden pro zápis a

jeden pro čtení). Její takto těsné spojení se souborovým systémem jí umožňuje pracovat s ní pomocí souborových operací. Důležité je, že uživatel do dat v rouře nijak nezasahuje. Funguje pouze jako nástroj přeposílání dat.

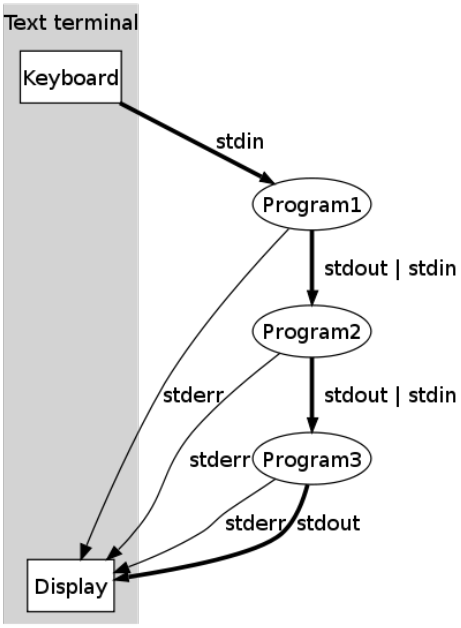
Tento fakt tedy může být v programu využit podobným způsobem. Jelikož půjde o reprezentaci a grafickou „nadstavbu“, roura ve výsledném programu nemusí vůbec nic spouštět. Jednoduše to bude prvek který jen bude ukládat výsledky získané z předchozího příkazu s možností posílat je rovnou na vstup a spouštěč dalšího. Nicméně zde si můžeme dovolit zjednodušení oproti unixovému řešení. Roura vlastně nemusí předávat výsledky následujícím blokům (reprezentantům příkazů CLI), ale tyto bloky se po zavolání mohou domáhat dat z předchozí roury samy (bude - li mít data přístupná a ve vhodném formátu). Výhoda tohoto řešení je, že není potřeba složitý scheduler a buffery. Přijde-li řada na blok v simulátoru, blok si sám zjistí zda má k dispozici data. Blok s příkazem ve výsledném programu tedy bude muset vykonávat trochu více práce, ale o to jednodušší bude celková synchronizace. Jelikož jde o reprezentaci roury, není nutné v tomto případě zcela kopírovat její funkci, stačí že se bude simulovat její chování a funkce navenek. Nijak to zde neovlivní funkčnost z pohledu uživatele. Navíc z pohledu funkce SPsim jako debuggeru to umožní náhled na data v dané části roury kdykoliv, bez ohledu na to v jaké fázi zpracovávání příkazů je.

Fronty, někdy také Kolony [14] vznikají tak, že se pomocí operátoru Roura [15] zřetězí více příkazů za sebou. Jejich standardní vstupy a výstupy se propojí do jednoho celku, který se z pohledu uživatele spouští najednou. Jde vlastně o přesměrování vstupních a výstupních proudů dat z uživatelovy konzole na jiný program (příkaz). Fronty jsou elegantní způsob jak stavět složitější skripty. Náhled na to jak funguje fronta poskytuje obrázek 2.5.

Fronta pak funguje tak (alespoň ve většině systémů na bázi UNIXu), že se všechny procesy ve frontě spustí ve stejný čas, jejich datové streamy se správně správně propojí, a celek je pak řízen systémovým schedulerem. UNIXové fronty mají od ostatních implementací navíc jednu výhodu. Je to způsob jakým ve frontách funguje buffering. Programy ve frontě mohou fungovat různou rychlostí. Posílající program může například produkovat výstup v kilobajtech za vteřinu, ale přijímající program, který je na řadě dovede svým vstupním proudem přijímat jenom v řádech stovek bytů. Nicméně díky UNIXovému řešení to nevadí - o data se nepřijde, shell řadí data v rouře za výstupem vysílajícího programu do fronty (anglicky queue, myšleno z pohledu datové struktury). Ve chvíli kdy je přijímající program připraven načíst data, operační systém mu přepošle obsah fronty a jí uvolní. V případě že se přijímací fronta zaplní, odesí-

[14] Anglicky pipeline

[15] zde vzniká v češtině poněkud matoucí nejasnost, angličtina odlišuje Pipe, tedy prvek roury, od Pipeline. České Roura označující příkaz roura se někdy vzájemně zaměňuje s tím, čemu můžeme říkat „fronta“ nebo „kolona“, tedy soustavou rour spojujících příkazy. V tomto textu se Roura používá pro příkaz Pipe, Fronta pro jejich zřetězenou soustavu.

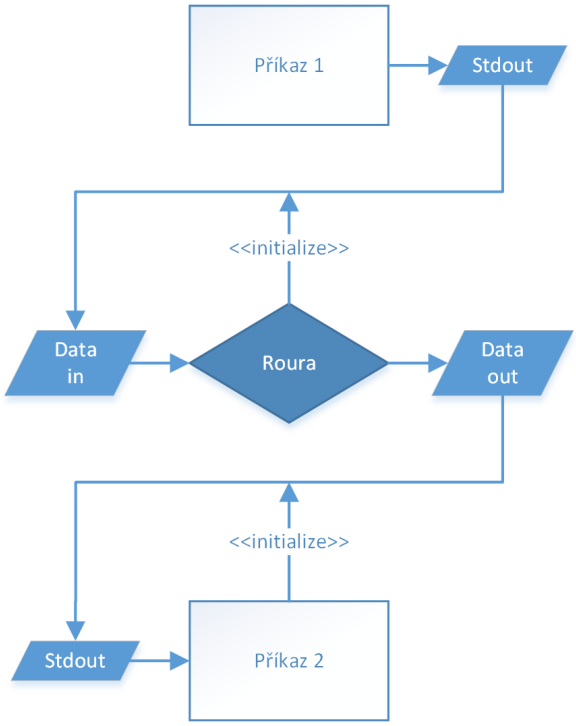


Obrázek 2.5: Schéma fronty tvořené rourami v UNIXu (pipeline) (wikipedia.org, volné dílo)

lající program je pozastaven až do doby, kdy se fronta opět uvolní [16]. Naštěstí, SPsim má fungovat i jako ladící nástroj. Ve zpracování fronty musí zároveň fungovat nějaké meziukládání dat pro náhled (i zpětný) uživatele. Dávkovací frontu tedy mezi jednotlivými příkazy v simulaci nahradí zápis do nějaké datové struktury která udrží data i po ukončení příkazu - a ta bude sloužit k načítání a zapisování, stejně jako je to v případě každého jiného simulovaného příkazu, jako na obrázku 2.3.

Fronta je tedy důležitým aspektem pohodlné práce s příkazy, a v našem případě příkazy filtrů. Fronta a roura je tedy alfou a omegou samotného simulátoru (nakonec i proto se jmenuje SPsim), a je tedy potřeba jí v simulátoru nějakým způsobem vhodně napodobit bez toho, aby to změnilo výslednou funkci. Vzhledem k tomu, že roura je z principu nějaké úpravy dat zcela nečinná (pouze předává data), není vlastně ani nutně pro potřeby simulace nutná. Simulátor to bude nejlépe řešit, tak že funkci roury částečně nahradí objekt, který jen umí uskladnit a zpřístupnit vstupní data. Ve chvíli kdy krok zpracování fronty dojde na rouru, roura sama inicializuje nahrání dat z výstupu předchozího příkazu.

[16] ve většině UNIXových systémů má tento vyrovnávací buffer velikost 65536 bajtů



Obrázek 2.6: upravené provedení v SPsim

Nabízí se možnost, že roura by automaticky ihned odeslala svá data do vstupu (což bude také objekt umožňující uskladnění a náhled na data) dalšího příkazu. Kvůli univerzálnosti, a kvůli tomu že na Rouru půjde v programu nahlížet jako na plnohodnotný blok (který ovšem nebude nic dělat než kopírovat ze vstupu na výstup), je tato možnost horší, než když si další blok procesu při spuštění načte data sám. Teoreticky to tedy umožní programu se vlastně obejít při tvorbě fronty zcela bez Rour, neboť roura data nijak neovlivňuje a příkazy se tedy budou moci i řetězit samy za sebou. Tato možnost tedy z pohledu celého simulátoru a debuggeru bude vhodnější. Ve výsledku se tak za simulátor Fronty schovává původní způsob nepraktického meziukládání dat. Je to ale v pořádku, neboť to plně slouží požadavkům takového grafického a krokovacího interpretu a zpětné revize vstupů a výstupů.

Souhrn poznatků z této sekce se dá shrnout do následujících zjištění:

- Program musí ukládat streamy mezi Rourou a Příkazy
- Program (a Roura) standardní vstup sama načítá od předchozího příkazu a výstup si sama meziukládá.
- Data v těchto bufferech zůstávají tak dlouho, dokud je něco nepřepíše.

2.1.5.1 Přesměrovávání proudů ze souboru

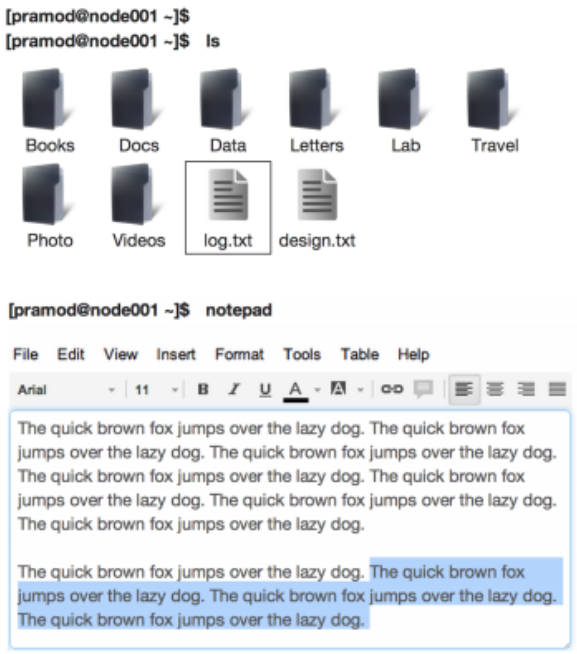
Ve výše uvedených případech a náhledu na problém se uvažuje, že proud dat vždy pochází buď jako vstup od uživatele z klávesnice, anebo přímo z výstupního proudu příkazu před ním spojeným pomocí roury (ve výsledném programu toto bude zprostředkované ukládající mezivrstvou). Toto přesměrování ale není omezeno jen na tyto dvě možnosti, vstupní proud může přicházet z více zdrojů. Častým použitím je přímé načítání dat ze souboru na disku. Některé příkazy toto inicializují přepínačem „-f“, jindy berou za vstupní soubor defaultně svůj parametr (nebo více parametrů). Výsledný program toto může umožňovat v případě přepínače „-f“ automaticky žádostí o vhodný parametr s názvem souboru. Ve druhém případě příkaz takto interpretuje data ve svých parametrech automaticky - budou-li vyplněna, příkaz bude načítat vstup z uvedeného souboru. Nebudou-li vyplněna, příkaz příjme vstup standardním způsobem. Případně může uživatel tuto funkci i obejít tak, že využije příkaz který tato data dovede načíst před ním (například cat) a rourou mu je pak předá. Je potom na uživateli jaký způsob si vybere, i když poslední jmenovaný je preferovaný kvůli lepšímu náhledu na tok dat v simulované rouře.

Stejným způsobem lze v příkazové řádce přesměrovávat i výstup (místo na konzoli či do následujícího příkazu například do tiskárny a podobně). Tato možnost nebude simulátorem přímo podporována. Jednak pro to že uživatel bude mít automaticky z designu možnost rovnou kontrolovat vstupní a výstupní data jakéhokoliv příkazu (a to i zpětně) na jakémkoliv místě Fronty, za druhé k tomuto může sloužit vlastní příkaz umožňující takové chování (CUSTOM s vyplněným ⟩, případně ⟩⟩). K dispozici je navíc implicitně i příkaz Tee, který toto umožní i uprostřed fronty. Možnost ovlivnění vstupů a výstupů bude tak ve výsledném programu dostatečná.

2.2 Rešerše existujících řešení

Pokud by člověk hledal nějaké existující programy, které mají ve svém popisu „Grafická příkazová řádka“, narazí obvykle na hybridní shelly a správce oken, zmíněné v sekci „Proč grafické rozhraní pro textové CLI“. Tyto programy se snaží nějak zpříjemnit klasickou příkazovou řádku, nicméně obvykle jenom tak, že jí integrují do klasického grafického prostředí oken a rozšíří její funkčnost o více klávesových zkratek. Nicméně zvláštní zmínku si zaslouží zajímavý systém Gracoli.[3] [17] a Rapidminer [18]

[17] http://www.gracoli.com/
[18] https://rapidminer.com/



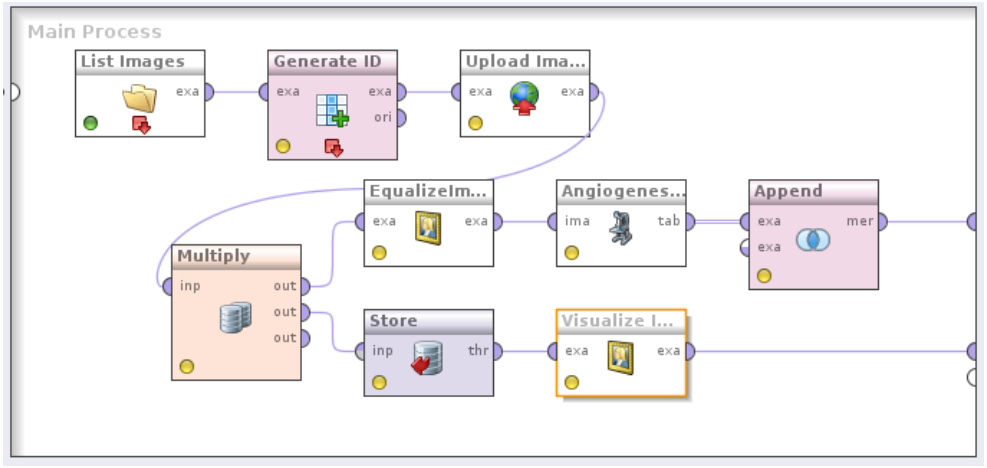
Obrázek 2.7: Prostředí Gracoli (obrázky z publikované práce autora; Pramod K. Verma, MD, Johns Hopkins University, 2013, www.cs.jhu.edu/ pramod. Publikace na http://pramodverma.info/gracoli/gracoli.pdf (k 5.2015))

2.2.1 Gracoli

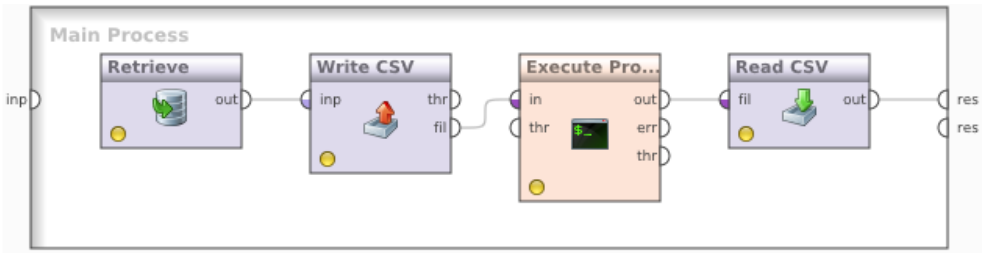
Gracoli skutečně rozšiřuje příkazovou řádku a graficky jí obohacuje, a to velmi elegantním způsobem. 2.7. Z důvodu nedostupnosti aplikace ke stažení je ale nemožné otestovat, zda toto rozšíření bude zahrnuto i do spouštění příkazů v rourách, anebo alespoň dávkově. Je možné tedy tento program považovat za inspiraci ke ztvárnění grafické řádky, nikoliv ale jako náhradu požadované reprezentace CLI a simulátoru umožňujícího stavbu rour a ladění.

2.2.2 RapidMiner

Důležitou původní inspirací SPsim je univerzální nástroj Rapidminer. RapidMiner je softwarová platforma, která přehledně a plně graficky umožňuje strojové učení, vytěžování dat, prediktivní analýzu a business analýzu. Je to velmi komplexní a rozšiřitelný nástroj který umožňuje jednoduché modelování právě pomocí přehledné grafické a interaktivní prezentace. V tomto splňuje stejné cíle jaké si klade za úkol simulátor SPsim, ačkoliv má mnohem větší možnosti. Grafické rozhraní RapidMineru je ukázkou takového vhodného využití debugovacích a pipelinových funkcí, neboť uživatel přehledně vidí jednotlivé funkční bloky kódu, reprezentované grafickými objekty, a komunikaci mezi nimi zpro-



Obrázek 2.8: Grafické rozhraní programu RapidMiner (dokumentace RapidMiner)



Obrázek 2.9: Spouštění a simulace roury s příkazy pro CLI (dokumentace RapidMiner)

středkovávají linky simulující komunikační kanály, umožňujícími rozbočování a slučování a další vyhodnocování. 2.8

Co se týče funkcionality simulování příkazové řádky jako takové a možností stavění rour, tato možnost v RapidMineru existuje, nicméně není zabudována přímá podpora a rozklikávací menu reprezentující jednotlivé příkazy. Uživatel má možnost tak stavět a testovat roury z obecnějšího pohledu a procesy musí plně definovat ručně. 2.9 Využití tohoto také zcela znehodnocuje z našich požadavků fakt, že je podporována pouze příkazová řádka MS. Windows

Závěry rešerše se tedy dají shrnout následovně:

- Nepodařilo se nalézt existující simulátor reprezentující vysloveně příkazovou řádku a roury.
- Uvedené programy mohou sloužit jako inspirace (zejména pipeline RapidMineru).

2.3 Různé přístupy k ladění

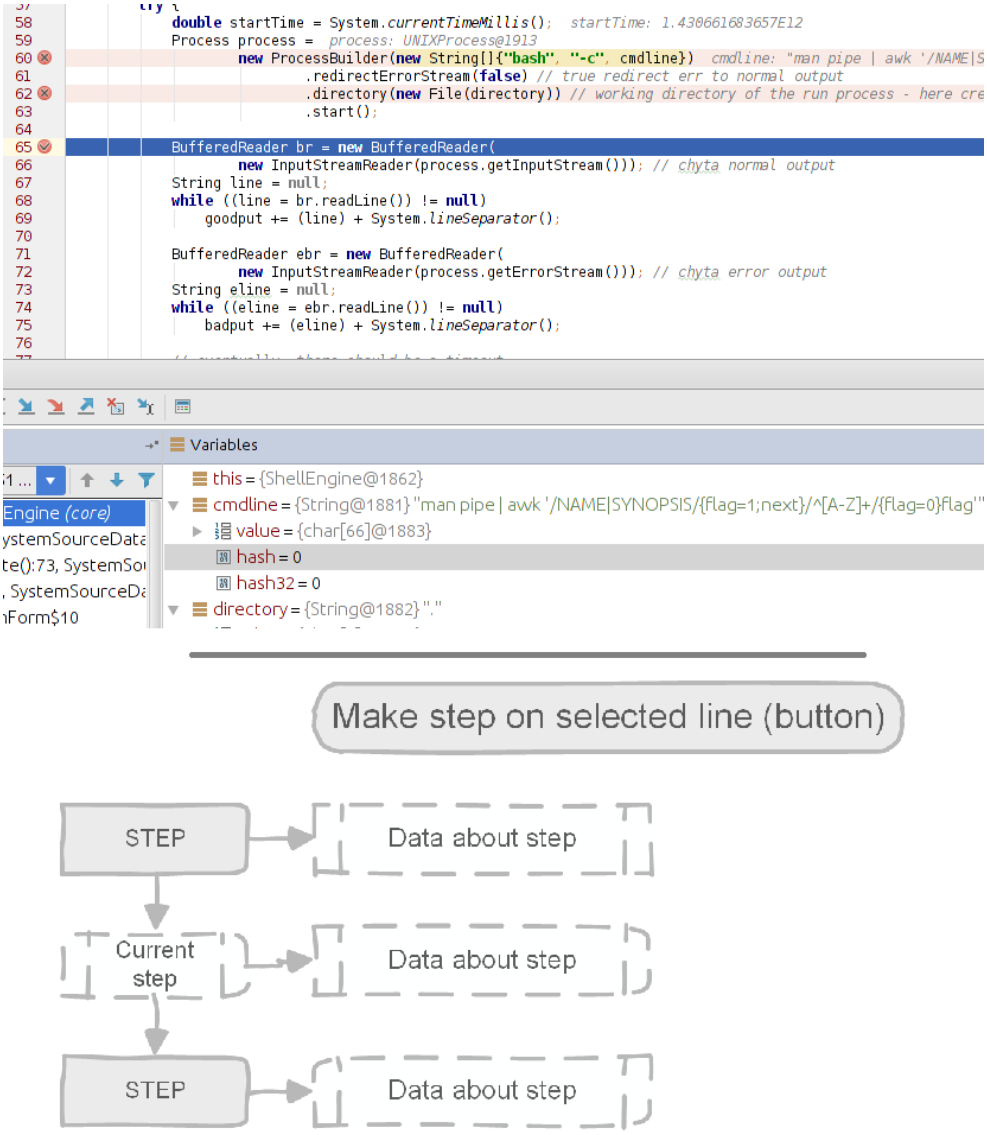
V předchozích bodech této kapitoly se nastínily požadavky na výsledný software. Bylo ospravedlněno zadání práce na grafickou reprezentaci příkazové řádky, stanovena podmnožina podporovaných funkcí a příkazů (a tím i omezení). Je také navržené jak by měl výsledný program pracovat s daty a tvořit roury a fronty, včetně toho jak mají být simulovány jejich datové toky. Rozšíření tohoto reprezentujícího simulátoru je vlastně funkce ladění, tedy debuggingu. Tato vlastnost vychází z navrhované schopnosti simulátoru spouštět jednotlivé příkazy roury zvlášť, postupně, nezávisle na sobě, a přitom zaznamenávat statistiky jejich provedení.

Jednotkou ladění v našem případě bude KROK. Krok bude symbolizovat proběhnutí jednoho prvku příkazové řádky, a tím je spuštění příkazu. Součástí tohoto kroku je tedy i získání dat ze vstupu a zapsání dat na výstup. Tento krok se může libovolně opakovat a spouštět nezávisle na sobě, bez ohledu na místo ve Frontě. Před a po každém kroku musí mít uživatel náhled na příkaz se vstupy a výstupy předtím, než se provede, a poté co se provede, i se všemi zaznamenanými statistikami a chybovými hlášeními.

Ladění a pohled na zpracovávaná data může mít více forem, a je potřeba vybrat tu, pro tento případ, nejvhodnější. Příkladem typického a uživatelsky přívětivého grafického ladícího nástroje je RapidMiner z předchozí sekce. Nicméně způsobů ladění (debuggingu) v grafické či částečně grafické formě existuje vícero. Pokusíme se probrat dva nejčastější přístupy, a probrat jejich pro a proti. Bude to tedy oficiální zhodnocení toho, jakého formátu by se měl SPsim dodržet.

Prvním z těchto debuggovacích přístupů je přístup, který je známý každému vývojáři, a tím je inteligentní debugger IDE.2.10. Jedná se skutečně o grafickou reprezentaci - jenom je jejím hlavním prvkem text. V případě našeho programu by to znamenalo, že uživatel by zadával data, interagoval a sledoval reprezentaci CLI ve skutečně grafickém prostředí, nicméně debugovací informace o obsahu rour by mu byly zobrazeny podobnou formou jako je na obrázku 2.10

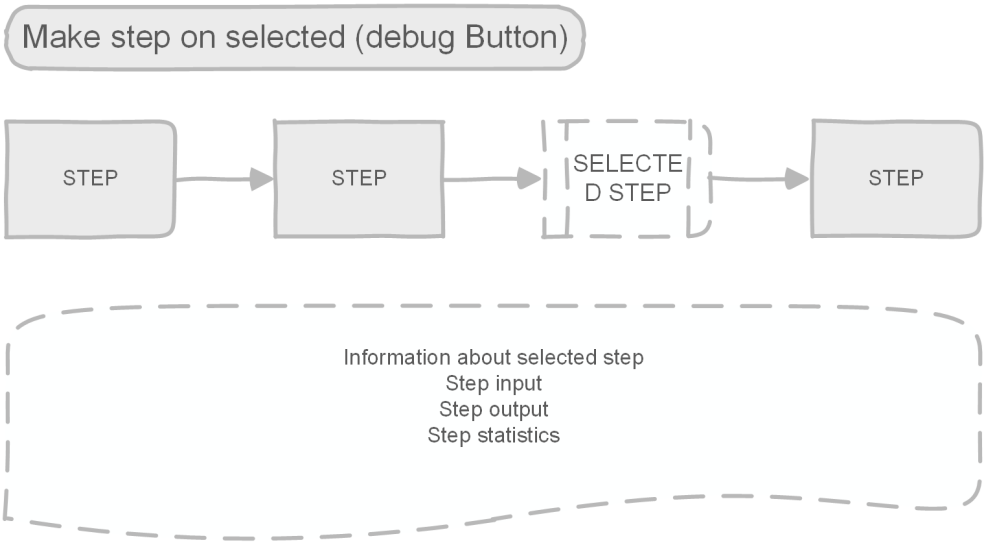
1. Výhody řešení v podobě řádkového debuggeru (po vzoru IDE):
    - Strukturální zobrazení bližší skutečné pipeline (řádky, kroky mezi nimi)
    - Přehlednost i dlouhých front a rychlejší přestupování mezi informacemi, vyšší hustota dat
2. Nevýhody:



Obrázek 2.10: Ladění a náhled na data v řádkové reprezentaci kódu po jednotlivých krocích (po vzoru IDE) s konceptuálním nákresem

- Nutnost zvláštního grafického rozhraní odlišného od grafické reprezentace řádky
- Méně přívětivé pro méně zkušené uživatele

Druhý způsob ladění je plně grafický, zakomponovaný přímo do grafického rozhraní programu, kde uživatel příkazy vytváří a organizuje do rour a fronty. Podobné rozhraní má například uvedený program RapidMiner, kde informace



Obrázek 2.11: Informace o průběhu procesů a statistiky o nich jsou k dispozici přímo v hlavním rozhraní programu

o provedených krocích jsou uživateli rovnou zobrazovány nějakou vhodnou formou (grafickým interaktivním prvkem či textovým výpisem v tom samém okně). Tento zcela grafický model, zjednodušen na konceptuální schéma, je znázorněno na obrázku 2.11.

1. Výhody řešení v podobě grafického debuggeru (podobně jako RapidMiner):
   - Není potřeba zvláštní grafické rozhraní
   - Pro zobrazení statistických a vstupně-výstupních dat dá využít existujících komponent
   - Uživatel může krokovat přímo mezi příkazy přímo ve tvaru, v jakém je zadal
2. Nevýhody:
   - Informace o kroku dostupná jen pro momentálně krokovaný nebo vybraný prvek
   - Pro dohledání debugovacích informací vlastností o jiném prvku musí uživatel na něj přepnout
   - Horší možnost okamžitého porovnávání mezi jednotlivými procesy

Vzhledem k výše uvedeným pro a proti bude optimální zvolit krokovací způsob uvedený ve druhém případě. Jednak tento způsob poskytuje jednodušší rozhraní, za druhé lépe odpovídá požadavku a smyslu této práce - zjednodušit

tvorbu příkazů a rour. Komplikování robustnějším IDE-podobném debugovacím „řádkovém“ prostředí by se tato jednoduchost znehodnocovala. Závěry této sekce jsou tedy následující:

- Jednotkou ladění a simulace příkazů bude jeden KROK
- KROK odpovídá spuštění jednoho prvku z reprezentované řádky (příkazu, roury..)
- Zobrazení informací ohledně provedeného kroku se bude odehrávat ve stejném prostředí, v jakém uživatel tvoří a zadává příkazy
- Zobrazení těchto informací se bude měnit v závislosti na právě vybraném příkazu

2.4 Požadavky na GUI a volba technologie

V této sekci se je nutné provést analýzu toho, jak bude výsledný program vypadat. Je potřeba především určit rozvržení. Rozvržení bude zcela grafické (díky závěrům předchozí sekce), je ovšem vhodné ustanovit jakou má orientaci, tvar, a jakým způsobem se má řešit vizualizace jejich komponent a vazeb mezi nimi.

2.4.1 Volba rozvržení

Jelikož rozhraní bude grafické a bude se skládat z bloků reprezentujících prvky příkazové řádky, nabízejí se tři možnosti hlavního rozvržení. Svislé, Horizontální, a Plovoucí.

Svislé uspořádání bloky příkazů řadí pod sebe, od shora dolů. Bloky s příkazem a nastavením se přirozeně roztahují do šíře a zachovávají si menší výšku - to je žádoucí z pohledu uživatele, aby se mu jich více vešlo najednou do jednoho okna. Prostor pro zobrazení informací a ovládacích prvků je pak na pravé straně, vespod či nahoře (kde tím zabírá prostor blokům), případně nalevo od bloků. Větvení je možné omezeně do pravé strany.

Horizontální uspořádání bloky příkazů řadí vedle sebe. V našem prostředí je přirozenější pokud je to zleva doprava, tedy stejně jako je to v CLI. Bloky symbolizující příkaz jsou užší a vyšší, parametry jsou umístěné v nich pod sebou. Prostor pro zobrazení informací a ovládacích prvků je nahoře a dole, případně vlevo a vpravo od bloků (čímž je jim ubírán prostor místo). Větvení je možné omezeně spodním směrem.

Plovoucí uspořádání je podobné jako v programu RapidMiner. Je ztvárněno vymezeným obdélníkovým pracovním prostorem, kam uživatel zadává

funkční bloky příkazů v podobě plovoucích obdélníků. Pořadí bloků za sebou je určeno jejich propojením a pouze pomocí jednosměrných linek (případně číslování, ale to ztěžuje orientaci uživatele). Bloky musí mít menší velikost, tedy i méně vlastního prostoru pro nastavení operátorů a patternů. Větvení je možné i složitější

Shrnuté výhody a nevýhody jednotlivých rozvržení s náhledem:

1. Blokové uspořádání svislé 2.12
   - V: Na většině monitorů více prostoru pro informační panely a ovládací prvky, operátory řazeny pohledově vedle sebe jako v CLI
   - N: Méně prostoru pro bloky, rolovací nabídky operátorů zasahují do dalších bloků, méně častý a méně intuitivní
2. Blokové uspořádání vodorovné 2.13
   - V: Více prostoru na většině monitorů (orientace na šířku), příkazy zadávány ve vizuálně podobné formě jako v CLI, operátory nikam nezasahují, uspořádání intuitivnější, běžnější, ovládací prvky umístěné nahoře a dole jsou přirozenější pro „přehrávání“ (kroky ladění)
   - N: Možné větení by bylo málo přehledné a plýtvalo prostorem
3. Blokové uspořádání plovoucí 2.14
   - V: Libovolné zobrazení bloků nezávisle na pořadí, ovládací a informační prvky možné umístit na libovolné straně
   - N: Více plýtvání prostorem, mnoho spojovacích linek, horší organizace příkazů, méně prostoru pro blok příkazu (a méně nastavitelných věci přímo v něm)

2.4.1.1 Větvení a kanály spojnic

Ještě dříve než se zváží tyto uvedené vlastnosti pro výběr finálního layoutu, je potřeba rozhodnout ještě jeden požadavek. Tím je podpora větvení. Větvení se dá představit v programu jako napojení dalšího řetězce rour k nějakému bloku s příkazem, ze kterého roura již vychází. Toto by umožnilo porovnávat více výstupů najednou ze stejného vstupu. Vzhledem k tomu, že uživatel má neustále kontrolu (bez ohledu na layout) nad všemi datovými toky ve Frontě, není tato možnost pro funkci klíčová. Uživatel totiž může okamžitě individuálně nastavit daný problémový prvek jinak. Takové řešení je pak přehlednější. Navíc je tato možnost omezená prakticky jen na jeden layout - plovoucí. Ve vertikálním a horizontálním by plýtvalo místem, a i tak by to nebylo příliš přehledné (zejména u vícenásobného větvení.

Další věcí, kterou je pro rozvržení nutné zvážit, je samotná existence spojovacích linek mezi jednotlivými bloky příkazů. Jelikož bylo v předchozích sekcích rozhodnuto, že roury nebudou mít žádnou zvláštní samostatnou funkci a budou fungovat jako „mrtvý“ příkaz který jen udržuje data, jejich existence, pokud pomineme plovoucí layout, ani nemusí být nutná. Blok Roury bude tak jako tak prezentován samostatným obdélníkem (a nakonec díky dřívějším závěrům navržené architektuře bloku příkazu ani nebude pro funkčnost simulace vynucován). Jelikož jsme odmítnutím větvení udělali největší výhodu plovoucího layoutu nepotřebnou a ve svislém či horizontálním rozvržení spojovací grafické linky nejsou potřeba (příkazy tvořené bloky jsou řazeny hned za sebou v jednom směru), můžeme tedy okruh vhodných layoutů zúžit na následující omezující podmínky:

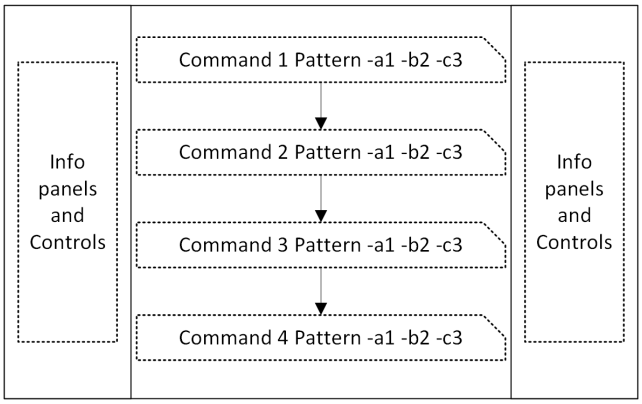
- Větvení nebude podporováno
- Linky nebudou potřebné
- Layout by měl být uživateli co nejpovědomější

Na základě těchto charakteristik můžeme vyřadit Plovoucí rozvržení a zbývá na výběr buď svislé nebo vodorovné. Jelikož vodorovné zobrazení lépe využívá prostoru dnešních širších monitorů a protože bude většinu uživatelů povědomější, volí se pro GUI programu tedy

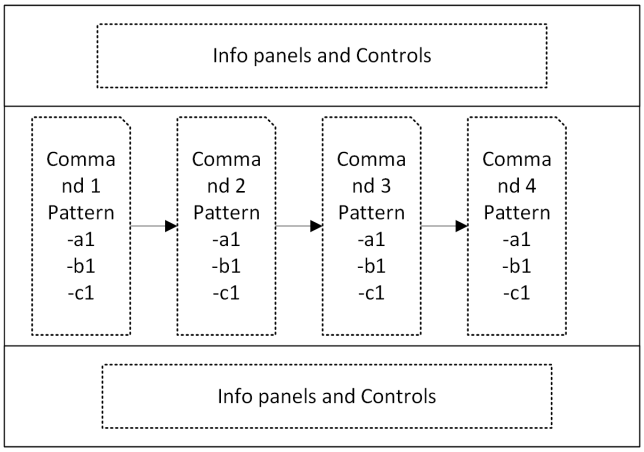
Vodorovné (horizontální) rozvržení.

2.4.2 Wireframe model

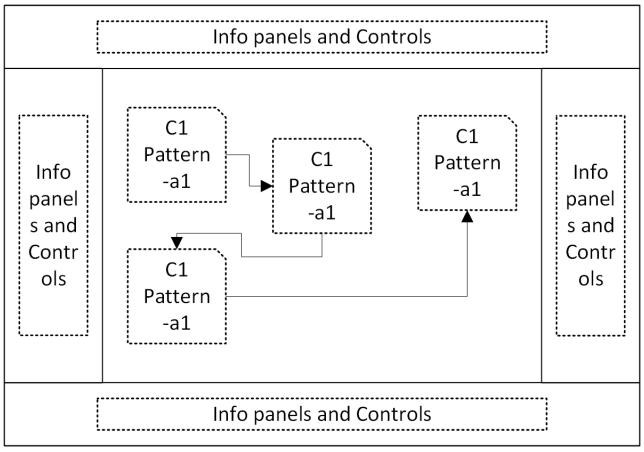
Na základě předchozích závěrů je v tuto chvíli už dost informací, aby se stanovilo jak bude rozvržen a vypadat GUI model.2.15 Bude mít horizontální rozvržení, kontrolní bloky budou hlavně nahoře a dole (z důvodu úspory místa, jinak by si s bloky reprezentující příkazy konkuroval a spodní prostor by zůstával nevyužitý). Jelikož by měl být seznam dostupných příkazů na očích a nikoliv schován (při stavění roury tak uživatel bude mít lepší přehled o nástrojích), a jelikož seznam je v použitelné podobě vertikální, bude ideální pokud se bude nacházet vedle bloků s příkazy. Nejlepší volba místa pro tento list bude vpravo, neboť zleva tak nebude začínající roura opticky ničím přerušována.
V horní a spodní části nad prostorem pro bloky příkazů budou ovládací prvky příkazů (a debugování obecně) - tedy zejména krok a nějaké podpůrné funkce (automatický krok, reset..). V předchozích závěrech bylo řečeno že tento model by měl umožňovat uživateli náhled na to, jak by jeho sestava příkazů vypadala v mateřské příkazové řádce systému. Je to tedy dlouhý a úzký textový panel. Umístí se pod místo s bloky, neboť nad ním by interferoval s ovládacími prvky. Pod tímto blokem se pak nachází tři hlavní panely. Jelikož je z



Obrázek 2.12: Možnost vertikálního rozvržení



Obrázek 2.13: Možnost horizontálního rozvržení



Obrázek 2.14: Možnost plovoucího rozvržení

pohledu samotného spouštění dobrý náhled na vstupní a výstupní data, budou tyto obsažena ve velkých textových polích spodní části obrazovky. Vlevo vstup, vpravo výstup (respektující směr roury). Tyto panely budou ukazovat relevantní informaci vůči právě vybranému bloku (příkazu). Vybrání bloku se udělá kliknutím myší na něj.
Pole vstupu bude mít možnost ručně přepsat vstupní data (tudíž simulovat přesměrování vstupu bloku na data přímo od uživatele). Výstupní pole obsahuje stdout a stderr. Vzhledem k tomu že vždy nastává jenom jedna z těchto hodnot, je pole sloučené (jinak by vždy jeden prázdný výpis zbytečně zabíral místo). Přepínat mezi těmito vstupy půjde pomocí tlačítka nebo oušek záložek.
Prostřední panel mezi vstupem a výstupem bude zobrazovat název momentálně vybraného bloku 2.16 (v případě že uživatel bude vybírat operátor tak operátoru), statistiky jeho provedení (čas, návratovou hodnotu) a zejména *relevantní nápovědu* vůči právě aktivnímu prvku. V případě celého bloku to bude relevantní nápověda a vlastnosti příkazu který zastupuje, v případě operátoru zase informace o jeho funkci a hodnotách, které přijímají jeho patterny (má-li jaké).

Aplikace ještě bude mít nahoře klasické menu, kde budou složitější konfigurační nastavení a informace o samotném programu a autorovi (známé „about“).

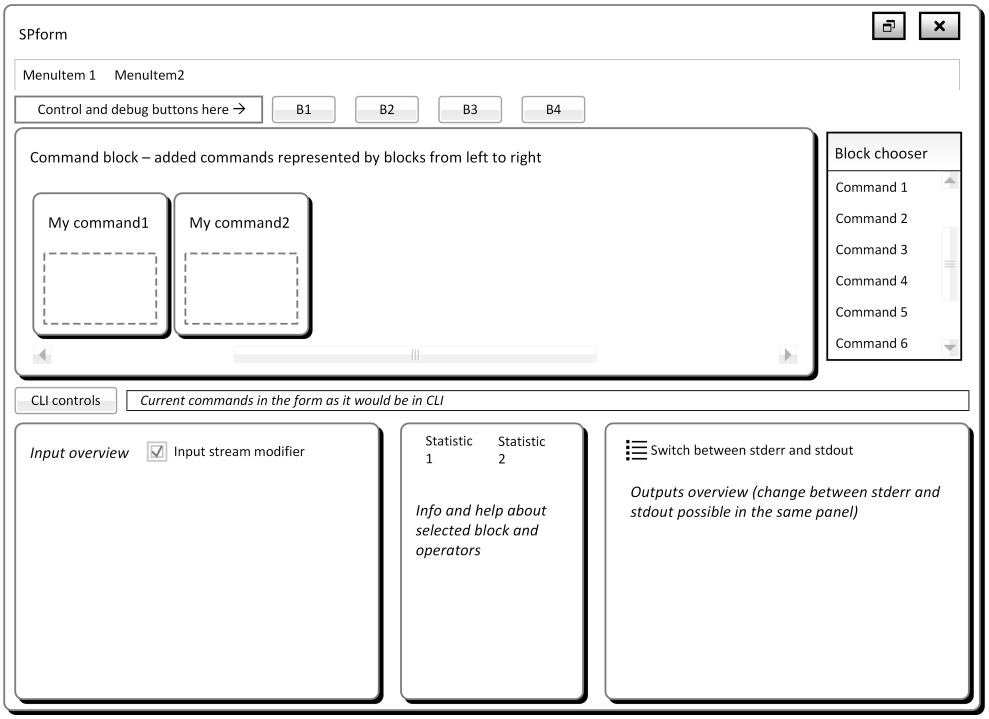
2.5 Emulace vs. Nadstavba, V čem spouštět aplikaci

Tato sekce je zaměřená na platformu, na které má jít aplikace spouštět (Desktop, Applet). V tomto směru analyzuje původní požadavky zadání práce a snaží se zodpovědět, jestli má cenu je přesně dodržet. Dále zodpoví zásadní otázku, která velmi ovlivní velikost a složitost a výslednou funkci aplikace - jestli má jít o skutečný emulátor se vším všudy, anebo o inteligentní mezivrstvu mezi uživatelem a existující CLI?

2.5.1 Emulovat nebo využít CLI?

Ačkoliv se to může zdát samozřejmé, nebyl prozkoumám jeden docela zásadní fakt. Tím je totiž co se tou grafickou *reprezentací* vlastně myslí. Grafický simulátor příkazové řádky může znamenat buď grafickou nástavbu *existující* příkazové řádky, anebo plně samostatný simulátor. Bylo řečeno že má být kompatibilní s operačními systémy založenými na Unixu a jejich shelly, a že má pracovat na více platformách. Nebylo však řečeno jak. Dvě možnosti takové *reprezentace příkazové řádky* jsou:

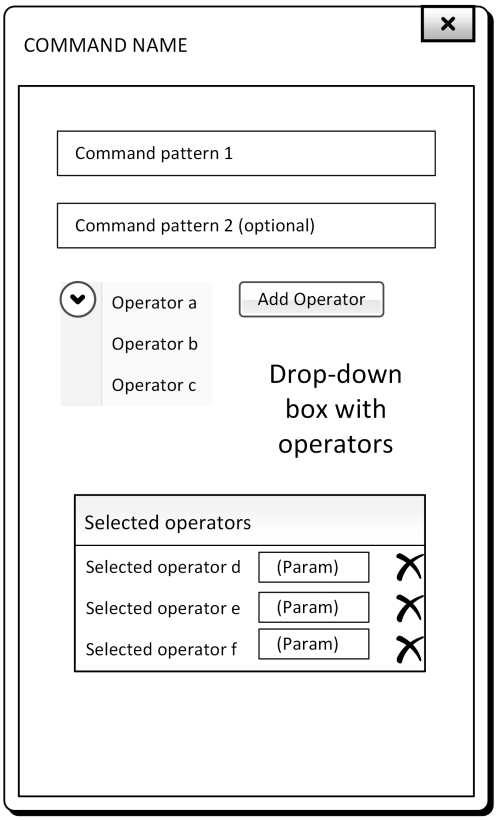
- Kompletní samostatný simulátor



Obrázek 2.15: Wireframe model grafické podoby. Detaily konkrétních prvků budou dány programovacím jazykem, grafickým frameworkem a upraveny podle situace

- Program využívající příkazové řádky shellu v systému

Kompletní samostatný simulátor by měl nesporné výhody - možnost spouštění naprosto nezávisle na systému, kterým by se vlastně jenom „inspiroval“. Pouze by si získal *informaci* o systémové příkazové řádce, o příkazech které podporuje, o operátorech těch příkazů atd. Poté by *nasimuloval* toto prostředí. Takový nástroj by však byl velice složitý. Jednalo by se svým způsobem o malý virtuální systém. Aby měl smysl, musel by vlastnosti shellu mateřského systému kopírovat velmi věrně, a navíc by musel mít kompletní sadu všech příkazů, včetně kompletních zdrojových kódů. Byl by to v tomto případě velký a komplexní systém, takřka virtuální stroj. To je nad schopnosti a rámec této práce.
Přirozená volba tedy padá na druhou možnost. Grafická *reprezentace* příkazové řádky je v tomto případě jenom jakousi nadstavbou, mezivrstvou mezi příkazovou řádkou a uživatelem. Pomocník, který má jen uživateli usnadnit tvorbu nějakým jednoduchým, člověku přirozeným rozhraním, a *přeložit* jí do jazyka a zápisu, kterému rozumí příkazová řádka na hostujícím systému. Tato mezivrstva má vlastní funkce, data může nakonec interpretovat v CLI jinak



Obrázek 2.16: Detail bloku příkazu. Operátory a pole s patterny nejsou u všech prvků stejné, ani jich není stejné množství. Různé operátory také mohou nebo nemusí mít vlastní pattern

mimo pohled uživatele, způsobem, kterému více sama rozumí.

Takže pro upřesnění a zamezení jakýmkoliv pochybám, celý SPsim bude VYUŽÍVAT stávající příkazovou řádku systému, včetně jejích vlastních příkazů (které reálně spouští ona). SPsim tak tedy zadává vhodné povely, stejně jako by to dělal uživatel, a příkazové řádce předává vstupy a bere z ní výstupy, případně měří rychlost její práce, získává a parsuje nápovědu a podobně. Jeho funkce tedy skutečně bude „být grafickou nadstavbou textové CLI“.

S tímto rozhodnutím a požadavkem, aby program fungoval na co největším počtu systémů (v rámci UNIXové rodiny), je nutné zvolit vhodný programovací jazyk. V zadání práce je jmenována JAVA. Je to dobrý nápad hned ze dvou důvodů;

- Javu ovládá autor této práce lépe než většinu ostatních jazyků

- Java je implicitně multiplatformní, jeden zkompilovaný kód tedy půjde spustit všude tam, kde je podporována standardní Java (dokonce je spustitelný na MS Windows, i když příkazy pak nebudou fungovat)

Java je tedy optimální volbou. Navíc má pokročilé grafické knihovny, které dbají o kompatibilitu zobrazení na různých systémech s různým vzhledem. Java si potom sama zařídí, aby se vzhled prvků a okna spouštěného SPsim odvíjel od hostícího systému.

2.5.2 Desktop, Browser, Web

V původním zadání práce se píše „GCLI implementujte jako desktopovou aplikaci a applet do webového prohlížeče“. Pro potřeby analýzy uvažujme ještě jednu možnost. Potencionální možnosti jak provozovat SPsim jsou tedy 3:

- Klasická desktopová aplikace
- Applet
- Siťová aplikace typu klient - server

Desktopová aplikace nevyžaduje příliš vysvětlení. Je to dnes defaultní způsob, jak pracovat na stolním počítači. Systémový shell a CLI je také program, ke kterému se dá jenom obtížně (pokud vůbec) přistupovat z webu (ačkoliv po síti to možné je). Požadavek na Desktopovou aplikaci je tedy zcela oprávněný a aplikace tento formát bude mít.

Problém nastává s požadavkem na applet. Applet je softwarová komponenta. Ta se spouští v *kontextu* jiného programu, kontejneru. Nemůže na rozdíl od desktopové aplikace běžet samostatně. Tímto kontejnerem a kontextem bývá obvykle webový prohlížeč, nebo GUI spouštěcí panel systému. Důležité je, že ačkoliv se v zadání nejspíš myslelo na běh ve webovém prohlížeči, vždy se spouští na KLIENTSKÉ platformě (na rozdíl od webové aplikace či obecně servletu). Navíc má applet díky kontejneru, ve kterém je spouštěn, omezen přístup k systému. V případě interpretace a využívání příkazové řádky je to problém, který by vyžadoval zvlášť benevolentní nastavení bezpečnosti na klientském počítači.

Závěrem tedy je, že applet nedává v tomto použití příliš smysl. Kontejner ve kterém se spouští by ho zbytečně omezoval. Navíc SPsim stejně musí volat a využívat příkazovou řádku systému, což ani s bezpečnostních důvodů nemusí fungovat.

Další potenciální možností jak spouštět SPsim by bylo formou webové aplikace. Simulátor by se tak skládal z klientské části a serverové části. Výhody tohoto řešení jsou takové, že SPsim by si mohl pouštět každý se síťovým

připojením (potažmo internetem) přímo ve webovém browseru (jak se dnes stalo běžným standardem pro většinu takových aplikací pokud mají grafické rozhraní). Bez ohledu na operační systém nebo klienta, každý kdo by měl přístup k aplikaci běžící na serveru by si mohl testovat tvorbu rour a přímo je na serveru spouštět.

To je lákavá představa která má použití například v administraci unixového serveru, kde je potřeba například zjistit informace o stavu systému komplikovanějším skriptem, který si uživatel nedokáže snadno sestavit jen v CLI přes SSH. Tato situace ovšem nenastane příliš často, a pokud má člověk přístup ke stroji na kterém server běží, může si spustit SPsim tam jako klasickou desktopovou aplikaci. Výhody klient-server zařízení se tím tak trochu ztrácí - SPsim je stejně vázán na prostředí serveru. Okruh využití se tím omezuje. Další problém s takovým řešením je bezpečnostní - SPsim, díky závěru z předchozí sekce, bude využívat stávající CLI a příkazy v něm přímo spouštět. Je to tedy otevřená brána přímo k příkazové řádce serveru, což mnoho provozovatelů takového serveru příliš neocení. Bezpečnostní hrozba by to pak byla i tehdy, kdyby SPsim vyžadoval autentizaci a dokonce i kdyby se spouštěl v nějakém sandboxovém režimu a omezenou příkazovou řádkou. Tím by se zase ale znehodnocovala jeho použitelnost. Celá implementace by navíc byla složitější a tedy i toto řešení zavrhneme.

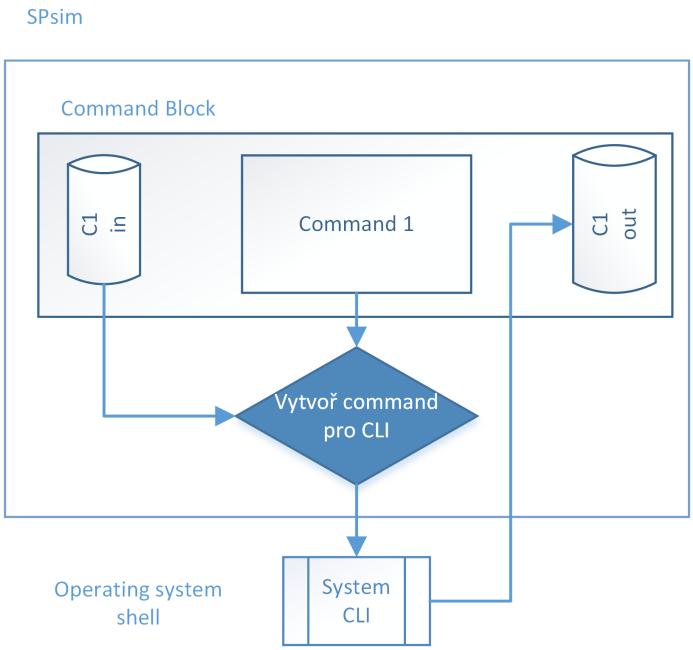
SPsim bude pouze desktopová aplikace.

2.6 Komunikace s příkazovou řádkou systému

V předchozích analýzách bylo rozhodnuto, že SPsim bude program běžící v systému jako desktopová aplikace, bude využívat existující příkazovou řádku, a bude moci sestavovat rouru z podporovaných příkazů (případně testovat jen příkazy samotné). Využitím existující příkazové řádky se tedy myslí to, že program SPsim si zavolá a vytvoří nějaký shell (volme například BASH z důvodu velké rozšířenosti), předá mu vhodně „přeložený“ příkaz, spustí jej, a získá jeho výsledek.

Je nutné rozhodnout, jak toto spouštění shellu simulátorem bude probíhat. Jako omezující podmínka také platí, že simulátor umožňuje stavět z příkazů roury, a příkazová řádka toto musí respektovat. Uživatel si ale na rozdíl od příkazové řádky může příkazy v rouře libovolně krokovat. Je tedy otázka jak tuto komunikaci se shellem zařídit, aby vracel správné výsledky podle akce uživatele.

V druhé sekci o vlastnostech příkazu jako takového (a rozhodnutí že si bude udržovat svoje streamy uložené pro náhled) byl dán základ architektuře, podle které bude příkaz postaven. Pokud voláme samostatný příkaz v SPsim a samostatný příkaz v CLI, není zde nic co by nás omezovalo. Ve skutečnosti se bude dít to, že SPsim vygeneruje výstup pro skript, spustí jej v CLI, a výstup zapíše do výstupní struktury Bloku příkazu. Vysvětlení na obrázku 2.17.

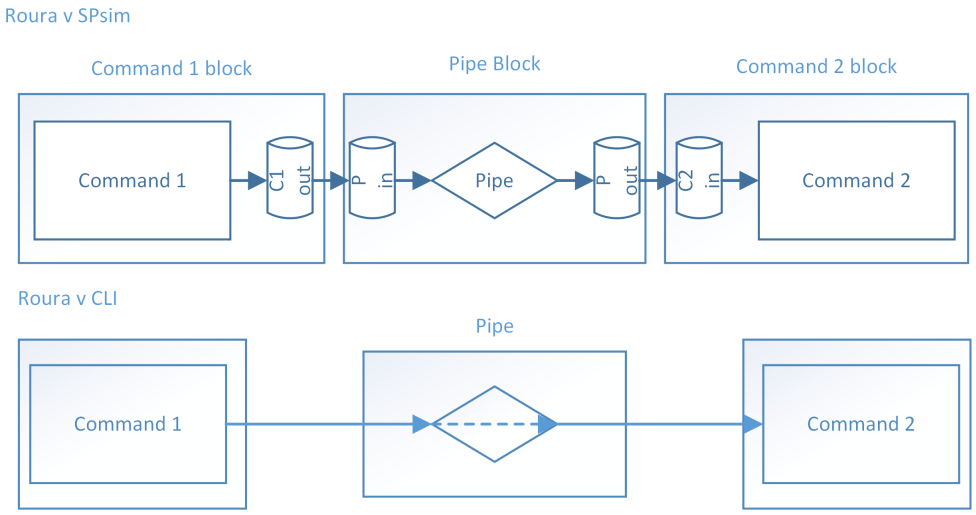


Obrázek 2.17: Způsob jak SPsim spouští a skládá příkaz pro CLI

Proces *vytvoř command pro CLI* musí využít nějakého triku, jak v příkazové řádce svázat vstupní proud z uložených dat v programu a samotný příkaz sestavený uživatelem. Tento proud by se při volání v CLI do příkazu dostal propojením rourou z příkazu předchozího, nebo jako zadaný soubor parametrem. Program to bude řešit tak, že vyvolá dva příkazy, jeden pro výpis vstupních dat a potom je rourou předá v CLI čekajícímu příkazu, který navolil uživatel. Detailněji je to popsáno v kapitole Implementace 3.

V sekci o rourách je zase uvedena modifikace originálního řešení pro potřeby simulátoru. Spojíme tyto dva poznatky dohromady do jednoho schématu porovnávajícího rouru SPsimu s představou roury, která se vkládá do CLI. 2.18

Kroky roury si téměř odpovídají, až na fakt, že v SPsimu, pokud by se měla spouštět celá roura najednou, je zde jedna fáze navíc a tou je překopírování dat z výstupního ukládající struktury na vstupní ukládající strukturu následujícího příkazu (nebo roury). Ladění krokováním jeden po druhém, stejně tak spouštění automatického krokování, není problém (krom delší doby vykonání než kterou by to trvalo rouře CLI). CLI o celkovém tvaru roury v SPsim nemusí vědět. Jelikož ani příkazy zapojené do skutečné roury v CLI neví že jsou samy v rouře (jen berou vstup a zpracují ho na výstup), postupné spouštění SPsimem způsobí vlastně to samé. Z pohledu roury v SPsim se pro uživatele nic nemění, a z pohledu CLI se ve skutečnosti žádná roura ani netvoří a jen se vyvolá jeden individuální krok.



Obrázek 2.18: Náhled na datový tok streamu přes rouru v SPsim v porovnání se (zjednodušeným) pohledem na to samé v CLI

Otázka je co dělat, pokud bude chtít uživatel spustit rouru nebo frontu skutečně najednou. V takovém případě by musel SPsim vygenerovat celý příkaz reprezentující celý řetěz najednou, jenom by stejným trikem zavolal ještě jeden „příkaz navíc“ před jejím začátkem, aby jí tím způsobem předal vstupní data. Sestavenou rouru by předal CLI, ta jí provedla, a vrátila výsledek. Bylo by to stejné jako na obrázku 2.17, jen by se do „command1“ schovala celá fronta. Pro uživatele SPsim je ale jedno jestli na vespod běžící CLI se volá jeho řetězec najednou nebo postupně jeden po druhém. Výsledná data budou stejná a krokováním se zajistí že bude znám tvar i uvnitř spouštěného úseku, což by v případě CLI roury nebylo možné. Funkce *spuštění skutečné roury rovnou* nemá tedy smysl. SPsim tak nikdy nebude předávat CLI v jednu chvíli víc, než příkaz s jednou rourou (pipe), i když uživatel bude vidět jako by se zpracovával najednou celý řetěz jeho příkazů.

2.7 Automatická extrakce dat

V zadání práce je důležitá část o automatické extrakci dat. Přesně se tím myslí extrakci relevantních pasáží z manuálových stránek a operátorů. SPsim je v tom případě platformově nezávislý nástroj, který poskytuje platformově závislá data. Co to znamená: Nejdříve Uživatel SPsim spustí. SPsim pak musí nějak ze systému zjistit *podporované operátory zadané sady příkazů*. Teoreticky by nějakým pokročilým dataminingem šlo rozpoznat i onu sadu kmenových filtračních příkazů na základě čtení všech manuálových stránek, ale jelikož máme pevně stanovenou množinu těchto příkazů (která by měla být na UNIX-

like systémech stejná), bude potřeba získat jen jejich operátory.

Současně s tím, aby uživateli při používání těchto operátorů byla nabízena relevantní pomoc, musí umět SPsim i získat data přímo o těchto operátorech.

Navíc je tu další problém - Zatímco pevně zadané příkazy nebudou nejspíš na různých systémech měnit příliš svůj zápis, tedy například množství vstupních hodnot (patternů), u operátorů se to říct nedá. Navíc operátory předem neznáme, a operátory samotné MOHOU a NEMUSÍ mít samy vlastní vstupní hodnotu. Systém tedy bude mít na starost čtyři úkony s automatickým obstaráváním dat ze systému. Ze zadaného seznamu příkazů zjistit:

- Část nápovědy věnující se popisu příkazu
- Seznam operátorů
- Nápovědu k operátorům
- Tvar těchto operátorů (zda mají vlastní vstupní hodnotu či ne).

\+ doplňující schopnost

- získat tvar a počet vstupních hodnot samotného příkazu

2.7.1 Manuálové stránky

Manuálové stránky příkazu se získávají vlastním speciálním příkazem ve tvaru „man *název_stránky*“ kde *název_stránky* je příkaz pro který chceme získat nápovědu, a příkazová řádka nám je vypíše ve formátovaném textovém tvaru. Tato nápověda je uložena ve většině unixových systémů v adresáři „/usr/share/man“ pro každý příkaz (v komprimovaném formátu). Není nutnou podmínkou aby měl program či příkaz dokumentaci, ale pokládá se to téměř za samozřejmé - chybějící dokumentace by se pokládala za známku nízké kvality programu. Manuálové stránky jsou v UNIXu už od 70.let 20. století. Díky tomu a neustálému udržování kompatibility si zachovaly původní značkovací jazyk pro textový procesor, ve kterém jsou psány: NROFF [4]. NROFF je předchůdce dnešního TeXu a ačkoliv je plain text, má poměrně dost neintuitivní a složitý zápis. Jeho vývoj pokračoval přímým nástupcem TROFF a GROFF (svobodná verze) které mají více schopností a možností, nicméně syntakticky zůstávají pořád velmi složité. NROFF (GROFF) se tedy stará o zobrazování manuálových stránek i pro konzoli. Pokusy autora o parsování informací přímo z tohoto zdrojového formátu skončily neúspěchem. Jelikož se nepodařilo najít ani žádný vhodný parser do modernějších a lépe zpracovatelných formátů, je nutné parsovat informaci z textového výstupu manuálových stránek a příkazu MAN. SPsim tak tedy musí získávat informaci k vytěžení tak, že nejdříve získá dokument voláním příkazu v CLI "MAN"a název ohledávaného příkazu. NROFF (GROFF) a podobné díky své z dnešního pohledu

zastaralosti nepodporují řazení do nějakých příkazově rozeznatelných sekcí (jako to například LaTeXumí příkazem `\section`, je proto bohužel nutné snažit se získat požadované informace na základě nějakých dalších charakteristik splňující nějaký regulární výraz (aby to šlo dělat strojově).

Manuálové stránky by měly dodržovat obecně vždy podobný formát:

- NAME (Jméno) - jméno příkazu s popisem toho co dělá
- SYNOPSIS (Shrnutí) - zde by měl být popis toho jak program spustit a jeho možností
- DESCRIPTION (Popis) - popis fungování
- EXAMPLES (Příklady) - příklady jeho využití
- SEE ALSO (Příbuzná témata) - seznam příbuzných témat a funkcí

V případě že se snažíme získat informaci o samotném příkazu, mělo by stačit extrahovat část začínající NAME a končící SYNOPSYS. V případě že budeme získávat manuálovou nápovědu k operátorům, je potřeba tyto operátory načíst ideálně z DESCRIPTION. Zde mají společnou vlastnost, že jsou oddělené od začátku řádky několika mezerami a pak ve tvaru „-operátor“. Text za nimi většinou označuje jejich použití a případné parametry a další řádky jsou popis.

2.7.2 Parametry příkazů

Parametry příkazů jsou uvedeny v SYNOPSIS, parametry (jsou li jaké) u operátorů jsou uvedeny za ním. Obvykle je to bývá ve tvaru operátor == PARAM s tím, že během tohoto zápisu mohou být na více místech mezery, hranaté závorky, dvojité pomlčky, aliasy pro operátor a podobně.

2.7.3 Problémy s parsováním

Ukázalo se, že ačkoliv je jednotný názor a doporučení na to jak psát manuálové stránky a jakého formátu se držet, není tomu tak ve všech případech a pisatelé těchto stránek podobné formáty nedodržují zcela. Příkladem je třeba příkaz „sed“, který na rozdíl od většiny ostatních příkazů nemá svoje operátory v sekci DESCRIPTION, ale ve vlastní sekci OPTIONS. Někdy zase splývá popis v NAME a SYNOPSIS dohromady, někdy sekce DESCRIPTION má velmi *zvláštní* formátování.

Je vidět že manuálové stránky jsou psány mnoha lidmi kteří vycházejí z nějakých „good practices“ a zvyklostí, ale nepočítá se se strojovým čtením a extrakcí dat. To je zhoršeno ještě použitím zastaralého textového procesoru, který nepodporuje značkování textu do použitelných a jasně daných logických a sémantických celků. V následující kapitole 3 je ukázáno jak se toto podařilo

částečně vyřešit pomocí samotných filtrů a regulárních výrazů.
Shrnutí: Dá se říci že pomocí regulárních výrazů se daří dobře získávat informace o samotných příkazech. Daří se i na testované množině příkazů uspokojivě detekovat operátory, a to zda mají vlastní pattern. Pro operátory a i ve valné většině případů i jejich konkrétní pasáže nápovědy. Nedaří se přesvědčivě detekovat počet patternů (vstupů) u příkazů.

Závěr a navržené řešení:

- Program bude mít dvě možnosti extrakce: **Asistovanou**, a **Automatickou**

- Asistovaná extrakce bude využívat konfiguračního souboru, který obsahuje seznam příkazů, počet jejich parametrů, seznam jejich operátorů bez parametrů a seznam jejich operátorů s parametrem. Program na základě těchto informací přesněji vyhledá manuálové nápovědy.

- Automatická extrakce je experimentální. Při ní program na začátku zná jenom a pouze seznam příkazů. Veškeré ostatní údaje si snaží vyhledat sám (jaké mají na daném systému možné operátory, zda tyto operátory mají vlastní pattern, a nápovědy k oběma). NEUMÍ zatím sama zjisti počet patternů samotného příkazu, tento je defaultně nastaven na 2. Toto číslo by nemělo snížit funkčnost příkazu - tam kde příkaz nemá dva ale jen jeden či žádný, vyhlásí program při jejich vyplnění chybu. Uživatel sám pozná díky nápovědě příkazu (kterou se daří extrahovat dobře), zda a kolik patternů má u příkazu využít.

Tyto opatření jsou rozumným kompromisem k problémům které vznikly při strojovém vytěžování dat z manuálových stránek. Neomezují nijak funkčnost, všechny důležité aspekty procesu fungují v obou případech. Použití asistované detekce konfiguračním souborem přináší přesnější výsledky a tvar graficky reprezentovaných bloků příkazů. Nevýhoda toho je, že nemusí být tento konfigurační soubor zcela pravdivý i na ostatních platformách. Byl vytvořen a testován na systému UBUNTU 14.04. Tento soubor se dá nicméně snadno nahradit vlastním. Formát tohoto souboru (v aplikaci bude uložen v adresáři resources/conf.dat) vypadá následovně:
zde začíná soubor

-název příkazu-
-počet patternů-
-list operátorů bez vlastního patternu, bez pomlčky a oddělené středníkem-
-list operátorů s vlastní patternem, bez pomlčky a oddělené středníkem-

Tento vzor se opakuje. Pokud chybí celá skupina patternů, nechá se místo ní prázdný řádek. Počet řádků tak bude vždy násobek 4, neboť vždy jsou 4 řádky

na příkaz. Výjimka je poslední řádek označený

-END-

Počet řádků tak bude x*4 + 1 kde x je počet příkazů.

Kapitola 3

Implementace

Tato kapitola se týká implementace programu SPsim na základě závěrů z kapitoly o analýze. Ještě před popsáním architektury a jednotlivých celků programu by bylo dobré zmínit obecné informace o kódu. Celý program má 1561 čistých a skutečných řádků kódu - bez mezer, komentářů, prázdných řádek a bez testů. Toto číslo se možná nezdá jako moc, nicméně autor snížil výsledný počet řádek dost i tím, že se vyhnul dělení kódu do samostatných modulů více než je nezbytně nutné a používal, tam kde to šlo, takové nástroje, které snižují počet řádek na co nejmenší možnou úroveň. Dalším faktem ovlivňujícím tuto hodnotu o velký faktor je využití integrovaného GUI developeru, který inicializaci a nastavení GUI generuje do řádků zhuštěných až na takřka nečitelnou úroveň. Reálný celkový počet řádků i s komentáři je tedy o dost vyšší.
Pro zajímavost jsem se pokusil vypočítat náročnost na vývoj projektu pomocí základního COCOMO [19] modelu, a vyšla mi hodnota asi 4 měsíce práce na jednoho člověka. Tento odhad se ukazuje jako poněkud pesimistický, neboť skutečná práce na programu odpovídala cca 2-3 měsíců, podle množství rešerše a analýzy.

Ačkoliv měl autor této práce s Javou relativně dost zkušeností, jako problém se v jednu chvíli ukázala práce a tvorba GUI. Nutné bylo tedy nastudování dosti informací a to jak v javě GUI pracuje s vlákny, neboť v některých případech a snaze oddělit logiku programu od grafiky aplikace zamrzala. Tyto problémy se naštěstí podařilo vyřešit.

3.1 Architektura programu

Program je psán v jazyce Java, využívá openJDK 7 a jeho grafická stránka je postavená na knihovnách SWING. K vývoji jsem si zvolil vývojové prostředí

[19] vzorec pro výpočet základního COCOMO zde použitý je : *měsíců na jednoho člověka* = 2.4 * KSLOC$^{1.05}$ kde KSLOC znamená „thousands of source lines of code“

IntelliJ Idea Enterprise, které má (nejspíše i díky tomu že je tato vyšší verze komerční) velmi pokročilé funkce. Ačkoliv většina dostupných IDE má schopností tolik, že je jeden programátor nedovede všechny ani využít, u IDE od IntelliJ je vidět tento komfort při psaní samotného kódu v jakémkoliv případě - schopnosti automatické detekce a doplňování kódu se z pohledu autora této práce zdají mezi konkurencí nepřekonatelné.
Celková architektura programu odpovídá modifikaci MVC[20] kde je model a view v některých výjimečných případech kvůli usnadnění komunikace modelu s formulářem propojeno (viz sekce datové jednotky a grafický formulář), a ze strany knihoven SWING se jedná zase o další modifikaci, neboť swing sám kombinuje Řadič(Controller) a Pohled(View) do jednoho logického celku [5] (což odpovídá jednodušší architektuře zvané „Model1“). Základní architektura programu mimo tento pohled se ale dá rozdělit na 4 hlavní komponenty, kterým odpovídá rozdělení zdrojových kódů v programu na 4 hlavní package.

- Datové jednotky (package commands)
- Intepret shellu (package core)
- Extrakce a příprava dat (package data)
- Grafický formulář (SPsimForm.java)

3.2 Datové jednotky

SPsim.src.commands.

Hlavním účelem datových jednotek je reprezentace příkazu v programu. Jde vlastně o datové kontejnery, které mají za úkol udržovat informaci o zadaném příkazu, jeho nápovědách, operátorech, a o změnách a parametrech které na nich uživatel pozměnil. Hlavní jednotkou v tomto případě je Blok. Blok reprezentuje v aplikaci uživatele jeden příkaz s parametry a patterny, v programu je to abstraktní třída která si udržuje všeobecné údaje jako je jméno, manuálový popis, chybové k a vstupní a výstupní data. Tato třída sama dědí z virtuálního konektoru z balíčku commands.connectors, který zase dědí z tříd Swingu JPanel nebo JTextfield. Tímto způsobem se tudíž řeší propojení s grafickými prvky. Je to v tomto případě výhodnější, protože v GUI pak stačí vytvořit rovnou instanci některého z potomků Block (nebo Pattern v případě textového pole), a data tak budou asociována s příkazem a tímto vlastním blokem budou pak rovnou zadávána uživatelem do modelu. Vlastní proměnné pak, kde je to třeba, jsou přetíženy metodou v potomkovi, a tím jsou s výslednou grafikou propojeny jeho proměnné. Alternativní metoda by byla mít zcela oddělené GUI od logiky programu a modelu, ale znamenalo by

[20] Model-View-Controller

to velký komunikační overhead, který v této situaci není nutný. Navíc momentální řešení ani příliš neomezuje portovatelnost programu na jiné GUI - stačí přepsat konektory z balíčku connectors.

Block sama o sobě je abstraktní třída, ze které dědí konkrétní třídy, které se teprve v programu inicializují. Důvod proč se nepoužívá pro reprezentování příkazu jen samotný Block je v tom, že příkazů je několik typů. Hlavní motivace je ta, aby příkazy samy o sobě uměly generovat skript ve vhodné podobě odpovídající svému obsahu, a proto každá implementuje tuto metodu (která je v Bloku abstraktní) jinak. Tyto typy jsou:

CoreCommand slouží pro naprostou většinu příkazů. Odlišuje se od ostatních typů tím, že může mít operátory a více parametrů (patternů).

FlowControlCommand slouží pro manipulační příkazy. Tedy tato třída je v momentální verzi programu vyhrazena Rouře (pipe), nicméně se dá použít i pro redirect do souborů a podobně.

CustomCommand, jak název napovídá, je třídou jejíž jediná funkce je přijímat od uživatele svůj „pattern“, který bude vracet jako vlastní skript pro zpracování.

Třída Operator pro uchovávání operátorů a informací o nich, IOdata pro konkrétní blok zajišťuje správu a udržování standardních streamů, třída CommandFactory vytváří na základě přijatých univerzálních tříd typu TemplateCommand vhodné instance konkrétních datových tříd.

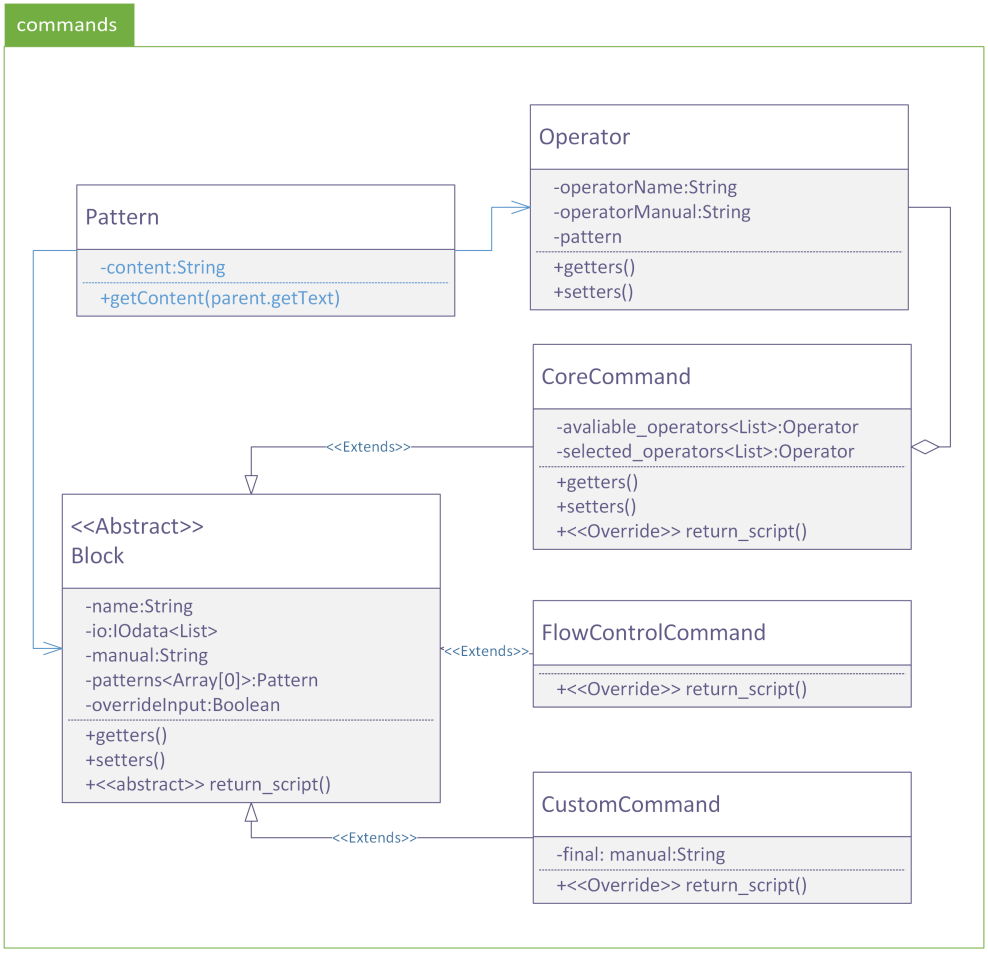
Náhled na to, jak jsou tyto vztahy pro hlavní datové třídy definovány, nabízí obrázek 3.1.

3.3 Intepret shellu

SPsim.src.core.

Interpret shellu (Třída ShellEngine.java) je vrstva programu, která umožňuje komunikaci s příkazovou řádkou hostitelského systému a zpracováváním skriptů. ShellEngine běží v programu jako *SINGLETON*. Singleton je z pohledu aplikace užitečnější, aby se nemusela vždy inicializovat celá třída znovu jenom pro zavolání jednoho příkazu.

Tato třída má tedy jednostranné použití - spouštět skripty. Příkazová řádka systému se vyvolá jako proces spouštějící systémový Bash s parametrem -c a cestou k adresáři, ve kterém se má daný příkaz spouštět. Bash se vytváří knihovnami Java.Lang.Process a Java.Lang.Processbuilder. Výhoda těchto knihoven je, že umožňují pohodlnou kontrolu průběhu procesu, včetně



Obrázek 3.1: UML diagram způsobu jak jsou propojeny hlavní datové třídy v programu. Spojení s GUI moduly není znázorněno.

rovnou zachytávání standardního a chybového výstupu a exit statusu. Vytvořený Bash provede příkaz a ukončí se. Důsledkem tedy je že na pozadí neběží příkazová řádka stále, ale volá se jen podle potřeby. Simulování rour to nijak nevadí a alespoň je zaručena větší „odolnost proti zanesení řádky“, neboť díky tomuto spouštění on-demand nemá paměť na jednotlivé příkazy v simulované rouře se tedy nemají jak nepřímo ovlivnit.

Důležitá informace nastíněná několikrát v kapitole "Analýza"je způsob, jakým v tomto selektivním spouštěním procesů simuluje vstupní kanál. Vstupní kanál je ošetřený následujícím trikem. Ve chvíli, kdy uživatel dá povel udělat spustit nějaký příkaz, v ShellEngine se vytvoří řetězec podle následujícího vzoru:

$$\textit{"echo "} + \text{IOdata.getInput()} + \textit{"/"} + \text{command.getScript()}$$

který se posléze spustí v BASH. Ve skutečnosti se stane to, že BASH zavolá povely DVA, oddělené rourou. Jeden je pro prostý výpis dat, a roura pak způsobí, že skript uživatelem zadaného příkazu jej dostane na stdin stejně, jako by se stalo v normální CLI. Pokud by se kanály v simulátoru ukládaly do souborů na disk, fungoval by trik stejně - jen by se místo echo volil příkaz *cat* a cesta k danému datovému souboru.

3.4 Extrakce dat

SPsim.src.data. Extrakce dat má na starost má na starost následující funkce:

- Přípravu příkazů do seznamu pro uživatele
- Extrakci konkrétních manuálových pasáží pro příkaz
- Extrakci konkrétních manuálových pasáží pro operátory

\+ v závislosti na nastavení operačního módu

- Detekci podporovaných operátorů a detekci jejich formátu

V tomto balíčku jsou 4 třídy. FileSourceData a SystemSourceData se starají, podle svého názvu, o asistovanou extrakci dat ze systému, anebo plně automatickou. Volají se podle zvoleného módu a vrací stejný výsledek - seznam objektů TemplateCommand. Využívají pro to sadu speciálně připravených filtračních příkazů s regulárními výrazy, kterými zpracovávají manuálové stránky požadovaného příkazu.

Tyto příkazy jsou uloženy v IDataPrepare interfacu jako final statické Stringy, a základní z nich mají následující podobu:

```
"awk '/NAME|SYNOPSIS/{flag=1;next}/^[A-Z]+/{flag=0}flag'"
"sed -n -e \"/^DESCRIPTION/,/^[A-Z]/ p\"| grep -v -e \"^[A-Z]\"";
"sed -n -e '/^\\s\\+-operator"[^a-zA-Z0-9]/,/^\$/ p'"
"grep '^\\s\\+-[a-zA-Z0-9]'"
```

Tyto příkazy (pro detekci relevantní manuálové pasáže příkazu až do další kapitoly, extrakce pasáží manuálových stránek věnujích se přímo danému operátoru či detekce samotných operátorů) jsou zjednodušeny tak, jak to jen autor dokázal, a následně se v programu kombinují pro dodání požadovaných výsledků. Třída IDataPrepare má v sobě ještě několik dalších momentálně nevyužitých skriptů na extrakci dat z manuálových stránek, které vznikly jako vedlejší produkt, ale které mají potenciál být použity při dalším vývoji programu.

TemplateCommand.java je univerzální datová třída pro blok příkazu před tím, než podle ní CommandFactory z balíčku commands nainicializuje konkrétní správnou instanci. List těchto template commandů je zobrazen uživateli v panelu výběru příkazů. Ve chvíli kdy uživatel na příkaz pokliká se aktivuje listener listu a pošle kopii vybraného prvku do CommandFactory.

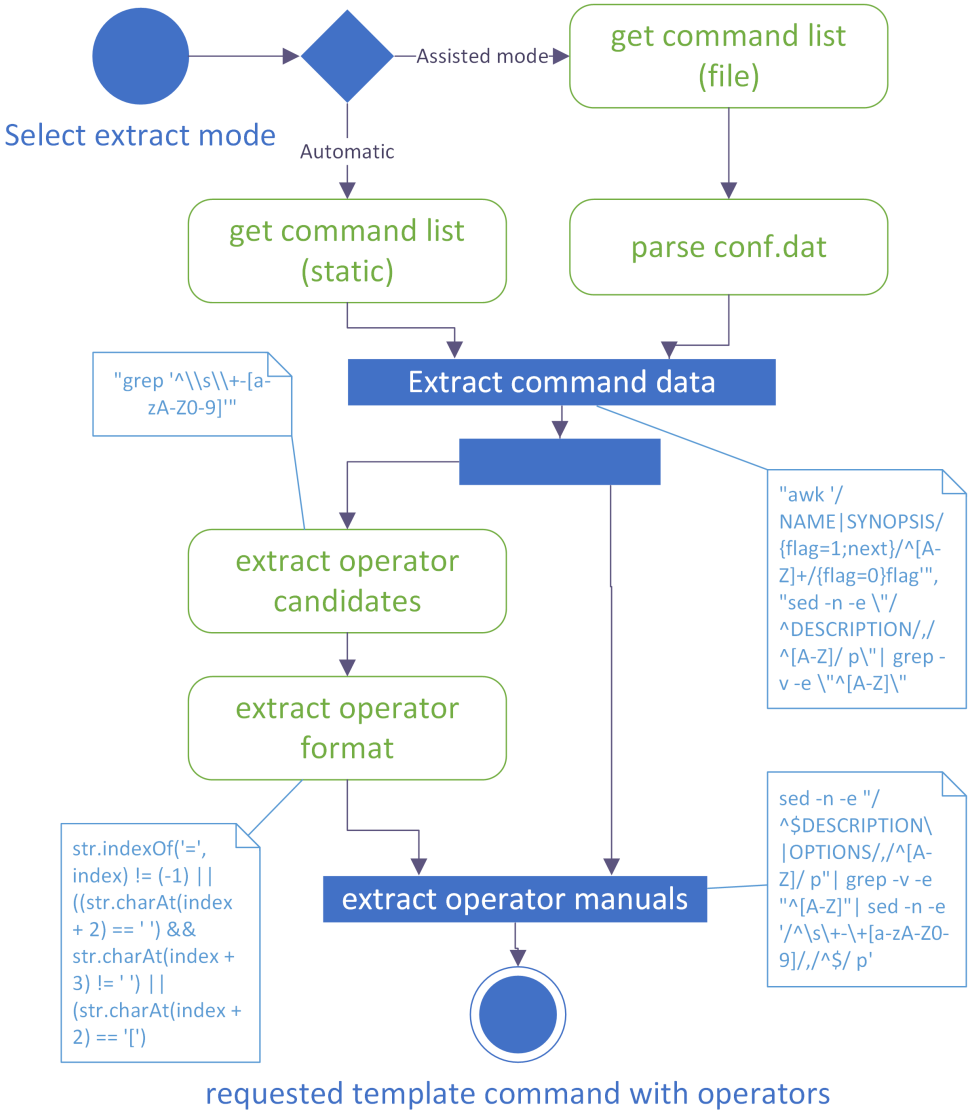
Diagram aktivit průběhu extrakce dat prvku je zobrazen na obrázku 3.2.

3.5 Grafický formulář

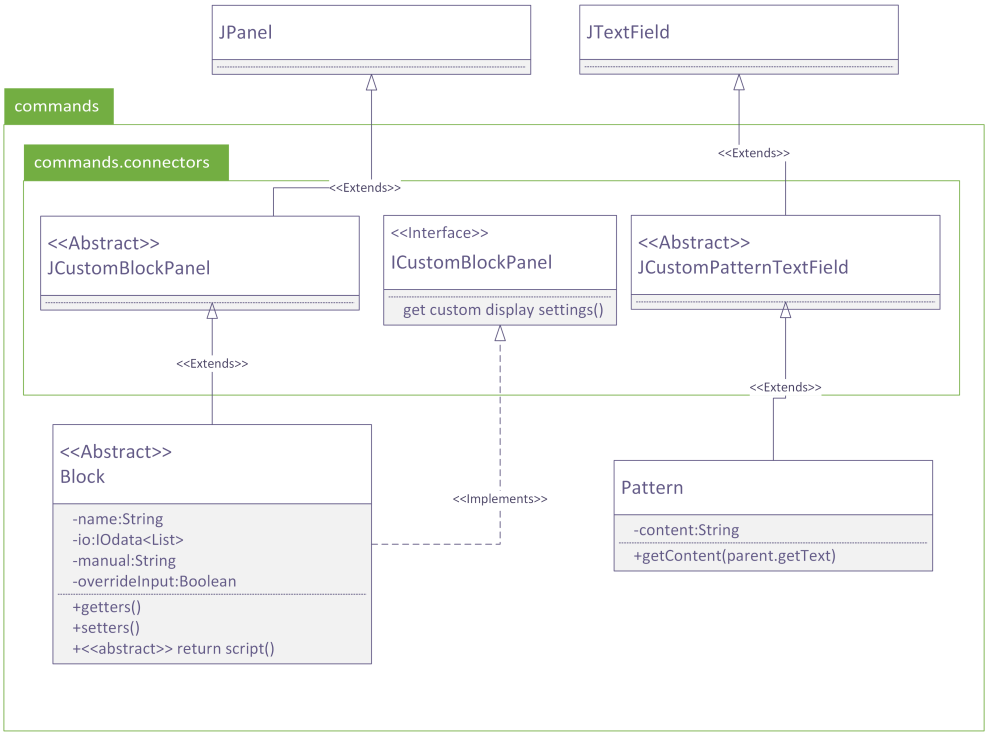
K vytváření hlavních a statických ovládacích prvků grafického rozhraní byl využit integrovaný GUI designer. Je to základní nástroj který se hodí hlavně na rozvržení statických prvků jako rámců, kontejnerů a panelů. Prvky které bylo nutné vytvořit manuálně jsou všechny dynamické části jsou uvedeny v SPsimForm.java, včetně jejich listenerů a handlerů. Tento soubor je díky tomu poměrně veliký (700+ řádek). Vytváření grafických prvků a nastavování listenerů je poměrně přímočaré, takže není potřeba se mu tu dlouze věnovat. Primární preferovaný Layout manažer pro většinu bloků je GridBagLayout, pro případy řazení bloků do fronty je to BoxLayout a pro bloky samotné také BoxLayout, jenom orientovaný na Y.DIMENSION. Výjmečně je pak používán BorderLayout pro rozmístění ovládacích prvků v Bloku a pro vnitřní uspořádání přidaného operátoru.

Je vhodné zmínit ještě způsob, jakým je (v sekci o datových jednotkách) už nastíněné spojení datového modelu bloku s grafickou reprezentací. 3.3 Tuto funkci zajišťuje právě abstraktní konektorová třída, ze které blok dědí. Tato třída zase dědí z požadovaného grafického elementu SWINGu, v případě bloků je to JPanel, v případě textového pole pro náhled skriptu je to JTextField. Blok ještě implementuje interface ICustomBlockPanel, které ve spojení s grafickým předkem JPanel nebo JTextfield ve své instancované podobě nastavuje barvu a rozměry bloku. Takto je například řešen CustomCommand a FlowControlCommand, které mají každý jinou šíři a jinou barvu.

Za zmínku ještě stojí panel pro náhled aktuální podoby skriptu v textovém režimu, který zvýrazňuje tu pasáž, která odpovídá aktivnímu bloku v



Obrázek 3.2: Extrakce dat v automatickém a asistovaném režimu. Automatický režim musí provést více kroků.



Obrázek 3.3: Znázornění způsobu jakým jsou datové komponenty propojeny s grafickou nadstavbou

rouře. Při jeho obnovení (nebo při provedení odpovídající akce) se volá funkce singletonu ShellEngine, které se pošle seznam bloků (instancializovaných jako Component, tedy ještě jako předka JPanelu) z hlavního panelu aplikace, kde uživatel staví bloky. Zde je vidět jasná výhoda silného používání dědičnosti v SPsim - ShellEngine zpracovává tento seznam přetypováním jen do Bloku, a volá abstraktní metodu na vrácení správné podoby skriptu. Jak je to možné? Pod prvkem Component se skrývá díky dědění Blok, a pod blokem přímo konkrétní zvolené instance datových tříd. Jelikož každá z nich implementuje tuto metodu Bloku po svém, přetypováním jen na Blok dostáváme správnou informaci rovnou od skutečných instancí pod ním, aniž bychom museli přetypovávat na konkrétní třídu pomocí „if INSTANCE OF“.

Tento String který reprezentuje skutečnou podobu skriptu je poté zobrazen v tomu určeném panelu a ihned na to je vlastní zvýrazňovací funkcí vypočítáno místo v řetězci, které odpovídá označenému prvku, tedy většinou části roury, a místo se zvýrazní.

Kapitola 4

Testování

Testování probíhalo dvojím způsobem. Testováním samotného interpretu příkazů (třídy ShellEngine) už během vývoje, a testování finálního produktu na základě zadaných scénářů (sady různých ověřovacích skriptů). Dále proběhlo ověřovací spouštění na třech virtuálních verzích linuxu - Ubuntu 14.04 (verze na které byl program vyvíjen a měla by s ním být zaručena největší kompatibilita), poté spouštění a zběžné otestování na Linuxu Mint 17.1 cinamon, a open SUSE 13.2 s prostředím KDE. Na všech prostředích (po nainstalování Javy) šel program spustit a zběžným vyzkoušením nebylo znát že by se jeho funkčnost odlišovala od testování na mateřském Ubuntu, pouze na SUSE běžel program pomaleji (to mohlo ale být dáno nastavením virtuálního prostředí). Pro zajímavost - program byl spuštěn i na MS Windows 8.1. Program šel spustit, jeho ovládací prvky reagovaly stejně, jediné co skutečně nefungovalo bylo spouštění příkazů - program správně ve své skryté konzoli vykazoval chybu nenalezení shellu.

4.1 Intepret příkazů

Pro testování příkazů pomocí interpretu se nabízí vytvoření unit testů. To bylo nicméně vynecháno z důvodu častých úprav zdrojového kódu - samotné jádro aplikace (interpret příkazů a extrakce dat) bylo vyvíjeno zvlášť s častými a velkými změnami jako samostatný program, a zvlášť bylo vyvíjeno grafické prostředí. SPsim byl vytvořen až na konci spojením těchto částí do jednoho, a integrita tohoto spojení pak byla testována testovacími sadami příkazů. Jádro se tak testovalo už během vývoje a průběžně v tomto odděleném stavu. Tento způsob spočíval ve spouštění několika příkazů a výstup byl ručně kontrolován s výstupem ze skutečné příkazové řádky. Dále také bylo testováno nezadávání údajů nebo chybějící či přebývající znaky v parametrech. Tentímto základním způsobem se podařilo odladit hlavně chyby typu null pointer a příliš mnoho nebo příliš málo mezer a uvozovek automaticky vložených při konstrukci skriptu pro příkazovou řádku. Nadstavby a další vrstvy volají ten

samý kód který byl tímto způsobem odladěn ručně, a tento kód se volá opakovaně. Bylo vlastně potřeba otestovat to, aby program správně skládal do skriptů pro CLI parametry, názvy příkazů a odděloval správným počtem mezer a uvozovek. Šíře možností které mohly nastat byly takto omezeny dost, aby se dalo prohlásit, že testování bylo pro potřeby programu dostatečné.

4.2 Scénáře

Po sloučení grafické a logické části programu dohromady byl program testován pomocí sady příkazů a také náhodným přesouváním jejich pořadí v rouře. Tato sada obsahovala asi 10 různých kombinací několika programů a roury (byly voleny tak aby byl alespoň jednou vyzkoušen každý podporovaný příkaz), které se ručně zadávaly a spouštěly. Testována byla hlavně schopnost GUI předávat data jádru, a tato data získávat zpět a zobrazit uživateli. Z důvodu těsného propojení datových objektů a interaktivních GUI prvků, kam se zadávají údaje, se ukázalo, že program sám o sobě nevykazoval chyby v interpretaci příkazů. Některé příkazy byly zaměřeny na záměrné vyvolání chyb - a program správně o chybách informoval (respektive předal informaci o chybě z příkazové řádky). Zbytek grafického prostředí nijak hlouběji a koncepčně testován nebyl.

Závěr: Ačkoliv testy nebyly prováděny žádným plně automatizovaným a speciálním způsobem, funkčnost (minimálně alespoň jádra) je poměrně dobře vyzkoušena. Je to díky charakteru a nakládáním s daty v programu, kde se neočekává nějaké složitější větvení kombinací momentálních stavů aplikace, aby se vyskytovaly neočekávané problémy. Pipeline zpracování a cesta dat od uživatele k intepretu řádky a zpět je díky zvoleným řešením v zásadě jednoduchá, dost přímá a není příliš prostoru pro větvení (a tedy nějaké neočekávané stavy). Není vyloučeno že se nějaká chyba (jako v každém software) časem objeví, ale věřím, že pro tyto účely byla aplikace otestována dostatečně.

Pro hlubší testování automatického získávání dat je potřeba mnohem větší datová základna - extrakce dat fungovala stejně na všech třech virtuálních systémech, ale to bylo zřejmě pro to, že na všech byla stejná verze těchto příkazů se stejnými manuálovými stránkami.

Kapitola 5

Zhodnocení

Na konci této práce přichází čas na zhodnocení, zda se podařilo splnit její cíle. Věřím že ano. Byly dostatečně prozkoumány důvody proč má takový program jako SPsim smysl. Analýzou a vývojem bylo zjištěno, jaké požadavky by na něj *měly být*. To bylo porovnáno s původními požadavky které v zadání *byly*. Ukázalo se, že většina z nich souhlasí a došlo se ke stejným závěrům, v případech kdy to tak nebylo (verze v podobě appletu, rozbočování) se snad povedlo dostatečně tyto odchylky obhájit. Na základě upravených požadavků byl vytvořen fungující prototyp programu, který byl navíc obohacen o další možnosti, které se ukázaly jako vhodné, praktické a zároveň v rozsahu práce dosažitelné (například interaktivní zpětná vazba náhledem na tvar skriptu ve formátu pro příkazovou řádku).

Oficiální cíl - vytvořit grafickou reprezentaci příkazové řádky simulující podmnožinu její funkčnosti - se podařilo splnit. V některých detailech více, v některých méně, ale věřím že byl dán touto prací solidní a obsáhlý základ pro další rozvíjení tohoto programu a celého konceptu.

Zhodnotit ovšem skutečnou použitelnost programu, smysl konceptu a přínos pro uživatele bude možné až časem, pokud ho budou moci využívat lidé. Program byl nakonec vytvořen *pro ně*, pro uživatele, kteří mají s příkazovou řádkou v textové podobě problém. Cíl tohoto projektu bude tedy splněn teprve tehdy, kdy bude moci uživatel prohlásit - tento nástroj mi pomohl.

5.1 Možnosti dalšího vývoje

Vývoj SPsim se může snadno ubírat dalším směrem. Implementovaná a prozkoumaná funkčnost tvoří pouze podmnožinu možností unixové příkazové řádky. Logickým krokem bude tedy rozšiřování jeho schopností. Jako jedna z prvních možností se nabízí chytřejší práce s příkazovou řádkou - například spouštět

testované příkazy v její jedné instanci. Momentální řešení dobře slouží potřebám roury a fronty, ale pro budoucí rozvoj by bylo dobré přidat například podporu systémových proměnných (a možnost si definovat vlastní), obohacení schopnosti textových oken zobrazujících vstup a výstup příkazu o zvýrazňování barvami a například nástrojem Diff. V delší perspektivě by mohlo jít i připravit výukový mód a nějakou verzi inteligentního vyhledávání operátorů a příkazů, například podle kategorií (pro rychlejší hledání toho co uživatel potřebuje). Možností je mnoho a cesta otevřená. Program by ovšem měl i v případných budoucích verzích pamatovat na to, aby byl jednoduchý na použití a přehledný.

Závěr

Pro autora je přínos této diplomové práce jednoznačný. Přinesla mi mnoho zkušeností. Simulovat příkazovou řádku a možnost usnadnit práci s ní v nějakém příjemnějším prostředí jsem bral jako výzvu - neboť sám mám s její textovou podobou plnou filtrů a rour četné a ne vždy šťastné zkušenosti. Další výzva byla tvorba samotné grafické aplikace pro desktopové rozhraní - něco, s čím jsem se do této doby pořádně sám nesetkal. Poučil jsem se a některé věci bych dělal nyní jinak, hlavně co se týče procesu psaní zdrojového kódu a analýzy, ale věřím, že ve výsledku se program povedl. Co se vlastního přínosu této práce týče - věřím že pomohla rozšířit překvapivě málo prozkoumanou cestu usnadnění tvorby řetězů komplexních příkazů v CLI. Graficky zpřístupnit CLI se pokoušelo už mnoho projektů, ale většinou se jí snaží jen přetvořit do náhrady pracovní plochy. Usnadnění jejího využití jako nástroje rour a filtrů zůstává pozapomenuto a je odsunuto s představou že jinak než dlouhým textem v příkazové řádce s množstvím nepřehledných operátorů to nejde. Doufám, že tato práce alespoň trochu pomůže svým přínosem tento fakt změnit.

Literatura

[1] IEEE, T.; Group, T. O.: The Open Group Base Specifications Issue 7. http://pubs.opengroup.org/onlinepubs/9699919799/, stav z 3. 5. 2015.

[2] IEEE, T.; Group, T. O.: 1003.1-2008 - Standard for Information Technology - Portable Operating System Interface (POSIX(R)). https://standards.ieee.org/findstds/standard/1003.1-2008.html, stav z 4. 5. 2015.

[3] Verma, P.: Gracoli: A Graphical Command Line User Interface. http://pramodverma.info/gracoli/gracoli.pdf, stav z 2. 5. 2015.

[4] McIlroy, M. D. .: A Research Unix reader: annotated excerpts from the Programmer's Manual, 1971–1986. *Technical report*, 1987.

[5] (Oracle), O. C.: A Swing Architecture Overview. http://www.oracle.com/technetwork/java/architecture-142923.html, stav z 4. 5. 2015.

Příloha **A**

Seznam použitých zkratek

GUI Graphical user interface

CLI Command line interface

Příloha **B**

Obsah přiloženého CD

- `software`
  - `resources` ..... soubory nutné ke správnému chodu programu
  - `src` ..... zdrojové kódy implementace
  - `readme.txt` ..... stručný popis programu
  - `SPsim.jar` ..... spustitelný soubor programu
- `text` ..... text práce
  - `DPKmochOndrej.pdf` ..... text práce ve formátu PDF
  - `DPKmochOndrej.rar` ..... zabalený soubor se zdrojovými daty v LaTeX