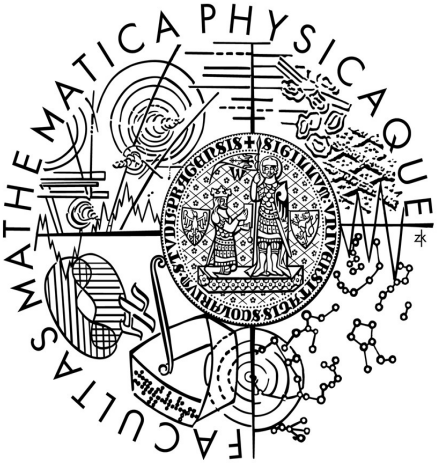
Univerzita Karlova v Praze

Matematicko-fyzikální fakulta

# BAKALÁŘSKÁ PRÁCE




Ondřej Dušek

## BashCommander

Katedra softwarového inženýrství

Vedoucí bakalářské práce: RNDr. Jan Kofroň

Studijní program: Informatika, programování

**2007**

Chtěl bych na tomto místě poděkovat panu RNDr. Janu Kofroňovi za vedení mé práce a cenné rady a poznámky. Také bych rád vyjádřil dík své rodině za podporu a trpělivost.

Prohlašuji, že jsem svou bakalářskou práci napsal(a) samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

V Praze dne 7. srpna 2007 Ondřej Dušek

# Obsah

# Abstrakt

*Název práce:* BashCommander

*Autor:* Ondřej Dušek

*Katedra (ústav):* Katedra softwarového inženýrství

*Vedoucí bakalářské práce:* RNDr. Jan Kofroň

*e-mail vedoucího:* jan.kofron@mff.cuni.cz

*Abstrakt:* Práce se zabývá implementací programu BashCommander, sloužícího k efektivní správě souborů ve Windows s použitím textových příkazů. Jde o dvoupanelový souborový manažer, těsně provázaný s konzolí pro textové příkazy, kde se používá syntaxe jazyka Bash. Součástí implementace je grafické uživatelské rozhraní se souborovým manažerem a terminálovým programem pro zobrazování konzole, interpret jazyka Bash pro Windows a několik jednoduchých programů pro práci se soubory, známých z operačních systémů typu Unix. Program není prostředí pro běh unixových aplikací, ani takové prostředí nevyžaduje – jako příkazy je možné spouštět libovolné nativní aplikace pro Windows. Co se týče implementace jazyka Bash a jeho interních příkazů, cílem je co největší přiblížení se originální verzi, i když s některými omezeními. Podporováno je i spouštění souborů se skripty v tomto jazyce, aplikace se snaží zajistit jejich snadnou přenositelnost. Součástí práce je dále analýza problémů implementace, diskuse jejich řešení a srovnání s existujícími interprety jazyka Bash pro Windows.

*Klíčová slova:* souborový manažer, Bash, Windows, terminálový program

*Title:* BashCommander

*Author:* Ondřej Dušek

*Department:* Department of Software Engineering

*Supervisor:* RNDr. Jan Kofroň

*Supervisor's e-mail address:* jan.kofron@mff.cuni.cz

*Abstract:* The thesis is concerned with the implementation of the BashCommander application, which serves to effectively manage files using text commands in the Windows operating system environment. It combines a two-panel file manager with a console for text commands, using Bash language syntax. The implementation covers a graphical user interface with the file manager and terminal program (to display the console window), Bash language interpreter for Windows and several simple file operating programs, well-known from the Unix-like operating systems. The program itself is not an execution environment for Unix applications, nor does it require one to run. As to the Bash language and its internal commands' implementation, the goal is to achieve as close compatibility with the original version as possible, although limited in some ways. The execution of script files written in this language is also supported, the application aims to provide their easy portability. The program is designed for Windows 2000 and newer. The thesis also includes an analysis of the implementation problems, discussion of possible solutions and a comparison of BashCommander with existing Bash interpreters for Windows.

*Keywords:* file manager, Bash, Windows, terminal program

# Text práce

## 1. Úvod

### 1. Motivace

Při práci se soubory má na většině současných operačních systémů uživatel několik možností, jaký způsob zvolit. Pravděpodobně nejpoužívanějším, ale z hlediska rychlosti a efektivity nepříliš výkonným je použití běžných souborových manažerů, jako např. Průzkumník Windows, kde se předpokládají hlavně akce prováděné za pomoci myši a přepínání mezi několika otevřenými okny programu.

Druhou, efektivnější možnost představují dvoupanelové souborové manažery („commandery“), známé již z prostředí MS-DOS a používané i na moderních systémech (Norton Commander, Total Commander). Umožňují rychlejší práci s více soubory, podporují různé klávesové zkratky a některé z nich mají mnoho pokročilejších funkcí, jejich hlavní předností je však důraz na ovládání klávesnicí a zobrazení ve dvou panelech.

Další možnost je využití příkazové řádky – shellu. Zjednodušuje uživateli opakované úkony pomocí skriptů a dávkových souborů. Naopak úplně základní akce (jako výpis seznamu souborů v adresáři) jsou oproti předchozím variantám o trochu těžkopádnější. Navíc na v dnešní době nejvíce používaném operačním systému, Windows (nejčastěji stále ve verzích XP a 2000), je i automatizace dávkovými soubory (.bat) poněkud nepohodlná. Operační systémy typu Unix nabízejí výběr několika shellů a tedy i skriptovacích jazyků, z nichž nejběžnější a poměrně přístupný a účinný je Bash[1]. Tato bakalářská práce se bude věnovat jeho začlenění do systému Windows a spojení s druhým efektivním způsobem práce se soubory – dvoupanelovým souborovým manažerem.

### 2. Zadání práce

Předmětem práce je vytvořit program pro efektivní správu souborů a používání textových příkazů pro Windows. Půjde o dvoupanelový souborový manažer, těsně provázaný s konzolí pro textové příkazy, používající syntaxi Unixového Shellu. Při práci s programem bude uživatel moci rychle přepínat mezi ovládáním panelů s výpisem souborů v adresářích a psaním do textového okna konzole. Změna pracovního adresáře v panelovém výpisu se odrazí i v konzolovém okně.

Výstup spouštěných textových aplikací bude možné zobrazit v konzoli. Konzolových oken program umožní otevřít několik a přepínat mezi nimi v panelech, podobně jako u linuxových terminálových programů. Pro textové okno konzole i výpis souborů bude možné nastavit font, barvy a jejich rozložení a toto nastavení uložit.

Konzolová část aplikace se pokusí svým ovládáním co nejvíce přiblížit ovládání Unixového Bashe - hlavně syntaxí příkazů a jazykových konstrukcí, ale i např. automatickým doplňováním názvů souborů nebo nastavením promptu. Integrovány budou i základní Unixové příkazy pro práci se soubory (cd, cp, ls, ln, rm, mv apod.), nepůjde ale o prostředí pro běh Unixových aplikací. Jako příkazy bude možné spouštět nativní aplikace pro Windows. Program bude podporovat i zpracování souborů s bashovskými skripty.

Projekt bude psán v jazyce C++ s využitím funkcí Win32-API, bude určen pro operační systémy Windows 2000 a XP.

# 3. Cíl práce

Cílem práce je vytvořit program BashCommander pokud možno podle definovaného zadání, s tím, že hlavní důraz je kladen na spojení a spolupráci jednotlivých částí – konzole a souborového manažeru; důležitá je také co nejlepší kompatibilita s Unixovým Bashem. Hlavním přínosem programu má být (pokud možno co nejhladší) spojení konzolových operací s prací v souborovém manažeru a tím zefektivnění operací se soubory ve Windows.

Program by měl být také použitelný jako o něco pohodlnější náhrada příkazové řádky Windows, díky možnosti otevřít více oken současně a větším možnostem nastavení a přizpůsobení uživateli. Aplikace by dále měla být využitelná pro samostatné spouštění skriptů (přímo z prostředí OS).

V práci nejde o implementaci Unixového prostředí na platformě Windows (jak toto provádí např. projekt Cygwin[2]) – BashCommander bude co nejvíce provázán se systémem Windows a jeho nastavením – jako např. domovské adresáře uživatelů, nastavení cesty ke spustitelným souborům apod. Navíc z něj bude možné spouštět všechny nativní aplikace pro Windows (ať už konzolové, nebo grafické), podobně jako je to možné v Unixových systémech s grafickou nadstavbou (např. KDE a Bash běžící pod terminálovým programem Konsole[3]).

Také se nejedná jen o přenesení Bashe pod Windows, i když právě tato část práce je z hlediska naprogramování časově nejnáročnější. Práce bude obsahovat i hlavní Unixové programy pro práci se soubory a hlavně terminálový program kombinovaný se souborovým manažerem – po tom se nepožadují příliš pokročilé funkce, ostatně např. přenos souborů po FTP nebo přístup do komprimovaných archivů, běžné funkce souborových manažerů, nevyhovují příliš spolupráci s konzolovou částí.

V programu budou zahrnuty jen příkazy z Unixu pro práci se soubory, ne programy pro automatizovanou úpravu textu. Tyto mohou být pro aplikaci cenným přínosem, díky provázanosti s nativními programy pro Windows je možné (a vhodné) použít dohromady s BashCommanderem jejich implementaci odjinud, např. GnuWin32[4].

# 4. Volba jazyka a platformy

Pro práci byl zvolen programovací jazyk C++, vzhledem ke spojení rychlosti výsledného kódu s možností objektového přístupu k problému. Řešení navíc nutně vyžaduje použití relativně nízkoúrovňových API-funkcí, které jsou z prostředí vyšších jazyků jako Java nebo C# dostupné jen obtížně. Protože byl od začátku jako cílový operační systém vybrán Windows (a přenesení Bashe pod Windows je jedním z cílů práce), není třeba uvažovat přenositelnost na jiné systémy. Windows verzí NT4, 95, 98 a ME a starších jsou v dnešní době i Microsoftem považovány za již překonané a používají se pouze na velmi malém procentu všech počítačů s Windows, proto je program určen pouze pro novější verze a profituje z využití několika nových funkcí Win32-API, které jsou dostupné jen pro Windows 2000 a novější.

Pro systémová volání je použito čisté Win32-API bez rozšíření MFC jednak vzhledem k předchozím zkušenostem autora, jednak kvůli otázce rychlosti kódu, nutnosti použití speciálních dynamických knihoven a umožnění kompilace zdrojových kódů ve volně dostupné verzi Microsoft Visual C++ (přestože k vývoji bylo použito plné MS Visual Studio .NET).

# 2. Analýza problému

## 1. Návrh

Při uvažování rozdělení celé aplikace se objevuje několik otázek. První z nich je propojení terminálového programu, souborového manažeru, samotného shellu a funkcí pro práci se soubory – je možné vše integrovat do jednoho spustitelného modulu, nebo naopak funkce více či méně striktně oddělit. V takovém případě je dále nutné řešit způsob a míru komunikace mezi takto vzniklými součástmi. Na druhou stranu sloučení všech součástí částečně nebo zcela zamezí jejich oddělenému využití zvenčí, což je dost výrazná nevýhoda.

Bez ohledu na GUI aplikace a funkce pro práci se soubory je nutné dále navrhnout vnitřní stavbu samotného shellu, který je zřejmě technicky nejsložitější součástí aplikace. Jde např. o problém dělení funkcí pro čtení a zpracování příkazů (a všechny operace které s nimi shell musí provést), případně jejich uchovávání s ohledem na znovupoužití. Řešení vyžaduje i způsob, jakým shell operuje s objekty operačního systému – např. rourami nebo procesy, tedy i způsob spouštění jiných programů, případně čtení jejich výstupu. V práci se uvažuje i spouštění shellu sebou samým, jehož efektu lze dosáhnout i jinak než formálně opětovným spuštěním vlastního modulu.

Podstatnou otázkou je také ovládání GUI aplikace, podle cíle práce nutně zaměřené hlavně na práci s klávesnicí. Je třeba vyřešit jednak převádění stisků kláves na znaky v konzolovém okně (vč. znaků mimo základní anglickou abecedu) a jednak jejich oddělení od klávesových zkratek pro práci s GUI programu. Také spolupráce výpisu seznamu souborů v souborovém manažeru s konzolovým oknem musí být co nejefektivnější a pro uživatele co nejbezbolestnější, problém souborového manažeru otvírá i další otázky jeho integrace s Bashem.

Vzhledem ke zvolené platformě, tedy čistému Win32-API, a programovacímu jazyku C++ je nutné i řešení provázání objektů jazyka C++ s navenek neobjektovým přístupem funkcí operačního systému designovaných pro použití z jazyka C.

Podstatnou otázkou je také rozsah podpory interních příkazů a vlastností Bashe – jde hlavně o uvážení frekvence jejich používání, užitečnosti včetně vztahu k systému Windows a jejich smyslu v něm a také náročnosti jejich naprogramování s prostředky Win32-API.

## 2. Technické problémy implementace

Většina technických překážek, které se objevily při vývoji aplikace BashCommander, měla původ v rozdílnosti systémů typu Unix, z kterých pochází Bash a jeho způsob práce, a systémů rodiny Windows – cílové platformy.

Standardní knihovny funkcí, nacházející se na většině počítačů s operačním systémem Linux a jemu podobnými, jsou mnohem lépe uzpůsobeny pro práci s konzolovými aplikacemi než API systémů Windows. To je mnohem více zaměřeno na grafické uživatelské rozhraní (které je v Unixových systémech řešeno jako nadstavba, ale s konzolí provázaná). Windows poskytuje programátorům funkce pro práci s konzolí (včetně změny velikosti okna, barev apod.) a to funkce dvou typů – tzv. vysokoúrovňové a nízkoúrovňové, první pro běžné operace (s relativně snadným použitím), druhé pro přímý přístup k vstupu a výstupu na konzoli (a tedy docela těžkopádné). Nízkoúrovňový vstup je řešen jako proud událostí myši, klávesnice apod., které je nutné rozlišovat a dále z nich zjišťovat, jaký mají vliv na práci programu; výstup je ve Win32-API řešen pomocí bufferu, který obsahuje dvourozměrné pole znaků spolu s jejich vlastnostmi (barvy). Vysokoúrovňové funkce bohužel neumožňují příliš mnoho pokročilých operací (např. čtení s čekáním na daný znak na vstupu nebo time-out) a jsou dost jednostranně zaměřeny na práci se standardní příkazovou řádkou systému, takže se pro použití v práci většinou nehodí. Některé operace se navíc ve Windows neprovádí vůbec (přeformátování textu při změně velikosti konzolového okna).

Pravděpodobně největším problémem konzolových oken ve Windows je praktická nemožnost detekce změny na výstupu – pokud aplikace spuštěná zevnitř konzolového programu něco vypíše do této konzole, vnější program se o změně nedozví a nemá možnost ji nijak propagovat ven, tedy ani do GUI terminálového programu (viz [5]).

V diskutovaných operačních systémech je také přístup k vstupu a výstupu do a ze souborů pojat rozdílně. V Unixu je v podstatě libovolné zařízení označeno jako soubor (vč. konzole) a je udržován seznam tzv. file-descriptorů otevřených souborů, který se předává „rodičovským“ programem spuštěným „potomkům“. Naproti tomu ve Windows se pro přístup k souborům (ale i jiným objektům, jejich druh je ovšem mnohem více rozlišován) používá speciálního typu `HANDLE` – jde vlastně o speciální číslo otevřeného objektu, které je možné jednak předávat spouštěným „potomkům“ a jednak posílat jiným procesům dalšími způsoby.

Problém s `HANDLE` a dalšími operacemi se může vyskytnout při pokusu o kontrolu spouštěných aplikací – ta ve Windows na rozdíl od Unixových systémů prakticky neexistuje, pokud jeden proces spustí jiný, je ten naprosto nezávislý a např. i po skončení původního běží dál. Pro komunikaci mezi procesy navíc nelze využít v Unixu hojně používaných signálů (oproti unixovým 32 druhům je prakticky použitelný jen jediný) a většinu jimi prováděných operací bez předchozí „domluvy“ aplikací nelze řešit ani jinou cestou. Pro Bash implementovaný v BashCommanderu z toho plynou některá omezení.

Komplikaci představuje také jiný přístup k přístupovým právům souborů v obou systémech. V Unixu lze standardně počítat s přístupovými právy pro uživatele, skupinu a ostatní, ve Windows nic takového neexistuje a stojí proti sobě dva systémy souborů – FAT32 bez jakýchkoliv přístupových práv a NTFS s mnohem složitější strukturou tzv. „access-control-listů“.

V GUI části se navíc objevuje další problém – ovládání klávesnicí. Nutně se celé okno programu musí skládat z několika dílčích oken (např. konzolový výpis, hlavní okno, panel nástrojů), ve Windows však jen jedno z nich může dostávat od systému informace o stiscích kláves (má tzv. „keyboard-focus“). Toto oprávnění lze mezi okny předávat.

# 3. Analýza možností řešení

Z hlediska celkového rozdělení aplikace lze uvažovat dva směry – k oddělení jednotlivých funkčností do různých spustitelných modulů, versus sloučení všech částí do jedné. Druhá alternativa má výhodu v tom, že odpadá řešení komunikace mezi částmi pomocí prostředků operačního systému a vše je možné řešit přímým voláním funkcí a předáváním parametrů. Problém je ovšem to, že některé funkce vyžadují GUI program a některé naopak konzolovou aplikaci – např. pouhé kopírování souborů nebo výpis adresáře, případně zpracování souboru skriptu žádné GUI nepotřebuje, pro interaktivní práci je ale nezbytné. Ve Windows ale nelze uspokojivě napsat aplikaci, která by vyhovovala oběma – lze vytvořit buď konzolovou aplikaci, která sice může vytvořit okno, ale její konzole zůstane viditelná (lze ji schovat, ale vždy bude nějakou dobu vidět), nebo GUI program, který se ale nebude schopen připojit k již existujícímu konzolovému oknu, ze kterého bude spouštěn a vždy bude mít jen možnost vytvořit novou konzoli – programátory Microsoftu navrhované řešení tohoto (a je tak upraveno i např. MS Visual Studio) je použití dvou samostatných modulů (detaily viz [6]). Sloučení všech funkcí do jednoho modulu navíc komplikuje použití jednotlivých součástí zvenčí (např. příkazu `cp` jako samostatného programu, nebo samotného Bashe jako interpret skriptů) – bylo by nutné použít předávání tajného parametru, což je pro vnější kompatibilitu s unixovými programy nemožné, případně posílání speciální zprávy systému, což zas pro uživatele při spouštění není proveditelné.

Druhou možností je oddělení všech funkcí – souborového manažeru, terminálového programu, Bashe i všech unixových pomocných programů. To ovšem přináší nemalé nároky na komunikaci mezi takto vzniklými moduly (především GUI aplikacemi a Bashem) – celé řešení je ale technicky s pomocí prostředků operačního systému proveditelné. Implementačně jednodušší (odpadá dost komunikace navíc a další problémy nevznikají) se jeví sloučení všech GUI-funkcí do jednoho modulu. Sloučení některých GUI a konzolových částí vytváří výše popisovaný problém, sloučení všech konzolových částí opět zamezuje použití unixových příkazů samostatně, proto není vhodné.

V kontextu samotného Bashe existují dvě alternativy přístupu ke spouštění dalších programů (hlavně sebe sama) a vytváření pipe-lines (řada programů spojených rourami, každý čte výstup předchozího a zapisuje do vstupu následujícího), totiž původní unixová, spouštění dalšího Bashe pro výstup z vnořených příkazů a pipe-lines, a použití nových prostředků – hlavně vláken. První možnost je zřejmě kompatibilnější s unixovou verzí Bashe, ale ve Windows naráží na problém neexistence funkce `fork` (klonování již spuštěného programu), která je hlavně pro rychlost tohoto řešení klíčová. Alternativa – použití vláken –

vyžaduje dodatečnou synchronizaci, kterou ale lze pomocí prostředků OS zajistit. Navíc otvírá možnost větší integrace proměnného prostředí Bashe (které se kvůli mnoha spouštěným „potomkům“ v Unixu tříští – např. při čtení z pipe-line příkazem while), což je sice vlastnost odlišná od unixové verze, ale pro programátora skriptů v naprosté většině případů příznivá.

Techničtějším problémem je přístup ke konzolovým funkcím Windows – je sice možné je úplně ignorovat a implementovat vlastní pseudokonzoli s pomocí rour, ale tím dojde ke ztrátě dalších informací (jako o barvách a stiscích kláves, které se nepřevádějí na platné znaky), takže pro jednoduché programy by byl takový systém vhodnější, ale složitějším, které používají nízkoúrovňové konzolové funkce, by uzavíral cestu. Další alternativou je použití nízkoúrovňového konzolového API Windows – to sice představuje naprogramovat znovu vyšší funkce konzole, protože pro běžné použití v programu je nutný jednotný přístup ke konzolím, rourám i souborům, zato dovoluje spouštět libovolné programy a provádět s konzolí i složitější operace (vyžadované např. Bashovskou funkcí `read`). Použití vysokoúrovňových funkcí na konzoli je implementačně nenáročné a pro spouštění libovolných programů vyhovující, ale omezuje to některé funkce Bashe (např. automatické doplňování názvů souborů).

Při řešení problému s přístupem k souborům také existují dva postupy – buď emulace způsobu práce se soubory v Unixu (plné řešení včetně různých pseudosouborů atd. nelze než přenechat projektům typu Cygwin, ale emulace file-descriptorů je proveditelná), nebo omezení se na standardní vstupy a výstupy (ty jsou naprosto nezbytné) a ponechání zbytku podle filosofie Windows. V otázce práv k souborům nelze než přistoupit na určitý stupeň předstírání, pokud je vyžadována kompatibilita s Bashem – přímo zasahovat do složitého systému práv NTFS s naprosto jiným způsobem fungování by nemělo smysl (a mohlo by způsobovat problémy), navíc by operace stejně nefungovaly na stále používaném souborovém systému FAT32.

Detekce změn ve výstupu na konzoli je, jak už bylo popsáno, ve Windows standardně neproveditelná, řešení proto musí problém obcházet. Jednou možností je pravidelné překreslování obrazovky bez ohledu na změny, což bohužel nemusí zachytit všechny změny a navíc jde o plýtvání hardwarovými prostředky při špatné konfiguraci obnovovací frekvence. Kromě toho ale zbývá jen přímá fyzická integrace konzolového okna do GUI aplikace, což naopak přináší komplikace při změně jeho pozice a velikosti a předávání příkazů a klávesových zkratek.

Problém kontroly aplikací spuštěných z prostředí Bashe lze řešit jen částečně – takový program je možné ukončit při ukončování samotného Bashe, ten ale už nemá přehled o jím spuštěných programech – alespoň ne běžnými prostředky; navíc takové ukončení programu je považováno za nestandardní a nemusí dojít k uvolnění systémových prostředků tímto programem alokovaných (viz `TerminateProcess` v [7]).

V souborovém manažeru je nutné řešit problematiku přepínání mezi konzolovým oknem a dvěma panely výpisu seznamu souborů. Pokud by uživatel

mezi oběma částmi přepínal (zřejmě myší nebo klávesovou zkratkou), znamenalo by to sice komplikovanější ovládání, ale tyto dvě části by mohly reagovat mnohem samostatněji. Naopak lze celou věc pro rychlejší práci implementovat úplně bez přepínání kontextu – uživatel by si musel zvyknout na více různých klávesových zkratek, protože se pro práci v konzoli a v souborovém výpise musí používat různé, ale ve výsledku by práce s takovým oknem byla snadnější. Celé toto řešení navíc skrývá problém s přenášením informací z klávesnice a implementací výpisu souborů (při použití standardního ovládacího prvku Windows – „listboxu“).

Problém s informacemi o stiscích kláves lze řešit zřejmě přeposíláním přímo informací o stiscích kláves, což ale není v případě použití standardních dialogových ovládacích prvků možné – nedodávají totiž úplné informace, jen notifikace stisků kláves (Windows používají pro toto systémové zprávy, hlavní nástroj komunikace systému s aplikacemi i aplikací mezi sebou, a to 3 druhy: pro stisk klávesy, puštění klávesy a napsání znaku, přičemž třetí typ správ se vyrábí (nepřímo) z prvních dvou pomocí systémového volání. Přímý převod z informací o stiscích kláves na napsané znaky není možný, stejně tak přeposílání stisků kláves mezi aplikacemi (jen znaky). Navíc není možné ze znaků nebo kódů kláves jednoduše syntetizovat zprávy o stiscích kláves – nejde to v případě stisku „modifikačních“ kláves jako Ctrl nebo Shift.). I při použití standardních ovládacích prvků lze problém vyřešit přepínáním „keyboard-focusu“, je ale nutné vyřešit, které okno bude všechny stisky kláves dostávat – v úvahu připadá buď hlavní okno GUI, nebo terminálové okno aktuálně zobrazovaného shellu; v obou řešeních není velký rozdíl, snad jen to, že terminálové okno má lepší přístup k shellu a hlavní okno zas snadněji rozhoduje o (globálních) klávesových zkratkách.

# 4. Zvolené postupy – hlavní struktura

Z možností rozdělení jednotlivých funkcí aplikace do samostatných modulů byla vybrána tato: celá GUI část je jeden spustitelný program, Bash je druhý jako samostatná konzolová aplikace a unixové příkazy (které nejsou v Bashi integrované) jsou každý samostatný program. Tím odpadá složitá komunikace mezi součástmi GUI a je umožněno samostatné použití Bashe nebo unixových podpůrných příkazů. Komunikaci mezi GUI a shellem ale bylo nutné vyřešit – pro cíl práce se nejlépe hodilo použití systémových zpráv Windows v kombinaci se sdílenou pamětí. GUI část programu standardně zpracovává systémové zprávy Windows jako každá běžná grafická aplikace na tomto systému, mezi nimi ale může dostávat i speciální od Bashe, v Bashi je na příjem zpráv od GUI dedikováno jedno speciální vlákno.

Bylo by možné použít i roury, ale bez systémových zpráv by se potom řešení nevyhnulo tajným parametrům spouštění aplikace; systémové zprávy Windows díky své rychlosti a neviditelnosti pro uživatele představují nejvýhodnější řešení. Takto sice při spouštění Bashe samostatně stále čeká jedno vlákno programu na příchozí inicializační systémovou zprávu od GUI, ale díky tomu je uspané a nespotřebovává systémové prostředky.

Jak už bylo zmíněno na počátku, bylo nutné nějak propojit Win32-API s objektovým modelem aplikace. Toto není příliš problém Bashe a textových příkazů, kde nejsou jeho funkce volány příliš často a povětšinou nejsou svázány s nějakým konkrétním objektem, jediný složitější případ je spojení `HANDLE` pro vstup nebo výstup s objektem a funkce na něm, detaily viz níže. V GUI byl nejčastější problém, spojení okna s objektem s ním spojeným (vč. všech potřebných proměnných), realizován pomocí programátorem definovaného pointeru, který je díky funkci Win32-API možné uložit ke každému vytvářenému oknu, a které okno dostane v parametrech vytvářející funkce (detaily viz `SetWindowLongPtr` v [7]). Procedura standardního tvaru, která přijímá systémovou zprávy pro dané okno, potom hodnotu tohoto pointeru zjistí a použije jako příslušný objekt, kterému předá zprávu. Během zpracování zprávy se tak volají už funkce na konkrétním objektu.

Schéma Bashe a jeho rozdělení na různé části podle funkce je zhruba následující: hlavní objekt typu `Winshell` zprostředkovává načtení i spouštění všech příkazů. Obsahuje objekt `Parser`, který je zodpovědný za správné dělení příkazů a jejich jednotlivých typů – ze vstupu vyrábí různé objekty všelijak strukturovaných příkazů podle jejich typů (např. `while`, `for`, jednoduchý příkaz a ostatní druhy příkazů Bashe mají své C++ třídy). Během práce uchovává i pouze částečně zpracovaný vstup. Hlavní objekt od parseru dostává objekty příkazů, které postupně provádí a následně shromažďuje pro znovupoužití. Mohou být uloženy ve speciálních polích, která implementují funkce pro jejich provedení najednou (případně částečné) – takto si je uchovává jak hlavní objekt, tak jednotlivé strukturované nadřízené příkazy. Detaily práce s nimi viz oddíl 3.4. Celé proměnné prostředí shellu je obsaženo v objektu třídy `Env`, příkazy a ostatní objekty celé aplikace na něj uchovávají ukazatel. Interní příkazy shellu jsou implementovány jako statické funkce pro tento účel vytvořené třídy.

Vzhledem k rychlosti výsledného programu a již popisované možnosti lepší integrace proměnného prostředí bylo před spouštěním stále nových kopií Bashe pro vnořené příkazy a roury upřednostněno použití vláken. Bash tedy, pokud není přímo voláno provádění skriptu v externím souboru, nevytváří svoje kopie – proměnné prostředí zůstává zachováno společné pro všechny příkazy, které nejsou spouštěny na pozadí (to se týká i všech příkazů v pipe-line kromě posledního). To znamená, že ostatní příkazy vidí změny prostředí provedené i ve vnořených příkazech a posledních příkazech v pipe-line – s unixovým Bashem toto sice není kompatibilní, ale v praxi k použití vhodnější. Synchronizace s takto vytvářenými vlákny nemusí být příliš složitá, většinou se jedná o prosté čekání na jejich skončení.

Práce s konzolovým oknem byla vyřešena pomocí nízkoúrovňového konzolového Win32-API. Funkce pro běžné čtení z konzole musely být implementovány znova, aby obsahovaly pokročilejší možnosti nastavení, využívané automatickým doplňováním názvů souborů nebo historií příkazů. S `HANDLE` na vstup nebo výstup (obecný) je svázaný objekt, který poskytuje obecné funkce čtení nebo zápisu. Tyto funkce pak rozlišují typ daného vstupu nebo

výstupu. Další detaily operací s konzolovým oknem a vstupem a výstupem viz 3.3. Třídy pro vstup a výstup jsou používány jak v Bashi, tak v jednotlivých unixových příkazech.

Simulace file-descriptorů jako v Unixu do Bashe zavedena nebyla, protože nativní aplikace Windows, které jím budou nejčastěji spouštěny, by tuto možnost neuměly využít – zbylo by upotřebení uvnitř Bashe samotného, které ale není příliš časté; pravděpodobně by nevyvážilo náročnost implementace. Proto se operace přesměrování vstupů a výstupů musí omezit na soubory, standardní vstup a standardní a chybový výstup.

Stejně tak práva přístupu k souborům jsou pouze simulována – tedy lze změnit práva souboru, případně standardní masku práv souborů (`umask`), na realitu to ale nemá efekt. Jediné, co příkaz `chmod` a příkazy jako `ls` berou v potaz, je standardní atribut souborů ve Windows „jen pro čtení“ – k takovým souborům se chovají stejně, jako kdyby na soubor žádná ze skupin uživatelů neměla právo zápisu. Tento atribut lze také příkazem `chmod` měnit.

Bash také neřeší problém ukončení jím spuštěných aplikací – při běžné práci toto není příliš problém (konzolové aplikace je ve Windows typicky možné ukončit pomocí signálu `Ctrl-C`), kromě toho by i při hlídání vytvořených procesů vznikaly již zmiňované potíže pro programy z nich spouštěné. Proto není implementována ani kontrola procesů spouštěných z Bashe „na pozadí“.

S provázáním GUI a Bashe je spojená otázka zjišťování změny na výstupu konzole – to je provedeno již nastíněným automatickým obnovováním stavu zobrazení – zde na frekvenci max. cca 20x/sec., každé obnovení musí mít od GUI potvrzené překreslení (takže nehrozí zahlcení); takto se navíc obnovuje jen to okno shellu, které je zrovna viditelné (aktivní, „topmost“), takže nejde o příliš hardwarově náročný proces. Oproti jiným podobným aplikacím (viz *UNIX pseudoterminal emulation frustrations* v [5]) navíc běžně nemůže dojít ke ztrátě zobrazovaných dat příliš rychlým výpisem – je totiž možné pomocí systémových zpráv vyvolat scrollování přímo v bufferu konzolové aplikace (který by měl být pro toto dost velký), zobrazení je přeposíláno do GUI.

„Keyboard-focus“ v GUI, tedy příjem zpráv o stisknutých klávesách, je za každých okolností (kromě případu, kdy nejsou otevřené žádné panely shellů) převáděn na aktuální okno shellu – i pokud je toto okno zobrazováno jako součást commanderu. Může tak jednodušeji posílat vstup svému shellu, pro zjišťování klávesových zkratek volá funkce na objektu hlavního okna. Pro práci v commanderu byla zvolena varianta bez přepínání kontextu, tedy vstup z klávesnice přijímá výhradně okno shellu a případně ho přeposílá právě aktivnímu panelu s výpisem souborů. To vynucuje použití více klávesových zkratek (které se nesmějí krýt), ale zase je možné nastavit i ovládání pohybu mezi soubory ve výpisu (což by v případě, že by tento standardní ovládací prvek skutečně přijímal vstup z klávesnice, nebylo možné). Práce s commanderem bez přepínání kontextu navíc vyžaduje rozlišování mezi spouštění aplikací z výpisu (stiskem klávesy `Enter`) a odřádkováním na konzoli – to je řešeno pomocí příkazů, které

konzolová část posílá GUI (více viz 3.1). Pokud v Bashové části běží příkaz, nelze spouštět další (ani přecházet mezi adresáři).

Commanderové části byla pro větší provázání s Bashem dána ještě jedna vlastnost – samotný souborový manažer umí jen vypisovat soubory a spouštět příkazy v Bashi – pro ty lze ale nastavit klávesové zkratky a do jejich textu je možné vložit názvy vybraných souborů z panelového výpisu, případně jméno adresáře nebo i požadovat před jejich spuštěním vstup od uživatele a ten v nich pak použít. Commander jen vloží do výsledného příkazu aktuální hodnoty požadovaných „proměnných“ (viz uživatelská dokumentace) a předá Bashi příkaz k jeho spuštění. Takto jde vytvořit (např. s pomocí v projektu obsažených unixových programů) všechny běžné funkce souborových manažerů – kopírování, přepisování, vytvoření adresáře apod., přičemž možnosti jejich nastavení se neomezují zdaleka jen na toto.

Pokud jde o podporu všech vlastností Bashe, snažil jsem se o co největší a zároveň smysluplný rozsah, který je s prostředky Win32-API a v rámci tohoto projektu dosažitelný. Jako výchozí popis vlastností sloužil referenční manuál Bashe[1], případně pro testy nepříliš podrobně zdokumentovaných funkcí GNU Bash ve verzi 3.1. V rámci podporovaných funkcí by měly být obě verze ekvivalentní. Z jazyka je možné použít veškeré konstrukce kromě substituce procesů („process substitution“), některé řídké syntaktické obraty (hlavičky funkcí bez „`()`“, přesměrování „`>&`“) také podporovány nejsou. Omezení má i podpora wildcardů a nastavení vlastností chování shellu. Vynechány byly některé nepříliš důležité (nebo ve Windows implementačně příliš náročné či nesmyslné) interní příkazy, jmenovitě: `bind`, `enable`, `local`, `logout`, `printf`, `shopt`, `type`, `typeset` and `ulimit`. Určité přepínače některých interních příkazů jsou navíc ignorovány (jde zejména o ty, které by v cílovém operačním systému neměly smysl, detailní popis viz uživatelská dokumentace). Jak už bylo výše popsáno, příkazy spuštěné na pozadí nelze nijak kontrolovat. Tím ale výčet nepodporovaných vlastností končí.

# 3. Některé detaily řešení

Tento oddíl obsahuje do detailu rozvedenou diskusi řešení problémů, z nichž některé byly již obecně probrané v předchozí sekci. Jde hlavně o komplikovanější části projektu, částečně z hlediska použití API systému Windows, částečně objektovým návrhem a integrací jazyka Bash. Technický popis všech hlavních součástí projektu je možné nalézt v přiložené programátorské dokumentaci.

## 1. Komunikace GUI a shellu

Základem komunikace mezi grafickou částí aplikace a shellem jsou, jak už bylo naznačeno, zprávy systému Windows. Obecně každá systémová zpráva má svůj identifikátor druhu a může nést až dva parametry (všechno jsou 32-bitová čísla). Ve Win-API je určitý rozsah čísel druhů zpráv ponechán k volnému upotřebení aplikačním programátorům, čehož BashCommander využívá. Celkem je v programu definováno 28 druhů zpráv pro komunikaci jednak mezi jednotlivými částmi grafické aplikace, a jednak mezi GUI a shellem. Ty slouží k navázání komunikace a potom k posílání příkazů a informování o jejich provedení. Pro předávání objemnějších zpráv je k dispozici oblast sdílené paměti.

V rámci GUI se mezi jednotlivými okny často místo volání funkcí na jim příslušných objektech (viz popis přiřazení okna k objektu v předchozí sekci) často používají právě systémové zprávy. Takto jsou např. panely informovány o aktivování (zobrazení „navrchu“) nebo hlavní okno o stisku některé klávesové zkratky. Hlavní důvodem volby tohoto postupu je vícevláknovost grafické části programu – kromě prvního vlákna, obsluhujícího hlavní okno, je po otevření každého panelu spuštěno další, které se věnuje jen zobrazování okna shellu a komunikace s ním. Použití systémových zpráv je nejjednodušší cesta k zajištění provedení požadované operace v jiném vlákně, pokud není příliš důležitá doba vykonání, což u většiny akcí programu nenastává – malé zpoždění např. u zmiňovaného stisknutí klávesové zkratky je akceptovatelné. Systémové zprávy v rámci jedné aplikace navíc mají tu výhodu, že jako jejich parametry lze předávat přímé ukazatele na různé datové struktury (např. když nově spuštěný shell požaduje přidání do seznamu panelů hlavního okna, posílá ukazatel sám na sebe). Samozřejmě, pokud nezáleží na tom, ve kterém vlákně bude akce provedena, prosté volání metod na výkonném objektu je jednodušší a tedy preferované.

Mezi běžícím procesem Bashe a GUI mají systémové zprávy několik funkcí. Jednou z nejdůležitějších je samotné navázání komunikace – po spuštění shellu grafická aplikace střídavě několik okamžiků čeká a pokouší se poslat speciální zprávu, otevírající komunikaci, nově vytvořenému procesu (v němž je hned po spuštění vyroben speciální thread pro příjem rozkazů od GUI, reprezentovaný třídou `MsgThread`, čekající právě na inicializační zprávu). Toto teoreticky může selhat (a pak dojde k vypsání chyby a otevření nového panelu není provedeno), v praxi se to však může stát jen v případě problémů se samotným operačním systémem. Navázání dialogu mezi oběma procesy je nakonec shellem

potvrzováno. Spolu s těmito zprávami se předávají `HWND` (identifikátory) cílových oken pro některé další zprávy (konzole shellu, okno panelu GUI za shell zodpovědného) a také `HANDLE` sdílené paměti.

Shell potom svému „pánu“ (master) – grafickému oknu – posílá žádosti o překreslení výstupu a změnu názvu panelu (název aktuálně spuštěného programu) a dostává potvrzení o jejich provedení (což za cenu o něco většího „provozu“ předchází zahlcení zprávami stejného typu, ke kterému v praxi bez tohoto opatření skutečně docházelo); GUI shellu (a to přímo onomu dedikovanému vláknu) předává rozkazy k vykonání všemožných akcí, např. scrollování konzolového okna, stisk některých speciálních kláves (`F10`, `TAB`, šipky apod.), zobrazení historie příkazů nebo automatické doplňování názvů souborů. Zprávy o stiscích kláves – běžných znaků – jsou posílány ve standardním systémovém tvaru `WM_CHAR` přímo konzolovému oknu shellu, takže se projeví při volání libovolných API funkcí čtení.

Automatické doplňování názvů souborů a historii příkazů kvůli způsobu implementace nelze použít při samostatném spuštění Bashe bez GUI; to ale projekt předpokládá jako většinou neinteraktivní čistě za účelem provádění skriptů, takže to nepředstavuje příliš výraznou nevýhodu. Výhodou je naopak možnost nastavení klávesové zkratky pro provedení těchto dvou akcí.

Jak jsem již předeslal, kromě systémových zpráv procesy pro předávání informací používají také sdílenou paměť (vytvořenou ve Windows běžným způsobem – mapováním fiktivního dočasného souboru do paměti). Při navazování kontaktu se shellem je vytvořena jedna stálá oblast sdílené paměti (`HANDLE` na ní se shellu posílá v inicializační zprávě), která slouží po čas celého jeho běhu. Obsahuje parametry konzolového okna (mění se přímo a na změnu se upozorňuje k tomu určenou zprávou), název právě spouštěného programu a prostor pro kopírování aktuálního vzhledu konzole. Před odesláním každé žádosti o překreslení shell překopíruje právě zobrazovaný pohled právě na toto místo. GUI odtud načítá znaky a postupně je zobrazuje. Sdílená paměť se používá ještě pro ovládání shellu ze souborového manažeru. V takovém případě je pro každý příkaz posílaný commanderem Bashi vytvořena nová oblast sdílené paměti (může se znovu použít již existující, příjemce toto ale nepozná a mapování provádí pokaždé) a do ní přepsán jeho text. Stejně se děje i při změně pracovního adresáře v commanderu.

Neobvyklým případem v komunikaci mezi GUI a shellem je vydání příkazu ke spuštění souboru vybraného ve výpisu v commanderu. Pokud uživatel stiskne `Enter` a má v souborovém manažeru vybraný nějaký řádek v seznamu, napřed se o této události dozví sám shell – jeho GUI okno má vždy privilegium přijímat vstup z klávesnice a automaticky ho (i včetně stisku této klávesy) přeposílá přímo k němu. Pouze shell navíc je schopen rozhodnout o následně prováděné akci – pokud totiž jeho vstup již obsahuje nějaký zadaný příkaz, musí se provádět ten, a když ne, je žádoucí aby commander otevřel nebo spustil soubor podle uživatelova výběru. Proto shell posílá své řídící aplikaci výsledné rozhodnutí a pokud jde o spuštění programu pod kurzorem, dostává jeho cestu a příkaz ke spuštění, v

případě změny pracovního adresáře vypadá výsledek dost podobně. V jiném případě dostává jen potvrzení přijetí původní zprávy – GUI může samo otevřít soubor dokumentu v systémem přiřazeném programu. Po dobu čekání na reakci GUI je celý proces shellu zmrazen.

## 2. Dva druhy panelů v GUI

Grafické prostředí BashCommanderu dává uživateli možnost otevřít několik panelů a rychle mezi nimi přepínat. Navíc jsou na výběr jejich dva druhy – prosté okno shellu a kombinace se souborovým manažerem. V kódu aplikace jsem pro to musel vytvořit pokud možno možno co nejnenáročnější ošetření – hlavním problémem byl vztah mezi jednotlivými okny (a jejich příslušnými C++ třídami): hlavním oknem programu, oknem shellu a oknem souborového manažeru. Hlavní okno (představované třídou `MainWin`) totiž musí udržovat seznam zobrazených panelů, nejlépe bez rozdílu druhu, spuštěný shell (třída `ShellWindow`) by se neměl chovat různě v závislosti na tom, běží-li pod commanderem nebo přímo pod hlavním oknem (jako jeden vlastní panel) a pro okno manažeru (neboli třídu `Commander`) je třeba spolupráce s oběma ostatními součástmi.

Pro vyřešení situace by bylo možné použít dvě různé třídy zobrazující okno Bashe, znamenalo by to ale problémy v případě přeposílání příkazů k nim od hlavního okna přes commander. Jako výhodnější se ukázalo využití dědičnosti pro třídy oken a přístupu k ostatním oknům přes společnou základovou třídu. Byla vytvořena obecná třída Controller pro cokoliv, co může ovládat okno shellu (hlavní okno, commander) a Shell pro cokoliv co může běžící Bash zobrazovat a kontrolovat (commander, okno shellu). V případě okna souborového manažeru se tu tedy objevuje vícenásobná dědičnost – vzhledem k hlavnímu oknu je považován za obecný panel, vzhledem k oknu shellu jde o jeho nadřízeného. Volání metod a systémové zprávy často předává dál příslušným směrem, může ale výsledky upravovat (např. v případě zjišťování velikosti konzolového okna vrací menší výšku, aby se do zbylého prostoru mohl vměstnat výpis seznamů souborů v adresářích).

Protože jsou obě základové třídy de facto abstraktní a neobsahují žádné datové položky ani těla metod a jde o „virtuální“ dědičnost, nedochází s jejím zavedením k žádným problémům. V jazycích typu Java by bylo možné použít místo základových tříd interface a výsledek by byl ekvivalentní, C++ tento syntaktický obrat nenabízí a tak je násobná dědičnost způsob, jak dosáhnout jeho efektu.

## 3. Konzolový vstup a výstup

Pro čtení z (libovolného) vstupu a výpis výstupu je v celém Bashi i dodávaných unixových příkazech používáno tříd `InputHandle` a `OutputHandle` (odvozených od základní třídy `Handle`), které tvoří nadstavbu nad standardními Windows-API funkcemi. Zvnějšku jde o obecné vstupy a výstupy, pro s těmito objekty pracující funkce tedy neznamená práce s rourami, konzolí a souborem

žádný rozdíl. Jednotlivé instance těchto tříd lze také téměř neomezeně konvertovat z a na standardní typ `HANDLE` – pro tento účel jsou v kódu napsány speciální C++ operátory. Třída `Handle` vlastně obsahuje jen onen identifikátor vstupu nebo výstupu a možnost zjištění, zda jde o konzoli či přesměrování do souboru.

Z obou odvozených typů je `OutputHandle` ten jednodušší, neboť jeho metody mohly být vytvořeny s použitím standardní API funkce `WriteFile`, která většinu potřebných kroků zajistí sama. Výstup do souborů a rour používá dle současných zvyklostí interní kódovou stránku Windows (před voláním `WriteFile` jsou do ní převedeny), pro zápis na konzoli je pro zachování znaků v kódování Unicode použito volání `WriteConsole`, které je na rozdíl od předchozího schopné se s širšími znaky vypořádat bez dalších úprav. Zápis do zvoleného `OutputHandle` se provádí voláním přetíženého operátoru `operator <<` a je tak možné používat tento typ podobně jako `ostream` z C++.

Vytvoření třídy pro čtení ze vstupu bylo poněkud složitější. Je zde nutné rozlišovat mezi různými druhy vstupu a pokud jde o konzoli, používat pouze nízkoúrovňové API společně s mnoha dalšími funkcemi pro dosažení schopností, které jsou potřebné pro funkce ve spolupráci s GUI (doplňování názvů souborů, historie příkazů) a interní příkaz Bashe `read`. Při čtení se pak střídavě čeká na událost na vstupu (konzoli nebo rouře, pro čtení ze souboru toto odpadá) a volá pomocná funkce, která dodá požadovaný (nebo menší) počet znaků. Čtení z konzole je zajišťováno nízkoúrovňovou funkcí `ReadConsoleInput`, kdy se obdržené informace o stiscích kláves následně převádějí na znaky a pohyb kurzoru.

Protože některé funkce požadují vrácení výsledku ihned po načtení určeného znaku, je nutné obdržený text kontrolovat na jeho výskyt a zkracovat – jeho zbytek se uloží do vnitřního bufferu a je odtud vrácen při příštím čtení. Tento zásobník se navíc musí měnit stejně pro všechny instance InputHandle se stejným interním HANDLE, aby byla zachována koherence načtených dat. To je řešeno použitím zjednodušených počítaných odkazů při kopírování objektů této třídy. Problém by samozřejmě vyřešilo i prosté načítání po jednom znaku, to ale není příliš výhodné z hlediska rychlosti – volání funkcí operačního systému vždy představuje zdržení.

Při čekání na vstup se může aplikovat i (průběžně přepočítávaný) time-out, potřebný pro zajištění jedné vlastnosti příkazu `read`. Samozřejmě je možné nastavit i horní limit na počet načtených znaků. Kvůli funkcím vyžadovaným GUI lze také vnitřní buffer již načtených znaků (které ještě nevyhovují podmínkám pro návrat z funkce, a čeká se tedy na další) měnit – např. automatické dokončování názvů souborů si zjistí jeho obsah, vezme poslední napsané slovo a snaží se ho dokončit podle souborů v daném adresáři, pokud uspěje, dopíše konec bufferu a ten vrátí zpět. Protože jsou příkazy předané z GUI prováděny v jiném threadu než samotná práce shellu, všechny operace s vnitřním bufferem jsou synchronizovány s pomocí „kritické sekce“, jednoho z k tomu určených prostředků operačního systému.

Zobrazování aktuálního obsahu bufferu konzole bylo také nutné řešit – nízkoúrovňové načítání nevypisuje napsané znaky automaticky. Kromě toho by po změně obsahu celého bufferu také docházelo k problémům. Proto si `InputHandle` musí pamatovat pozici kurzoru v okamžiku, kdy čtení začalo, a od ní vypisovat napsané znaky (to se provádí jednoduše pomocí funkce `WriteConsole`, používá se standardní výstup programu bez ohledu na jeho přesměrování v rámci příkazů Bashe). Větší problém je aktuální pozice kurzoru – tu je nutné podle šířky konzole a počtu stisků klávesy `Enter` dopočítat. Vnitřně je totiž pozice kurzoru reprezentována jen jednorozměrně, což usnadňuje přecházení po napsaném vstupu, který bylo nutné zobrazit do více řádků (pozice je upravována na základě obdržených informací o stiscích kláves šipek, `Home`, `End` i samotných znaků).

Pro příjem příkazů ze souborového manažeru je v `InputHandle` implementována ještě jedna funkce – kromě samotného vstupu je možné čekat ještě na určenou událost (synchronizční objekt systému), a pokud ta nastane, přerušit čtení a vyvolat speciální výjimku. Načítání vstupu v hlavním objektu Bashe (a nikde jinde, ne např. při přesměrování výstupů příkazů apod.) takto nastaví před čtením tuto událost a v případném ošetření vzniklé výjimky provede commanderem požadovaný příkaz. Tak je zajištěno jeho synchronní vykonání – uživatel během něj nemůže spouštět další příkazy (které by způsobovaly nesrovnalosti na výstupu nebo problémy se synchronizací).

# 4. Parser a příkazy v shellu

Parser (reprezentovaný stejnojmennou třídou) je jednou z nejdůležitějších součástí Bashe – stará se o kompletní převedení vstupu na jednotlivé (i strukturované) příkazy, vynechává jen nahrazování prováděné po spuštění příkazu a také dělení vnořených příkazů, aritmetiky a testů (o což se stará jiná třída nebo i jiný `Parser`). Vstup, dodávaný do něj po řádkách, se dělí podle metaznaků shellu, přičemž jejich význam je dále brán v potaz (díky několika příznakům, které se mohou vrstvit na zásobník). Pro zpracování složitějších jazykových konstrukcí parser operuje se zásobníkem, na který se ukládají načtené dílčí příkazy nedokončených struktur – např. všechny operace jedné `while` smyčky. K jejich uložení se používá speciálních tříd (odvozených od třídy `Control`), závislých na druhu konstrukce (odpovídají řídícím klíčovým slovům jazyka, které ji uvozují – např. `case`, `elif`, `until`). Pokud je nalezeno ukončovací slovo, parser vytvoří z obsahu vrcholu zásobníku finální objekt příkazu, který je vložen do aktuálního vrcholu zásobníku po odebrání použitých struktur. Když je po vytvoření hotového příkazu tento zásobník prázdný, může být příkaz vrácen ven k provedení (parser vrací vždy příkazy dokončené na aktuální řádce, takže jich může být i více).

Každý Bashovský příkaz je instancí nějaké třídy odvozené od (abstraktní) třídy `Command`, která poskytuje funkce společné pro všechny druhy, jako např. přesměrování výstupů. Výjimku tvoří pipe-lines a podmíněné konstrukce, kdy

každý jejich člen je reprezentován samostatným objektem; při vykonávání se na tento fakt ale bere ohled. Některé typy příkazů mohou obsahovat další – např. všechny příkazy v cyklu – a takto vytvářejí v podstatě stromovou strukturu. Na třídě `Command` je definována virtuální funkce `exec`, která příkaz provede – proto je možné spouštět postupně všechny operace cyklu bez ohledu na jejich strukturu. Všechny se také umějí převést (včetně svých vnořených) zpět na text, což slouží zejména historii příkazů. Pro veškeré operace, které potřebují pro svoje vykonání – nahrazování jmen souborů, proměnných, aritmetiku apod., inicializují jednotlivé příkazy speciální objekty (běžně jde hlavně o třídu `Expandor`) a volají jejich funkce během svého provádění. Kontrola toku pomocí Bashových `break` a `continue` je zajištěna speciálními typy C++ výjimek, které zachycují právě jen strukturované příkazy – při běhu příkazu se tak tok kódu vrátí přesně na potřebné místo.

Jako hlavní kontejner pro příkazy je používána speciální třída `CmdVect`, odvozená od standardní šablony `vector` z knihoven STL, a to jak uvnitř samotných strukturovaných příkazů, tak globálně v hlavním objektu shellu pro udržení historie příkazů. Slouží také k předávání příkazů ze vstupu po analýze parserem. Díky jejím funkcím lze příkazy jednoduše spouštět skupinově, včetně pipe-lines nebo podmíněných konstrukcí s operátory `&&` a `||`. Použití odvozenin standardních šablon obohacených o další funkce pro jednoduchou práci s danou strukturou je obecně v Bashové části programu docela časté.

# 5. Proměnné prostředí shellu

Veškeré informace o proměnných a jiných objektech, nacházejících se v prostředí běžícího shellu, jsou uloženy v (až na výjimky při spouštění příkazů na pozadí) jediné instanci třídy `Env`. Ta může kromě hodnot proměnných poskytovat i další funkce, výpisy apod. Nejde z hlediska návrhu o nejčistší řešení, ale o vzhledem k víceméně náhodné potřebě různých funkcí různými programy a nutnosti jejich centrálního uložení zřejmě nejjednodušší. V proměnném prostředí se tak nachází seznam všech proměnných (včetně speciálních jako je kód skončení posledního spuštěného příkazu), aliasů, hashů cest ke spouštěným programů, polí i uživatelem nadefinovaných funkcí (ten je implementován kontejnerem typu `multimap` s vyhledáváním podle názvů) společně se zásobníkem volání funkcí, zásobníkem veškerých přesměrování, objektem nastavení a objektem ošetření signálů.

Díky zásobníkové struktuře seznamu přesměrování je možné uchovávat informace o všech otevřených vstupech a výstupech bez jejich vzájemného přepsání a jednoduše vracet na dotaz trojici aktuálně platných standardních handles bez jejich vzájemného přepsání. Tato funkce je velice často využívaná (všechny čtecí a výpisové funkce i nastavení vstupů a výstupů pro spouštěné programy se bez ní neobejdou), proto bylo užito jednoduché „vyrovnávací paměti“ pro návratové hodnoty.

Ošetření signálů je součástí proměnného prostředí hlavně kvůli nutnosti zpřístupnění z interního příkazu `trap`. Je sice možné nastavit ošetření všech unixových signálů, reálně funguje ale jenom `SIGINT`, tedy `Ctrl-C`, společně s interními Bashovými signály (`DEBUG` apod.). Z podobného důvodu `Env` obsahuje i objekt nastavení shellu. Operace s proměnnými a poli slouží hlavně nahrazování hodnot, ostatní funkce jsou využívány povětšinou pro interní příkazy shellu, jako např. declare, readonly, hash apod., jimi prováděné nastavení ale berou v potaz všechny součásti shellu.

# 4. Hodnocení

## 1. Srovnání s jinými implementacemi Bashe ve Windows

Pokud je mi známo, neexistuje žádná aplikace, která by měla alespoň zhruba stejné vlastnosti jako BashCommander, a tedy se s ním přímo dala srovnávat. Samozřejmě ale je možné ho porovnat s jinými projekty, které převádějí Bash pod operační systém Windows, už proto, že právě Bash je z hlediska kódu nejobjemnější části projektu. Srovnání s běžnými souborovými manažery není příliš na místě – je jasné, že v počtu funkcí tento projekt zdaleka převyšují, integraci se textovým shellem však u nich nalézt nelze.

Pravděpodobně nejznámější projekt, který se zabývá převáděním unixových programů, tzn. i Bashe, do prostředí Windows, je Cygwin[2]. Je tedy zřejmě prvním kandidátem ke srovnávání. Na rozdíl od BashCommanderu v tomto projektu jde o zajištění všech běžných unixových API-funkcí a tím přenos jednoduše překompilovaných zdrojových souborů např. linuxových programů na operační systém od Microsoftu. Pro Cygwin tedy existuje obrovské množství unixových utilit a Bash je jen jedna z nich. V současné době (srpen 2007) jde o aktuální verzi Bashe 3.2 – je vytvořená ze stejných zdrojových kódů jako GNU Bash, takže je s ním naprosto kompatibilní. Lze pro něj použít i některý grafický terminálový program díky podpoře systémů X-Window. Bash v BashCommanderu má určitá omezení (viz sekce 2.4), nicméně na druhou stranu je lépe provázán se systémem Windows – běhové prostředí Cygwinu vytváří pro své programy speciální kořenový adresář, vlastní domovské adresáře uživatelů atp., takže nutně dochází k dělení programů na ty, které jsou v Cygwinu, a ty „venku“. Tento projekt se naopak snaží o integraci do již existujících struktur. Navíc pro něj není třeba žádné zvláštní běhové prostředí ani žádné podpůrné dynamické knihovny.

Velice podobným případem je Bash jako součást projektu MSYS pod MinGW[8]. MinGW taktéž převádí unixové (GNU) programy na Windows, MSYS je zaměřen hlavně na umožnění spuštění konfiguračních a instalačních skriptů, nikoliv na interaktivní práci uživatele, která je ale také možná. Oproti Cygwinu nebo i BashCommanderu však nelze než spokojit se se standardním konzolovým oknem, které nabízí operační systém. Podpora vlastností GNU Bashe je taktéž stoprocentní a to ve verzi 3.1 – také dochází ke kompilaci z původních zdrojových kódů, ale pro Bash je (alespoň při stažení binárních souborů přímo od zdroje) stále potřeba podpůrných knihoven a integrace do systému také vypadá podobně jako u Cygwinu.

Na internetu lze také nalézt samostatné Bashové aplikace pro Windows, které nejsou součástí větších projektů – např. Steve.org.uk GNU Bash for Windows[9], založený na zdrojových kódech starší verze Bashe 2.03. Jeho podpora je tedy také úplná, ale dnes jde o již zastaralou verzi (z roku 1999). Nepotřebuje žádné běhové

prostředí, jen jednu dynamickou knihovnu, která je s ním ale dodávaná (jde ve skutečnosti o kód od GNU kompilovaný s pomocí předchůdce Cygwinu). Tato implementace funguje, má ale problémy při spolupráci s textovými utilitami GnuWin32 (citlivost na znaky konce řádků `CR LF`) a integrace do Windows také není úplná. Lepší výsledky v tomto ohledu dosahuje win-bash[10] (také nejde o přímé přepsání, ale port existujících kódů od GNU, zde úplně bez potřeby nějakých dynamických knihoven), podpora funkcí je ale založena na velice staré verzi GNU-Bashe 1.14.2 (1994). U obou těchto projektů je ale uživatel opět nucen používat konzoli systému Windows.

Bash v BashCommanderu tedy neposkytuje tolik funkcí a úplnou kompatibilitu jako jeho kolegové z projektů Cygwin nebo MinGW (i když výhoda těch nepodporovaných vzhledem k operačnímu systému je otázkou), nepotřebuje ale žádné běhové prostředí se složitou konfigurací a mnohem více spolupracuje s Windows a jeho programy. Množiny funkcí Bashe mého projektu a popisovaných dvou samostatných implementací pro Windows jsou zřejmě neporovnatelné – v každé verzi chybí jiné a BashCommander navíc některé operace provádí jinak; přidává ale emulaci terminálu (odhlédněme od souborového manažeru) a proti prvnímu případu výrazně lepší integraci do operačního systému, vzhledem k druhému jen o něco lepší. Při přenášení Bashových skriptů je nutné provádět úpravy zřejmě ve všech případech, i když pro můj projekt jich zřejmě bude o něco víc; pro interaktivní práci je BashCommander pravděpodobně o něco pohodlnější.

# 2. Hodnocení splnění cíle práce

Moje hodnocení bude zřejmě nutně subjektivní, při pokusu o něj se ale budu snažit soustředit na fakta. Pokud jde o splnění zadání projektu, bylo ho z větší části dosaženo – výsledný program je souborový manažer, těsně provázaný s oknem shellu (ještě těsněji, než byl původní předpoklad – bez přepínání kontextu a se spouštěním příkazů klávesovými zkratkami), případně terminálový emulátor, obojí kombinované s interpretem jazyka Bash. Podpora syntaxe Bashe a unixových příkazů také odpovídá zadání, jediné, co jsem předpokládal ve větším rozsahu, jsou možnosti nastavení barev pro konzolová okna, automatické doplňování názvů souborů a obecně větší přívětivost k uživateli. Za hlavní přínos práce považuji jednak implementaci jazyka Bash pod Windows, která je s operačním systémem těsně provázaná, a jednak kombinaci souborového manažeru s textovým shellem bez nutnosti přepínat mezi dvěma aplikacemi.

Užitečnost programu by rozhodně právě snadnější operace s grafickým rozhraním o hodně zvýšily – zadání práce s něčím takovým nepočítalo, ale výrazným vylepšením by byla rozhodně spolupráce s vnitřní „schránkou“ Windows pro vkládání a kopírování textu na konzoli nebo podpora technologie drag-and-drop v případě přesouvání souborů v commanderu. BashCommander nyní spolupracuje s operačním systémem velice těsně, pokud jde o textové prostředí Bashe, v GUI však je co zlepšovat.

# 5. Závěr

Program by měl nalézt využití zejména u uživatelů, kteří ve Windows často operují se soubory pomocí dvoupanelového manažeru a zároveň používají příkazovou řádku nebo některou z již existujících implementací Bashe. Spojení commanderu a shellu je nový přístup, proto míru jeho využívání nelze ještě odhadovat. Rozšiřování schopností programu a další vylepšování jistě výrazně zlepší pohodlnost práce s programem, ale dle mého názoru je už v této verzi v praxi použitelný.

Přehled všech funkcí aplikace a jejich technický popis obsahuje přiložená dokumentace. Její text je v angličtině, neboť všechny identifikátory v kódu programu, komentáře v kódu i výpisy konzolové aplikace a popisky grafického uživatelského rozhraní jsou napsány v tomto jazyce, který je např. pro plánované zpřístupnění programu na internetu vhodnější.

# Odkazy

[1] Bash Reference Manual http://www.gnu.org/

[2] Cygwin http://www.cygwin.com

[3] KDE Konsole http://www.kde.org

[4] GnuWin32 project http://www.gnuwin32.org/

[5] Critical comments http://world.std.com/~jmhart/critcom.htm

[6] MSDN Magazine 02/04 http://msdn.microsoft.com/msdnmag/issues/04/02/CQA/

[7] Microsoft Developer Network http://msdn.microsoft.com/

[8] MinGW http://www.mingw.org/

[9] Steve.org.uk GNU Bash for Windows http://www.steve.org.uk/Software/bash/

[10] win-bash http://win-bash.sourceforge.net/

# Příloha A: dokumentace (anglicky)

## 1. About the program

### 1. Description

BashCommander is an implementation of the Bash language for Windows 2000/XP – i.e. a shell application, along with a set of common little UNIX file handling programs, such as *ls*, *rm*, *mv*, *cp*, and a GUI terminal application, capable of running several instances of the shell in tabs, in the same way as the well-known Linux terminal applications – e.g. Konsole for KDE, all combined with a two-panel file manager (the so-called „commander“). It covers most of the features of the GNU-Bash as known today, including the language constructions, variables, arrays, arithmetic and the most important internal commands, and aims to provide an efficient command-line interface for Windows. However, support of some common UNIX features, e.g. file access rights or file descriptors, is quite limited due to differences between the two operating systems. Some of the implemented commands are tweaked so that they fit the Windows environment better (that covers mostly the line feed character and backslashes in pathnames). There are also some minor differences in the language, so it's not 100% compatible with GNU-Bash, but most of the Bash scripts should work under BashCommander without the need for major changes. The file-manager is closely connected with the shell window – i.e. a change of directory in the shell window reflects in the active panel of the commander and vice versa, and all the operations of the commander just launch appropriate commands in the shell window. It's possible to simulate all the typical file-manager functions (copying and moving of files, creation of a directory) that way.

All the programming code is written in C++ using bare Win32-API. For almost all of the internal operations, BashCommander does not need to use child processes and works rather with threads. Therefore the execution speed is relatively high. The whole set of applications is available under BSD-style license, including the source code (Visual Studio 2005 may be needed to compile the sources easily).

### 2. Features

The BashCommander implementation of Bash scripting language contains almost all of the main language constructions, including simple commands, command lists, pipes, loops, such as while, until or for-loops, conditional constructs, functions, case commands, aliases and more. Variables, including one-dimensional arrays with number indices are also supported. Splitting of the commands and several types of substitutions are done in the same way as in GNU-Bash. This includes brace expansion, tilde expansion, variable (parameter)

expansions, command and arithmetic substitutions, word splitting and quote removal. Process substitution is not supported and file name expansion is little bit more limited (patterns support only `*' and `?' as wildcards). Winbash also supports arithmetic and test commands with most of the usual options. Start-up scripts are also supported, at launch Winbash reads the *.bashrc* file in the *Application data* folder.

Some of the Bash features are implemented differently, mostly due to implementation reasons or better conformity with the Windows environment, i.e. the pipes are created via background threads, so the last command in the pipe may affect the current environment (if it's an internal command), as opposed to Bash.

BashCommander also includes all of the commands found in the original Bourne shell – some of them may have limited features (i.e. the *signal* command, since the signals under Windows don't have such a great significance as in Linux). Some Bash commands, at least the most important, such *echo* or *read*, are supported.

There are a few basic file operation programs that come with the Winbash installation. It's *ls* for listing the contents of directories, *cp*, *mv* and *rm* for copying, moving and deleting files; *mkdir* and *rmdir* to operate directories and *chmod*, which is used to alter file access rights – but only the `*read-only*' Windows file attribute is changed. Most of the other programs support all the options that make sense even under Windows. Other programs, such as the UNIX text utilities, are not supplied and I recommend using their GnuWin32 implementation.

The GUI provides the execution of several Bash or commander instances in easy-to-switch tabs. It's aimed at easy keyboard control – user may customize the keyboard short-cuts for most of the operations, such as console scrolling, tab creation or switching. The GUI saves its settings and window position in a file in the *Application data* folder.

All the commander windows work in close connection with their shell window – the user may customize keyboard short-cuts that launch commands in the shell. The commands may contain values of several variables, such as the names of selected files in the active listing panel, the name of the current directory or a value that will be asked for via input box. There's no need to switch between the file listing panel and the shell window in the commander, the user types in the shell window and may control the listing at the same time.

## 3. Limitations

As mentioned earlier, this Bash language implementation does not include all the features of the GNU-Bash. Some of them have no meaning under Windows, due to different access e.g. to file rights, some may be implemented, but the complications of this process were not worth the effort (or at least for now and for me). Winbash currently does not support process substitution, several rare syntax phrases (such as `>&' for redirecting output and error stream to the same file, or function headers without `*()*') and some Bash commands. It also has not some

common Bash variables set-up (such as *BASH_VERSION*). The background job control is also not supported.

For more comfortable user experience, it would also be appropriate to add some more features, such as integration with the Windows clipboard, more customization options (colours etc.), improve the auto-completion (allow the user to switch backwards) and maybe add more mouse support (scrolling). More UNIX programs than those supplied are usually needed, but since their GNU-Win32 implementations are usually sufficient, it's not a very significant problem.

# 2. User manual

## 1. Installation, uninstallation

BashCommander is designed for Windows 2000 and newer (and has been tested on Windows 2000 and Windows XP). On older versions of Windows, such as Windows 95, 98 and ME, the application will not start up (and the behaviour of the installer is undefined).

The program is provided as an easy-to-use installation package. To install, just run the executable and follow the instructions. For some steps, the administrator rights are needed – the program adds its installation directory to Windows *PATH* environment variable, in order for the user to be able to execute the UNIX file handling routines provided comfortably. The program short-cuts in *Start menu* and *Desktop* are also created for all users. However, the program settings and start-up scripts are stored separately for each user, in their *Application data* folders (under *BashCommander* folder). The existence and access to this folder is vital for smooth application run.

The user must agree with the BSD-licence under which is this program being released, and select an installation directory. There are no limitations in this step, but I recommend leaving the default setting. Whether the *Start menu* and *Desktop* short-cuts are created is up to user's selection. However, the program always tries to change the *PATH* variable.

Uninstallation may be done via the short-cut in the *Start menu*, which is added upon installation, or through the common Windows *Add/Remove programs* interface. Just select the uninstallation and confirm its proceeding.

## 2. Using the GUI

To start the program GUI, click the icon in *Start menu* or on the *Desktop*, or execute the *bashcom.exe* program in the installation directory. When launched, an empty window with no open tabs appears. There are six icons for the most basic operations with shells. The first of them creates a new shell instance, which is run in a tab. The second one opens a new commander window (including the shell). The third tries to (force) close the currently active tab (no matter what type of tab). The next two buttons serve to switch between the shell tabs (in both directions, left and right). You may also switch the shells and commanders by

clicking on the tab names in the bottom. If a shell is closed (e.g. by typing an `*exit*' command), its tab (either shell window or file manager) disappears instantly. The last button opens the settings window (simple dialogue window involving several settings, for detailed description see section Settings and start-up scripts).

To use the shell in an open tab, just click or otherwise activate the window (tab switching, opening a new window) and simply type. The console supports most usual editing features, such as the *Home* and *End* keys, the auto-completion of file names, and command history. The default auto-completion key is *Tab*, but this may be customized in the settings window. The same is for command history keys (by default *Alt+Up* and *Alt+Down*). The command history always displays whole commands, even if they were structured or spanned across multiple lines. The history and auto-completion features are unusable when running the shell application outside of GUI.

Using the file manager window is also very simple. You may type into the console the same way as when opening a simple shell window, but at the same time it's possible to operate the files in the listing panels – you may select and de-select files using mouse or keyboard (default selection key is *Ctrl+Space*), switch between the two panels (using *Ctrl+Tab*), change the current drive (*Alt+F1*, *Alt+F2*) and toggle some action with the highlighted file in the active panel (by double-clicking or hitting *Enter*). If the file is an executable, it's run in the shell. If it's a directory, the listing view and shell's working directory change to that directory. Otherwise the commander tries to open the file in the default program. However, the behaviour of the *Enter* key changes if you have an unfinished command typed in the console. In that case, this command will be executed and the selected and highlighted files in the listings have no effect on that. At the time a command is being run in the shell window, no other operations than file selections are allowed in the list-view: hitting *Enter* or double-clicking will toggle no action.

When working with file manager window, you may also use custom keyboard short-cuts to run some previously defined commands in the shell window. These may contain some variables regarding the current state of the file manager view – e.g. the current directory, list of selected files etc., and therefore provide the functionality of the usual file manager. You may define them using a special dialogue window in settings (for details, see Settings and start-up scripts).

# 3. The Bash scripting language

The language is almost the same as GNU-Bash, so for detailed descriptions please see the Bash reference manual. The following section describes the language very briefly, but mentions the differences from the "original" version.

## 1. Commands

The whole Bash language consists of commands. The commands may be "simple", such as executing a program or a shell built-in (a small subroutine that does a simple operation), or compound, such as list of commands connected with

`&&` or `||` etc. The input is separated by the parser into commands (regarding the so-called shell meta-characters, such as `[new-line]` or `;`, and the individual commands are split into words, regarding blank characters (`[space]`, `[tab]`, `[new-line]`).

The first word of a command is the command name – it may be a name of a shell built-in, an external program or a shell function; the rest of the command are parameters. A simple command may also contain some variable assignments (preceding the command name, see Environment, variables) and redirections. Before a command is executed, variable assignations and redirections are applied and it undergoes several types of expansions (same as GNU-Bash).

A redirection means that the command input or output is being written or read from somewhere else than usually. In BashCommander, you may redirect the input or output or the error stream from/to a file (and you may append to the file), or redirect error stream to output and vice-versa. The syntax is quite straightforward – for input redirection, it's `(0)< [file]`, for output `(1|2)> [file]`, appending to a file `(1|2)>> [file]` and combination of the outputs `(1|2)>&(1|2)`. Other types of redirections (involving file descriptor numbers other than standard 0-input, 1-output, 2-error stream) are not supported. Here documents, here strings and manipulation with file descriptors are also not supported.

When a command is executed in a normal way, Bash waits for it to finish (also if an external program is executed). If you want a command to be executed on the background (then you don't have any means of controlling it from within BashCommander), divide the command from others with `` `&' ``, instead of `[new-line]` or `` `;' ``. Each command also returns its exit status – a small non-negative number. If this equals zero, that means the command ended without any errors. The return status is set as a special environment variable and may be tested for.

## 2. Plain commands

When a command is read, its first word is checked to see if it has an alias (as soon as upon parsing) – and if found, it's expanded (and may be expanded recursively). Upon execution, Bash first takes the command name and tries to find such user-defined function. If this fails, a built-in command is searched for. If there is no built-in command, Bash searches for a program of that name (and/or path).

When searching for the program, the common executable file suffixes (`` `.bat' ``, `` `.com' ``, `` `.exe' ``) are appended to its name if needed, so you may use `` `program.exe' `` as well as `` `program' `` only. If the program name contains any (forward or backward – they are always converted to backslashes upon search) slash characters, it's regarded as a pathname and only this path is tested. Otherwise BashCommander looks for the program in all the directories defined in the `PATH` environment variable – and same as Bash does not include the current directory in the search, which is what the standard Windows command-line does.

### 3. Structured commands

BashCommander also supports all the structured commands you may find in Bash – this includes the code flow control commands usually found in any programming language, as well as command lists to simplify redirections or execute something in a sub-shell. The list of the supported commands:

- **command lists** – sequences of commands, connected with `&&' or `||' , which means that a command is executed only if the commands that precede it in a command list exited normally (for `&&') or with errors (for `||'). In other cases, the given command and all its successors are not going to be executed.
- **pipelines** – sequences of commands, connected with `|'. The second command in a pipeline reads the output of the first one, and gives its output to the third and so on. The first command in the pipeline reads from the standard input, the last one writes to the standard output. As opposed to GNU-Bash, there is no child process creation for all the pipeline members (if they are not external programs). However, all the commands except the last one are executed in background (using different threads), so they cannot affect the shell environment, but the last pipeline member can.
- **until** - the syntax is: `until test; commands; do executive; commands; done`. This is almost the usual "repeat-until" loop, except that the condition is tested in the beginning. It behaves exactly the same as in Bash – runs the test commands, and if the exit status of the last one of them is non-zero, runs the executive commands and test commands again. Otherwise the looping ends.
- **while** – is similar: `while test; commands; do executive; commands; done`. As in GNU-Bash, this continues looping until the exit status of the last of the test commands is non-zero.
- **for** – a different kind of loop (and also same as that in Bash). The syntax is: `for var_name (in words); do commands; done`. This takes the variable and for each iteration, sets it a value from the list given, or from the positional arguments (see Environment, variables) if there is no list. The loop iterates as many times as the number of values given (or the number of positional arguments the shell has got).
- **arithmetical for** – different from the usual `for` loop, more similar to the one that is found in C/C++ and other programming languages. The syntax: `for(( initial_expression ; test_expression ; iterate_expression )); do commands; done`. It treats the things it finds in the brackets between the semicolons as arithmetical expressions and evaluate them in the same way. Repeats the loop as long as the test expression has a value of non-zero. For details on arithmetical expressions, see Arithmetic.
- **if** – the usual conditional construction – the conditions are also commands, that are being tested for non-zero status. The first branch for

which a non-zero status is found is executed. This behaves the same way as in Bash. The syntax is: `if condition1; commands; then executive; commands; elif condition2; commands; then executive; commands; else executive; commands; fi`. As in GNU-Bash, there may be more `elif` branches, or the `else` and/or `elif` branch may be missing.

- **case** - matches a word against sets of shell patterns and executes the commands corresponding to the first fitting one (if the word matches one of the patterns in the set). Has the same behaviour as in GNU-Bash, the syntax is following: `case word in pattern1a | pattern1b ) commands ;; pattern2 ) commands ;; esac` . The BashCommander parser, however, does not allow the lists of patterns to be enclosed in both parentheses as in GNU – there must be the right parenthesis only. For details on patterns (there are some limitations), see Patterns.
- **select** – allows for user selections, i.e. simple menus. The syntax is similar to the `for` command (including the fact that given no value list, it uses the positional arguments): select `var_name in (value1 value2 ...); do commands; done`. It also has the same behaviour as Bash – repeats asking user for input (the line number), sets the control variable accordingly and executes the commands in the loop.
- **arithmetical command** – computes a given arithmetical expression, along with any side effects it may have (variable assignations) and returns a zero (OK) exit status, if the value of the expression was non-zero. The behaviour is the same as in Bash, arithmetical expressions are described in detail in the Arithmetic section. The syntax is: `(( expression ))`.
- **test command** – tests for a given condition (see Tests) and returns a zero exit status if it was fulfilled. Also has the same behaviour as the test commands in GNU-Bash. The syntax: `[[ expression ]]`.
- **command groups –** lists of commands put together either to simplify redirection (one redirection for all the commands), or to be executed in a sub-shell and not affect the current environment (i.e. variables, current directory etc.). This also has the same behaviour as in Bash. The commands look like this: `{ commands; }` for plain command group and `( commands )` for sub-shell execution. The commands that are about to be executed in a sub-shell are written to a temporary file and a new BashCommander process is created to execute them. The temporary file is deleted afterwards. Be sure to have your Windows temporary files directory set-up and accessible.

All the types of commands may, of course, be nested, which creates the structure of the whole scripts.

## 4. Expansions

Before the execution of each (simple) command, BashCommander performs several expansions – most of them work the same as in GNU Bash. All the common expansions are performed, except for process substitution, which is not supported by BashCommander. They are done in this order:

1. **Brace expansion** – parameters within braces are expanded, i.e. `d{a,b,c}e` is expanded to `dae dbe dce`, numeric or one-character parameters separated with two dots are expanded as a whole sequence, i.e. `1{1..5}` expands to `11 12 13 14 15` . This may be turned on or off using the `set` built-in.
2. **Tilde expansion** – expands parameters beginning with unquoted tilde. All the characters up to the first unquoted slash are considered "tilde-prefix". If the tilde-prefix is just the tilde character, it's expanded to the current user home directory path (usually *c:\Documents and Settings\User* or similar), if it's `` `~+' ``, it expands to the current directory path (`PWD` environment variable), `` `~-' `` expands to `OLDPWD` environment variable. If the tilde prefix is a user name of a user of this computer (which has his/hers home directory in *Documents and Settings*), it expands to his/hers home directory. Other types of tilde expansion found in Bash (using directory stack, or in variable assignments) are not performed in BashCommander.
3. **Variable expansion** – expands the environment variable names (prefixed with `$`, may and in some cases must be enclosed in curly braces, e.g. `${var}`) to their value. The array variables with subscripts (see Environment, variables) must be enclosed in curly braces. Unset environment variables have a value of empty string. Other, more complex types of parameter(variable) expansions are also supported (same as in GNU-Bash):
   - `${var:-word}` expands to var, if var is set, to parameter otherwise
   - `${var:=word}` expands to var; if var is not set, expands to word and word is assigned to var.
   - `${var:?word}` expands to var; if var is not set, words is written to error stream and an non-interactive shell exits.
   - `${var:+word}` expands to word if var is set, to empty string otherwise.
   - `${var:offset:length}` expands to sub-string of var of given length, starting at given offset. The length setting may be omitted.
   - `${prefix*}`, `${prefix@}` expand to list of names of environment variables whose names start with prefix.
   - `${!array[*]}`, `${!array[@]}` expand to list of indices set in the array variable.
   - `${#strlen}` expands to length of the given variable value.

- `${param#pattern}`, `${param##pattern}`, `${param%pattern}`, `${param%%pattern}`, `${param/pattern/string}`, `${param//pattern/string}` may be used to strip parts of a parameter matching the pattern off the beginning or end, or replacing them with a given string. For details please see the Bash reference manual.

4. **Command substitution** – runs the contents of this string as a command and then replaces it with its output. The syntax is `` `command text` ``, or `$( command text )`. BashCommander creates a special thread to run the command and read its output. This command may access and change the BashCommander environment, no sub-shell is created. The command substitutions may be nested, but for the backward-apostrophe syntax, the nested apostrophes must be backslashed. BashCommander does not allow the `$(< file)` construct.
5. **Arithmetic expansion** – expands arithmetical expressions to their values. The expressions must be enclosed in double parentheses, such as `$(( expression ))`. The expressions may be nested. This works the same way as the Bash version, except that the behaviour upon computation error is undefined. For details on the expression syntax, see Arithmetic.
6. **Word splitting** – splits the words, regarding the characters of the `IFS` environment as blanks (i.e. word delimiters). Quoted, apostrophe-enclosed and backslashed blanks are retained. Unquoted empty words are removed. If `IFS` is unset, no splitting occurs.
7. **File name expansion** - for any word with an unquoted wildcard character (`?' or `*'), a file name expansion is performed, i.e. the word is treated as a pattern and expanded to a list of file names, that match it. If no file names match the pattern, no expansion is performed. The pattern support is quite limited in comparison with GNU-Bash. For details see Patterns.
8. **Quote removal** – as the last thing just before the execution, BashCommander removes all the "outer" quotes, apostrophes and backslashes – i.e. those that are not backslashed, or enclosed in quotes or apostrophes themselves. The quotes and backslashes that resulted from tilde expansion, variable or command substitution or file name expansion are retained (because the results of those expansions have their value backslashed) – this works the same way as Bash.

## 5. Environment, variables

Like Bash, BashCommander has its execution environment – a set of things common for all executed commands. The environment may contain (string) variables, aliases, functions (described in the next section) and one-dimensional arrays. Variables and arrays may be set as integer (with the `declare` command), but BashCommander does not treat them in any special way (like GNU-Bash), or they may be set as read-only (functions too) using the `readonly` command. The

BashCommander environment is not identical to the Windows program environment, and it's operated internally. You may, however, declare variables (and functions) as exported (via the `export` command), and then they are copied to the Windows program environment, too., and therefore they are visible in all the programs launched from within BashCommander (e.g. running scripts etc.).

Variables (and arrays) may be assigned to at the beginning of each command – the syntax is: `var_name=value`. Shell expansions (except word splitting and file name matching) are performed on the value before the assignation. If the command following the assignation is not empty, given values are valid for this command only. Otherwise they're valid for all the following commands (until any further change). Array variables may be assigned to directly – `array=([subscript]=value [subscript]=value ...)`, or through their subscripts – `array[subscript]=value` . Arrays are one-dimensional, with numerical subscripts (they are evaluated as arithmetical expressions). If you assign a non-array value to an array variable, it's assigned to the array element with zero subscript. The same happens with variable substitution. This behaves the same as in GNU-Bash.

Aliases are in fact also string variables. They may be declared using the `alias` command and are expanded upon command parsing – and may replace the command name.

There are also some variables with special meaning, that are always set – the most usual Bourne shell and some Bash special variables:

- `1`, `2`, `3`, ... etc. contains the values of the positional arguments of this shell instance (useful in scripts) – and may be altered using the `set` or `shift` commands.
- `?`, `$`, `-`, `#` contain the last command's exit status, this instance of Bash' Windows process ID (the same you find in the *Windows Task Manager*), the list of currently active settings (that may be changed via the `set` built-in), and the number of shell positional arguments, respectively.
- `PS1`, `PS2`, `PS3`, `PS4` – the values of the various prompts BashCommander may display – is exactly the same as in Bash.
- `IFS` – the separator, used to split words in the shell expansions performed.
- `PWD` – the current home directory.
- `*`, `@` - all the command arguments together. for the differences between the two and further details, see Bash reference manual.

There are more variables, such as `HOME` or `PATH`, that are always set on any correct Windows installation, because all the Windows environment variables are imported into BashCommander environment upon its launch. BashCommander therefore uses Windows `PATH` variable for programs search, and Windows home directory settings e.g. for tilde expansion.

The BashCommander environment is shared for all the commands that are being executed synchronously, including nested commands (command substitution), the last elements of pipelines and commands executed upon

request from the file manager window. The environment is always copied for all the commands that are executed in background – that means the changes made by these commands are not visible to others. All the elements of pipelines except the last one are also executed in background, so their changes to the environment are also not visible. Note that for other programs executed by BashCommander (i.e. running a script, or another executable, or executing a list of commands in a sub-shell), only the values of the exported variables (and functions) are visible. This behaves differently from GNU-Bash – the environment is much more shared in general.

## 6. Functions

Functions are a way of storing lists of commands for later (and repeated execution). Functions in BashCommander behave equally as in Bash, with some limitations. Winbash parser does not allow all the usual ways of declaring them. The declaration of function looks like this: `function fc_name(){ commands; }`. The word `` `function' `` may be omitted, but the parentheses are required, and the declaration must be a separate command.

To execute a defined function, just use its name as a command name, e.g. execute command `fc_name param1 param2 ...`. The parameters you use are available inside the function in special variables named `1`, `2`, etc. (the same as the positional arguments outside of a function). All the variables in BashCommander are global – you may not create local variables (unlike in GNU-Bash) – i.e. if you change a variable inside a function, the change persists even after the function execution finishes (unless the function is executed on background). Inside of a function, you can also see the variables declared previously outside it.

Functions may be set as exported – in that case, they are converted to their string value and set as variables in the Windows environment. All the BashCommander instances search for function code in the Windows environment and import it, if any code is found – so you may export a function and use it inside a script then. This works equally to GNU-Bash.

## 7. Arithmetic

Like Bash, BashCommander has the ability to compute simple integer arithmetical expressions – they are implemented completely and behave equally as in GNU-Bash 3.1. The arithmetical expressions are used in array subscripts, arithmetical commands and arithmetical expansions. Inside of an expression, normal variable substitution is performed, but apart from it, you may use variables directly by name (i.e. `a` instead of `$a`) even if just reading the value.

The syntax of the expressions is the same as in Bash and similar to any usual C-like programming language. You may assign variables, compute compound expressions using parentheses, and access array subscripts, too. The supported operations are:

- `var++, var--, ++var, --var` – incrementation and decrementation
- unary `+, -, !, ~` - plus, minus, logical and bitwise negation
- `var1 ** var2` – exponentiation
- binary `*, /, %, +, -, <<, >>` - multiplication, division, remainder, addition, substraction and bitwise shifts
- `<=, >=, <, >, ==, !=` - comparisons (their value is 0 (false) or 1 (true))
- `&, ^, |` - bitwise AND, XOR, OR
- `||, &&` - logical AND and OR
- `expr ? expr1 : expr2` - ternary operator – conditional expression
- `=, *=, /=, %=, +=, -=, <<=, >>=, &=, ^=, |=` - various types of assignations
- `expr1, expr2` – comma (both expressions are computed, the first value is thrown away)

Constants are also interpreted the same as in Bash – decimal, hexadecimal, octal and `[base#]number` syntax is supported. As in Bash, no overflow checking is done, just the division by zero raises an error. For operator precedence and further details see the Bash reference manual.

## 8. Tests

As in GNU-Bash, you may let the program test for some actualities, e.g. the existence of a file, or non-emptiness of a string. The tests are used in test commands and in the `` `test'/`[' `` built-in. The test commands consists of expressions – they may be enclosed in brackets, negated with `` `!' ``, or connected with `` `&&'/-a `` (for `test` built-in) and `` `||'/-o ``. The possible expressions are:

- `-a, -e, -u, -O, -G file` – true if file exists
- `-d, -g file` – true if file exists and is a directory
- `-f file` – true if file exists and is not a directory
- `-n string` – true if string is not empty
- `-o option_name` – true if the shell option with this name is enabled (details in the description of `set` built-in)
- `-p file` – true if file exists and is a named pipe
- `-r file` – true if file exists and is readable
- `-s file` – true if file exists and its size is non-zero
- `-t number` – true if given number is a file descriptor referring to a terminal (i.e. if it's 0, 1 or 2 and the given standard handle is not redirected)
- `-w file` – true if given file is writable
- `-x file` – true if given file is executable (tests it according to the file's suffix).
- `-z string` – true if the length of the given string is zero

- `-N file` – true if file exists and its last access time is lower or equal to its modification time.
- `string == pattern`, `string != pattern` – in test commands, matches a string against pattern. In `test` built-in, just tests for equality of the two strings.
- `string1 < string2`, `string1 > string2` – tests for string alphabetical order,
- `file1 -ot file2`, `file1 -nt file2`, `file1 -ef file2` – tests for age (older-than, newer than) or equality of two files (same file-nodes).
- `int1 -eq`, `-ne`, `-lt`, `-le`, `-ge`, `-gt int2` – compares two integer values

For other options found in GNU-Bash, BashCommander returns always false, for they cannot be meaningfully tested in Windows (e.g. actualities involving sockets and others). The `-u`, `-O`, `-G` and `-g` expressions have different meaning in BashCommander, since the ownership of a file cannot be tested in the UNIX way.

### 9. Patterns

Shell pattern implementation in BashCommander is limited in comparison to GNU-Bash. A pattern in BashCommander is a word that contains one or more unquoted characters `` `?' `` or `` `*' ``, known as wildcards. The quotation mark stands for exactly one arbitrary character, the asterisk may represent any number (including zero) of any characters. A word matches a pattern, if there exists any allowed replacement for all the wildcard characters that sums up to the original word.

## 4. Supported internal commands

BashCommander supports all of the Bourne shell built-in commands, but some of them may have limited options (as some of the options don't have any meaning on Windows). It also supports a set of most used Bash built-ins, sometimes with limitations, too. The list of all supported commands may be displayed with the `help` command.

### 1. Bourne shell built-ins

- `. file [arguments]` (dot) command – reads and executes commands from a file (with given arguments), in the current shell context. The file, if its name does not contain any slashes or backslashes, is searched for in all the `PATH` directories and current directory (as in GNU-Bash non-POSIX mode).
- `: [args]` (colon) – does nothing beyond argument expansion and redirections (same as in Bash).

- `[ [expression] ]` - the same as `test` command, except for the square brackets (same as in Bash).
- `break [n]` – exits a `for, while, until` or `select` loop. Exits as many nested loops as the number specifies (default is 1). (same as in Bash)
- `cd [-L|-P] dir` – changes directory. The `-L` and `-P` options are accepted, but don't change the behaviour of the command.
- `continue [n]` – continues a `while`, `for`, `until` or `select` loop in its next iteration, exits one nested loop less than the number specifies (same as in Bash).
- `eval [args]` – the arguments are read and executed as a command again, so that the expansions are performed twice (same as in Bash).
- `exec [-cl] [-a name] [command [arguments]]` – executes another process and immediately after that exits (there's no *exec()* function on Windows, so this is a simulation). The `-l` option is accepted, but has no meaning. `-c` clears the environment for the new process, and `-a` passes the given string as a new program name.
- `exit [n]` – exits the shell, returning a given exit status (default is 0) (same as in Bash).
- `export [-fn] [-p] [name[=value]] ...` - exports (or reverts the export for `-n`) given variables (or functions for `-f`). With `-p`, just lists exported variables/functions (same as in Bash).
- `getopts optstring var_name [args]` – simulates the UNIX *getopts()* function – takes subsequently the options from either positional arguments or supplied arguments and returns them in given variable (same as in Bash).
- `hash [-rl] [-p filename] [-d] [name]` – hashes a program name to a path, so there's no need to search for it later. With default settings, this is done automatically for all commands. `hash` displays the paths found for supplied command names (`-l` – in a reusable format), clears the table or just given hashes (`-r, -d`) or forces hashing a command to given file name. The `-t` option is currently not supported.
- `pwd [-L|-P]` – prints the current directory name. The options are accepted, but don't have any meaning.
- `readonly [-apf] [name] ...` - may set a variable, an array or a function as read-only. Lists the read-only variables or functions for `-p` (same as in Bash).
- `return [n]` – returns from a function or a script executed with `source` command (with given exit status) – same as in Bash (remark: currently does not return the right exit status in source scripts).
- `set [+/-aBheknfutvCx] [+/-o opt_name] [args]` - switches on or off some shell settings and/or replaces the shell arguments. The other possible settings characters are accepted, but don't do any change. Otherwise the behaviour is same as in GNU Bash. The list of possible settings:

- `-a / allexport` – exports all variables
- `-e / errexit` – exits upon ERR trap signal
- `-f / noglob` – disables file name expansions
- `-h / hashall` – hashes all paths of the commands executed (default – on)
- `-k / keyword` – the search for variable assignments in a command won't stop after finding the command name
- `-n / noexec` – just parses the commands, doesn't execute them
- `-u / unset` – triggers an error upon the usage of an undefined variable
- `-v / verbose` – echoes all the read shell input
- `-B / braceexpand` – turns on brace expansion (default – on)
- `-t / onecmd` – exits after the execution of one command
- `-C / noclobber` – file redirections are not allowed to overwrite existing files
- `-x / xtrace` – prints the commands before executing them

- `shift [n]` – shifts shell the positional arguments (variables `1, 2 ...` ) by given offset to the left. The value of the `` `#' `` variable is adjusted accordingly (same as in Bash).
- `test [expression]` – tests for a given expression, similarly as in test commands. For details on the expression syntax see Tests (same as in Bash).
- `times` – prints the CPU times under user and kernel mode for this shell instance and its child processes (which is not guaranteed by Windows, just an estimation) (same as in Bash).
- `trap [-lp] [arg] [sigspec]` – sets a trap command (`arg`) to handle a signal – it' s possible to set a trap for all the UNIX signals, but most of them will never be hit since Windows does not support them. Meaningful values of `sigspec` are `SIGINT`, `ERR` or `DEBUG` – executed upon Ctrl-C, command with non-zero exit status or the execution of any command. For `-l`, prints a list of available signal specifications, for `-p`, prints the trap for given signals (same as in Bash).
- `umask [-p] [-S] [mode]` – sets the internal umask (file access rights creation mask) – under Windows, it has no effect, but pretends to be behaving equally to `umask` in Bash.
- `unset [-fv] [name]` – unsets a function or a variable in the environment (for `-v` the name refers to a variable, for `-f` to a function) (same as in Bash).

### 2. Bash built-ins

The `bind, enable, local, logout, printf, shopt, type, typeset` and `ulimit` built-ins are not supported.

- `alias [-p] [name[=value]]` – sets the aliases or displays current settings (with `-p`) (same as in Bash).
- `builtin [name [args]]` – runs a shell built-in, used to override functions with same name (same as in Bash).
- `command [-pVv] [name [args]]` – either runs a shell built-in or an external command (and overrides functions), or prints a description of the given command (for `-v`/`-V` which vary in the amount of details). The `-p` option is ignored, otherwise behaves equally to Bash.
- `declare [-afFirtx] [-p] [name[=value]]` – declares new environment variables, arrays (`-a`), functions (`-f`/`-F`) or lists their values (`-p`), or sets/finds out the integer (`-i`), read-only (`-r`) and export (`-x`) status. The `-t` attribute is ignored, (and integer attribute may be set, but has no special meaning) otherwise behaves same as Bash.
- `echo [-neE] [arg ...]` - prints its arguments to the standard output. If `-n` is set, the ending new-line character is suppressed. The other options are ignored (escape-characters are not interpreted).
- `help [-s] [command_pattern]` – displays a usage message for commands that match given patterns. The detailed description is not supplied, so the `-s` option is assumed always turned on. If no parameters are given, `help` displays the list of all commands supported by BashCommander.
- `let expression [expression...]` - each expression (one argument) is evaluated the same way as in an arithmetical command (same as in Bash).
- `read [-rs] [-t timeout] [-p prompt] [-a array] [-n nchars] [-d delim] [name ...]` - reads the values of given variables from standard input. May read elements of an array (`-a`), end when given number of characters is read (`-n`) or a timeout (seconds) elapses (`-t`), divide the values with a custom delimiter (`-d`) and display a prompt (`-p`). For `-r`, reads raw input, `-s` suppresses the echoing of characters read. Splits the characters using `IFS` and sets the resulting fields in the variables supplied, or in `REPLY` variable (the supported options work in the same way as in Bash).
- `source filename` – see the `. (dot)` command (same as in Bash).
- `unalias [-a] [name ...]` – removes the aliases for given names, or all aliases (for `-a`) (same as in Bash).

## 5. Supplemented UNIX file operating programs

BashCommander installation contains a small set of most-used UNIX file operating programs. Their executables are copied to the BashCommander installation directory which is added into Windows `PATH` environment variable, and so they may be executed comfortably from within BashCommander (and also *Windows Command Line*, for example) by just typing their name. Usually they

work the same as their GNU versions, except that they lack long option names and some options may also be unsupported. They are:

- `cat [-AbeEnstTv] [file ...]` - echoes the contents of files (or standard input, if none are given) on the standard output. It's able to number non-blank lines (`-b`), print `` `$' `` at the end of the lines (`-E`), number all the lines (`-n`), suppress blanks longer than one line (`-s`), show `[tab]` characters as `` `^I' `` (`-T`) and show non-printing characters except `[new-line]` and `[tab]`. `-A` is the equivalent of `-vET`, `-e` is `-vE`, `-t` is `-vT`. The supported options work equally to GNU `cat`.
- `chmod [-Rcvf] mode file [file ...]` - in UNIX, this changes the file access rights, this implementation supports only the changes in the *read-only* attribute of files (this means that if you take away the right to write to a file for the owner, the file is marked as read-only). Otherwise the supported options work the same as in GNU `chmod` (see chmod reference for details).
- `cp [-abfipRTuvx] [-t target_dir] [-S backup_suffix] file1 file2 [file ...]` - copies source files to given destination. It can create backups of existing files (`-b`) with default or pre-set suffix (`-S`), force overwrite of existing read-only files (`-f`), prompt before overwrite of all (`-i`), preserve file attributes (`-p`), work recursively (`-R/-r`), copy all files into target directory (`-t`) or not consider any target directory (`-T`), omit newer files from rewriting - "update" (`-u`), display the names of files being copied (`-v`), or never copy across physical drives (`-x`). The `-a` option is equivalent to `-pR`. Other options of GNU `cp` are ignored.
- `ls [-aAbBcCdfFgGhklmnopqQrRsStUxX1] [-I ignore_pattern] [-w width] [file ...]` - lists the contents of a directory. The supported options are (other options are ignored):
  - `-a/-A` – display also files whose names begin with `` `.' `` that are hidden by default (`-A` does not display `` `.' `` and `` `..' ``)
  - `-b/-q` – print non-graphic characters as their octal escapes/as `` `?' ``
  - `-B` – ignore files whose names end with `` `~' ``
  - `-c` – if used with `lt` or none of these, sort the files by their access time, for `l` only display access time (instead of modification time)
  - `-C` – list entries by columns (default)
  - `-d` – treat directories same as files (do not list their contents)
  - `-F/-p/-Q` – append file type indicator, append `` `/' `` to directory names, append double quotes
  - `-g/-G,-o/-n` – in long listing (`-l`), display no owner/group specification/with their numerical values (all of which in fact under Windows have no real meaning)
  - `-h` – display human readable size (in kilo/mega/gigabytes instead of bytes)
  - `-I pattern` – ignore files matching the given pattern

- `-k` – size of block should be one kilobyte (instead of 512 bytes)
- `-l` – long listing (with modification time, size, owner, group and access rights)
- `-m` – comma separated listing
- `-r` – reverse sort order
- `-R` – recursive
- `-s` – prefix the output with the total size of listed files (in blocks)
- `-S/-t/-X/-U` – sort by size/modification time/extension/don't sort (default – sort by name)
- `-w` – force given screen width
- `-X` – list entries by lines
- `-1` – one-per-line output

- `mkdir [-pv] dir [dir ...]` - creates a directory. If `-p` is set, creates all the parent directories if they don't exist. For `-v`, shows output for current actions.
- `mv [-bfiRTuvx] [-t target_dir] [-S backup_suffix] file1 file2 [file ...]` - moves files (i.e. gives them a new pathname). Its options are equivalent to `cp`, except that it lacks `-a` and `-p` (file attributes are always preserved, as long as the file is moved, not copied). `cp` and `mv` executables are in fact the same and the action is decided by the name they were called.
- `rm [-frRiIv] file [file ...]` - removes a file from the disk. It's possible to force removal of read-only files (-f), recursively remove directories, too (-r/-R), prompt before each removal (-i) or only before removal of more than 3 files or recursive removal of a directory (-I) and display a message for all the actions performed (-v).
- `rmdir [-pv] dir [dir ...]` - removes a directory (directories) – they must be empty, the options are the same as for `mkdir` (the `mkdir` and `rmdir` executables are exactly same).

## 6. Settings and start-up scripts

In the settings window, the user may alter several program options. It's possible to turn off the toolbar and and use all the space for shell windows. In that case, you may only turn the toolbar back on using a keyboard short-cut (*Ctrl-Shift-P* by default). You may also alter the font setting – only mono-spaced fonts are suitable. The font is changed in a standard Windows font dialogue window. A background colour for the shell windows may be set-up, too. The most important setting is the keyboard short-cuts. It's possible to customize the short-cuts for most of the operations. The set-up is very straightforward – just select the `*Settings*' button near to *Keyboard short-cuts* caption in the dialogue window, then select the desired action from the list, double-click its name and press the desired short-cut. Only the short-cuts involving some valid key are accepted (valid keys are character keys + arrows, *PgUp*, *PgDn* etc. and *F1-F12* keys). Be sure not

to use just plain characters (e.g. `*D*' as a short-cut), since you will need to type them in the console window, which this would prevent. You may also reset the keyboard settings to default.

To define the file manager commands short-cuts, click the custom commands *Settings* button in the settings dialogue window. You may then add a new command or delete an existing one by clicking the corresponding button, or edit a previously defined command by double-clicking it. A custom command definition window appears, where you need to set-up the keyboard shortcut (by clicking the appropriate button) and fill in the command syntax. You may also want to add titles to any input boxes that may appear when executing the command (which will be explained in a moment). The command syntax may contain anything a Bash command may contain, as well as some special variable values. These variables are connected to the running file manager instance; the Bash commentary mark `#' is used as their prefix. They may be the following:

- `#1`,`#2` ... `#9` – these expand to the first, second etc. file that is selected in the active file listing panel.
- `#*`, `#@` – expand to the list of all files selected in the active/inactive panel.
- `#D`, `#O` – expand to the name of the current/opposite panel directory.
- `#F` – expands to the name of the currently highlighted file
- `#A`, `#B`, `#C` – when these are used, an input box with given question (that is to be filled in the mentioned edit fields) appears and the variable is expanded to the input the user types in.
- `##` - is the same as `#' alone, so that you may even supply the commands with a commentary.

The settings are saved immediately after clicking the *OK* button of the main settings dialogue windo, which also closes it. The application also remembers its window size and position, and starts always in the same position it was previously closed in. All the GUI settings are stored in the file *settings.conf* in the *Application data\BashCommander* directory. If you delete the file, you revert all the settings back to default.

## 7. GUI short-cuts table

This table shows the actions, along with the default short-cuts that are pre-set upon program installation (and the user may revert back to them in the short-cut settings window by clicking the *Reset defaults* button). The first set of short-cuts may be used anywhere in the program:

| Action | Short-cut |
|---|---|
| Close the active tab | Ctrl-W |
| Command history – go back one command | Alt-Up |
| Command history – go one command forward | Alt-Down |

| Action | Short-cut |
|---|---|
| Move one tab to the left | Ctrl-PageUp |
| Move one tab to the right | Ctrl-PageDown |
| Open a new Winbash instance (tab) | Ctrl-N |
| Open the settings dialogue window | Ctrl-Shift-P |
| Scroll the console buffer – one line up | Shift-Up |
| Scroll the console buffer – one line down | Shift-Down |
| Scroll the console buffer – one page up | Shift-PageUp |
| Scroll the console buffer – one page down | Shift-PageDown |
| Shell auto-complete function | Tab |

The second one is for use within file manager windows only (however, the two set must not overlap – i.e. if you set the same key combination for second action in any of the sets, the first action will not have any short-cut assigned.

| Action | Short-cut |
|---|---|
| Move the highlight one line up | Up |
| Move the highlight down | Down |
| Move one page up | PageUp |
| Move one page down | PageDown |
| Switch between panels | Ctrl-Tab |
| Select/deselect highlighted file | Ctrl-Space |
| Toggle file selection and move up | Ctrl-Up |
| Toggle file selection and move down | Ctrl-Down |
| Select all files | Ctrl-A |

# 3.Programmer's manual

## 1. Used language

The whole application code is written in C++ language, with heavy use of the STL libraries (mostly `<string>`, `<sstream>`, `<vector>` etc.). Many of the internal supportive structures are derived classes of some STL containers or have STL container features. Use of parameter pointers is preferred over the use of plain parameters to improve speed at most places – this means that most of the structures have internal pointers and need to have explicit destructor functions. The return values of supportive functions may be either `const` pointers, and then they should not be deleted, or a normal pointers and then it's up to the calling

function what to do with the pointer. In most cases it's written in the source code at the function definition what the function returns (a direct pointer or a copy), and also what it does with its parameters – some functions take their parameters and save them in some internal structure, after which the parameter pointer must not be used in calling function.

For all the input and output operations, window creations etc., bare Windows-API functions are used. There are some functions used that are not available in Windows NT4/95/98/ME or older, so the application won't run on older versions of Windows. Some *common controls* library functions are used, too, so *comctl32.lib* is needed for compilation. The definition of the Windows version is in the header file *common.h*, which is included in most other files. For the code editing and compilation of the binaries, Microsoft Visual Studio .NET 2005 has been used – the Visual Studio project and solution files are supplied with the sources.

All the string operations are done with wide-strings (Unicode) – i.e. the `std::wstring` and `std::wstringstream` classes are used instead of `std::string` and so on. Even all the hard-coded constants in the application are wide-strings. All the input that is read from files and pipes is automatically transferred to wide strings, using Windows-API functions.

## 2. Source code files structure

The BashCommander sources are divided into four directories – *BASH*, that contains all the sources exclusive to the shell application itself, *COMMAND*, containing the source files of all the implemented UNIX file operation commands, *COMMON*, in which there are files needed for more than one part – i.e. all the support functions for both Bash and UNIX utilities, or a header file for memory mapping definition, shared by Bash and the GUI. The last directory – *GUI* – contains the files needed exclusively by the GUI. All the C++ code is divided into object declarations (in *.h* files) and definitions (in *.cpp* files) – one header (declarations) file may contain declarations of more than one C++ class, and the definitions may be contained in more code files.

### 1. Common supportive libraries

There are some objects that are shared by Bash and the UNIX file utilities – mostly related to reading standard input and writing output, and to matching the shell patterns. Some definitions of commonly-used exceptions are also shared. All the files of this type are located in the *COMMON* folder.

The most significant of them are probably the definitions of input and output objects, in the files *streams.h* and *streams.cpp*. The exception definitions are in *except_common.h*, *except_common.cpp* and *outofmem.h, outofmem.cpp* – all the exception classes for both Bash and UNIX file operations are derived from the class *AbstractException* which is to be found in the first pair of files. The other one contains special exceptions used to handle out-of memory errors (`new` failures). All the pattern matching is done using the *Matcher* class, from the files

*matcher.h* and *matcher.cpp*. The last thing shared by Bash and the UNIX utilities is *ArgVector* class – simple class, derived from a vector of wide-string pointers – this is used almost everywhere for command arguments and similar objects.

## 2. The simple file operating programs

All the sources of the UNIX file operation utilities reside in the *COMMAND* folder. It's just the commands' main files (with same names as the commands) itself, and four support libraries. First of them is *cmdexcept.h* & *cmdexcept.cpp*, containing a definition of the commands' standard exception. The other is *cmdenv.h* & *cmdenv.cpp* – definition of the *CmdEnv* class that stores all the settings for the command (i.e. what options were turned on). The next pair of files contains the *Question* class – it's used to ask the user questions and return yes/no/always/never as an answer. The last one is *fileop.h* and *fileop.cpp* with the *File* class – an abstraction of a Windows file, containing functions to test its attributes, size etc.

All the utilities also use the input and output objects from the *COMMON* folder and exceptions, too. *Matcher* class is required by `ls` only (to test the ignore patterns). The `rmdir` and `mv` utilities don't have sources of their own, because their features are included in `mkdir` and `cp` commands – their executables are created simply by copying (which is set-up in the *Visual Studio* project).

## 3. The Bash application

The Bash application has its sources located in the *BASH* folder – the *wmain* function is in *main.cpp*, the rest are definitions of the main classes and supportive classes. All of the source files for Bash implementation include the *winsh.h*, which contains all the standard-defined constants. Some more constants, closely connected with the Bash language (i.e. meta-characters, blank charcters etc.) are in *lang_const.h*. The main program object is located in *winshell.h* & *winshell.cpp*. The environment, which is shared by most of the objects, is in *env.h* & *env.cpp*, in a class named *Env*. Definitions of various kinds of variables (arrays, functions, hashes, aliases) are in *var.h* and *var.cpp*.

All the classes that ensure the parameter expansions etc. are declared in *expandor.h*, and defined in several files – *argexpandor.cpp, expansionbase.cpp* and *expandor.cpp*. Other classes serve as support for the expansions, such as *tester.h* and *tester.cpp*, or *arithm.h* with *arithm.cpp*. All the expansion classes along with the *Executor* class (*executor.h*, *executor.cpp*) serve the various command classes (for them to be able to execute themselves) – these are located in the *command.h*, *command.cpp* pair of files.

The parser, responsible for splitting the input into commands, uses the files *parser.h* and *parser.cpp*, as well as some supportive objects from *control.h* and *control.cpp*. All the classes that throw any exceptions use *execept.h* and *except.cpp*, that contains definitions of most of the exceptions. Special exceptions, used for control-flow changes (break, continue etc.) are written in *flowctrlexcept.h* and *flowctrlexcept.cpp*. The files *intcmds.h, intcmds.cpp* and

*intcmds-support.cpp* contain the implementation of all the supported shell internal commands.

The rest of the files are mostly some supportive structures, such as signal handler (*sighandler.h, sighandler.cpp*), shell settings (*settings.h, settings.cpp*) or definitions of various containers (e.g. the command container, or input and output handles container), residing in *supstruct.h* and *supstruct.cpp*. The routines needed for communication with the GUI (if run under GUI) are located in *msgthread.h* and *msgthread.cpp*.

### 4. The GUI

The GUI code consists of relatively small number of files, in the *GUI* folder. The *wmain* function is located in the *main.cpp* file. The most important GUI object, enclosing the main window, is in the *mainwin.cpp* file. Its declaration, along with the some supportive declarations, is to be found in the *mainwin.h* file. The object responsible for displaying one shell tab is in *shellwindow.h* and *shellwindow.cpp*, the file manager window is represented by *commander.h* and *commander.cpp*, with a support class in *inputbox.h* and *inputbox.cpp*. The abstract base classes for these windows are located in *controller.h* and *shell.h*.

The rest of the files involve mostly the settings of the whole GUI and keyboard shortcuts – *keycode.h, guisettings.h*, *guisettings.cpp*, *shortcuts.h*, *shortcutdata.cpp* and *shortcutdialog.cpp*; and custom commands settings in *customcmds.h* and *customcmds.cpp*. In the other files, the *GUIException* class is defined (which is thrown for GUI errors and should not occur in the ideal case) (*guiexcept.h, guiexcept.cpp*) and some constants (*gui_const.h*).

The source code also contains some resource files needed by the Windows applications – dialogue windows and icons for the buttons and the application itself. These involve all the **.ico* files, as well as common *resource.h* and *winbash_gui.rc* files.

## 3. Supportive libraries – objects

The supportive libraries, shared by Bash and UNIX file utilities contain a few small, but very important classes that are used all across the project. All of them are enclosed in the namespace *cmdcommon*. The most important of them are the input and output handles, which are going to be discussed in the subsection.

Then there are a few exception classes – it's *AbstractException* - the base class for all the exceptions (outside of GUI), which defines a simple interface with the *what()* function (as seen in `std::exception`, that, alas, does not support wide strings), the *WException* which is the base class for command/file exceptions and contains an error message, and the special *OutOfMemoryException*, set in all the non-GUI programs as the handling routine in case `new` fails due to memory shortage (this is done via the Win32-API function `_set_new_handler`).

The perhaps most used class of all is *ArgVector* – in Winbash, its more sophisticated derivee is used. It's itself derived from the `std::vector< std::wstring * >` class, but has more features – it's possible to decide

whether you want a member of it being just erased from vector, or erased and freed in the memory – most of them involve freeing or not freeing memory, this provides an easy way to insert and delete string pointers without the need of copying the actual strings. It also has the ability to convert itself to a string (by concatenating all its elements).

The *Matcher* class is responsible for all the pattern matching operations in the whole project (except for file name matching, which is currently done via the Win32-API FindFirstFile function). It contains static public functions *match* and *replaceLongestMatch*, that provide the all the interface needed – it has not got any constructors, so all the usage of this class is just through its static functions.

### 1. The input and output handles

The input and output handle are used in all the console programs from the project. They, in fact, beyond other things, recreate the insufficient Win32-API high-level console functions, using the low-level ones. All the objects contain a Win32-API `HANDLE` value, and may be converted to and from it. The base class for handles is *Handle*, its derivees are *InputHandle* and *OutputHandle* (the names are self-explanatory).

The *OutputHandle* enables the Windows file or console output or pipe handles to be used in a same way as C++ streams – it has a heavily overloaded `operator <<`, able to output wide strings, numbers and other things. The operator is also `NULL`-proof. Otherwise, the *OutputHandle* is the simpler class – it does not need to alter any Win32-API console functions.

The *InputHandle*, on the other hand, contains many internal variables and functions to be able to recreate the feel of console input, and at the same time support such things as auto-completion (resetting the buffer) or time-outs for input. The main functions are the two overloads of *readLine*, one of which has more features (such as the above-mentioned time-out) and uses the other. Both the functions use *readFile* and for console handles also *processConsoleInput* and others.

All the internal variables (buffers, flags etc.) are created as `mutable`, because `const` handles are used in many places of the code. And logically, the truly one needed thing that must not be changed when reading input from the handle is the handle value itself. All the functions that change the buffer (*buf* member) are thread-safe – i.e. they need to initialize and enter the critical section that is connected with this handle (the *critSect* member). Because for file and pipe input, it is never known which character is going to be the delimiter and calling the read operation for one char only in a loop would be a waste of system resources, the handle also contains a buffer of pre-loaded input (*preloaded* member). Therefore one must be very careful when copying the handle – the buffer is not copied completely in the copy constructor, only the pointer is copied and a special parameter – instance count is incremented. So the copying from *InputHandle* to another *InputHandle* is correct, but a copy through standard Win32-API `HANDLE` spoils the pre-load buffer.

The *InputHandle* class also contains various support functions – i.e. to return the last error that occurred when reading from the file, or check whether the buffer contains requested amount of characters or the delimiter. To enable console scrolling (any redrawing of the buffer spoils it), for console input handles there is a flag that turns the redraw off, until a first valid character is read (i.e. a key is pressed).

To support receiving of commands from the file manager, *InputHandle* has another feature – it's possible to set-up another handle to be waiting for (i.e. an event handle) other than the internal one and if this handle is OK with the wait function (i.e. the event occurs), an exception (of type *InterruptedException*) is raised. This is used in the Bash application itself only.

## 4. GUI – code schema

The GUI code is quite simple – the main object is (practically) the main window – *MainWin* class. It controls the individual shells (each of which has its own thread) and commanders run in tabs (classes *ShellWindow* and *Commander*) and may open the settings dialogue, represented by the *SettingsDialog* class. The *MainWin, ShellWindow* and *Commander* classes and their C++ inheritance connections are described in detail in the subsections. There are several settings dialogue windows in the GUI, and corresponding to them there are setting dialogue classes: *SettingsDialog*, *ShortcutDialog* and *CustomCommandDialog*.

Generally, if a class is bound with a window, its instance is the window itself, i.e. upon window creation, the pointer to the object the window belongs to is set-up at the standard Win32-API `GWLP_USERDATA` place holder, and in the window procedure, all the messages are redirected to a similar procedure invoked on this object. So the window is totally bound with its instance of a class and all the actions are processed through this instance. The class itself has the window Win32-API `HWND` value as one of its members, so the connection is mutual.

The GUI settings are stored and loaded from a file via the *Settings* class. There's a special `struct` to represent the settings in the memory – *SettingsData*. The keyboard short-cuts have their settings in special classes, which are enclosed in the global settings. The most important is *ShortcutData* – a container capable of testing for the presence of a given shortcut. For file manager custom command short-cuts, there's the *CustomCommandData* class that controls and stores all the settings.

### 1. The main window

The application main window is bound with and created in the class *MainWin* – this class is a singleton. It contains all the data for program settings, keyboard shortcuts and also windows settings (which mostly overlap). This class also stores a list of pointers to *Shell* classes – currently open tabs (there's no difference as to whether it is a commander or shell window – both of them inherit all the members of *Shell* class). It is responsible for creating new and closing the running

(both types of) shells. To communicate with the shells, it uses windows messages (each of them has its own message processing procedure).

It also does some services for the individual shells/commanders – i.e. it is responsible for changing the caption of a tab when a shell asks for it. Because the shells need to access the application global settings (as read-only), it provides a method of receiving a pointer to them (*getWinInfo*). It also provides a way to access the keyboard short-cuts settings, or search within them (there are two functions, one for file manager windows and one for plain shells), the actual work is, however, done within the *settings* member (which is of *SettingsDialog* type).

## 2. The shell tabs

One *ShellWindow* instance is responsible for all the communication with its Bash instance, for reposting any key presses and redrawing the output of its screen buffer. It also receives the messages from the main window (i.e. to close the shell). The object itself has a private constructor and must be created via the *Create* function, which runs a special thread for its instance, too. The thread function is *shThreadFunc*. The shell graphical window may be created and controlled either by the main window directly or by a commander – it holds a pointer to a general *Controller* object, which is a common base class of both of them.

Apart from the reposting of input and receiving messages from the Bash instance and main (or commander) window, which is handled directly in the *processMessage* method (sometimes through *postShellKey* and other supportive methods), it needs to be able to redraw the output of the Bash console (*updateScreen*) and also resize the console, if the user resizes the main window (*resizeConsole*). If the user switches between the Shells, they are notified by the main window that they are, or are not, going to be the topmost ones from that time on (*setTopmost*). If a Shell is not set as topmost, it does not redraw the console output (because it's not visible anyway).

## 3. The file manager windows

The file manager windows needed to be put somewhere in between the main window and shell windows, because they are created as tabs under the main window, and themselves create the shell windows. The C++ multiple inheritance has been used to achieve this effect. There are two abstract classes – *Controller* and *Shell*, that serve as common base classes for *Commander* and *MainWin* or *Commander* and *ShellWindow*. Both of them have no data members and no function code defined, so there are no problems with the fact that *Commander* class is derived from both (it in fact needs to have all the functions – some of them are invoked by the main window, some of them by the shell).

Therefore, it serves mostly as an intermediate stage between the shell and main window. Many of the functions are just passed on in the right direction, but some values are altered by the commander – e.g. the height of the console window for shells (that is lower within file manager, there needs to be place to put the file listings to). It also ensures the redrawing and selecting of files in the listing panels

is done and requests to run custom commands are passed to the shell (*executeUnderCursor*, *postShellCommand*, *postShellCd*).

# 5. Bash application code

The application code, except for the above mentioned support libraries that are shared with the file utilities, is stored in the *BASH* folder. The most important object are the *WinShell* and *Env*, which represent the shell instance itself (and not only one in the whole program) and the execution environment. They then use and/or contain the other objects as their members. The most important of them are described in the following sections.

## 1. The main object

The main object is *WinShell* – it represents the whole program. Upon start-up, one instance of *WinShell* is created by the *wmain* function. Its main method is *work* – this tries to read and subsequently execute commands, until the end of input or an ending command is found. It also catches the various exceptions the program may raise, i.e. syntax errors or program execution errors, as well as control flow exceptions. The normal way of handling an exception is to display its message and continue.

*WinShell* has its own *Parser* object that receives the input and returns it split into command objects. It also has its own *Env* object, that means the environment where all the variables and other settings are stored, e.g. the standard handles, shell settings and signal handles. However, not every *WinShell* instance has an *Env* of its own – it is to say that in several situations, such as when running a command substitution, or for sourced scripts, other *WinShell* objects may be created. Those shell objects share the *Env* with the main and basic one (which is by that time waiting until their execution ends).

This objects also retains its command cache, which is used for storing commands before their execution as well as for command history. Some methods involve reading the commands (from the standard input handles that are stored in *Env*!) or displaying the prompt. This object is also responsible for processing the command arguments (i.e. running the right script file) or the start-up scripts.

A lot of work for *WinShell* is done in cooperation with its member – the message thread (*MsgThread*). This means mainly sending keys to its own input, displaying the commands in history or the auto-completion feature. The shell object also stores the current script file name and line number which is needed by most of the commands for displaying error messages.

## 2. Environment

The *Env* class represents the shell environment. A pointer to the (usually only one, except for background threads) instance is spread across the whole memory – each command has a pointer to it, the parser and *WinShell* class, too, it's needed for all the expansions that are performed – it's a service class almost for every other "executive" class in the application.

This class stores all the user variables, arrays, command hashes, functions and aliases in its *globalEnv* member (it is a map, name-to-value) – these are all derived classes of the *Var* class. It's capable of setting, unsetting, swapping values, listing all values etc. – all the functions directly correspond to the needs of the internal commands and expansions. It also handles export and read-only statuses of variables. Some special variables have also special means of setting them, i.e. the command exit status and the shell arguments (*setLastExitStatus*, *setShellArgs*). All of the setting functions take directly the pointers supplied to them and store them in the environment – no copying is done, but the pointers must not be used after the call to a setting function. The getter functions return a direct constant pointer to the environment (unless specified otherwise in the code), so the value of e.g. a string may be used directly, but for write operations, a copy is required. Some of the getter functions just return the value e.g. of an array element, some return directly the object that stores it.

Besides that, *Env* is used to store all the input and output handles. The *getStdHandles* method returns a `const` pointer to the handles that are currently valid for the application – all commands, before their execution, push their input/output handles (containing any redirected handles, or `NULL`'s) onto *Env*'s so-called pipe stack (*pipeStackPush*). After their execution, they are no longer valid for the environment and may be pushed off (*pipeStackPop*). The *getStdHandles* function browses the stack from the upper side until it finds valid (non-`NULL`) handles for all the standard streams and returns the result in a copy. If some of the handles are still `NULL` all the way through the stack, it returns the standard handles for the shell process. This computation is moreover buffered and performed only when the handles change. *Env* also stores a copy of the process input handle (needed because of the buffering, see Input and output handles) and a copy of the input handle from the time the shell was launched (needed if the `read` function is called from within a script, that has its main input handle redirected (to read the commands)).

The *Env* class also stores some other settings, e.g. whether a nested thread is currently run, whether the Ctrl-C has been sent, whether a command list is being run, `umask` setting and a few more. It has the signal handler as its member object (and that is shared by all the instances of *Env*, as only the first is created via the plain constructor, the others must be *clone*-d and take only the pointer). The current loop depth (for `break` and `continue` commands) is stored there (*loopDepthGet* etc.) and a directory stack is implemented, but currently remains unused.

Last but not least, the *Env* objects store the shell call stack for functions – when a function is launched, it's push on the call stack (*callStackPush*) and for subsequent requests for positional arguments or `return` command, the right information is supplied. After the execution of the function call command, it's pushed off again (*callStackPop*).

## 3. The parser

The parser splits the input into the commands as the lines are read. Its main function is *parse*, which is given a line of input and always returns all the complete commands that were found – if a line is read, and contains an incomplete command, it is temporarily saved in the parser's internal structures, until its end is read. Therefore, the function may return an empty vector as well as several commands – if they are on the same line of input. With "commands", even structured commands, including loops and nested loops are meant here. Other functions return a status whether it contains an incomplete command in its member structures, and allow for complete parser *reset* (e.g. upon Ctrl-C hit).

Internally, the parser stores a control stack (*ctrlStack* - consists of *Control* objects) for keeping the various incomplete structured commands, such as the parameters of a `while` or `for` loops. The beginnings of compound commands are pushed on the stack, and if and end, such as `` `done' `` or `` `fi' `` is found, the whole command is completed (created as an object) and stored in the buffer to return it. For reading simple commands (even as parts of compound commands), the parser has a *tokenStorage* vector, containing all the read words (split by blanks). It also stores some flags (*nestStack*) in a stack, to search for the matching quotes, or the end of arithmetical expression or command substitution and so on. The behaviour of the parser is influenced by the contents of the *ctrlStack*, as well as *cmdExeFlags* – a set of flags useful for pipelines, command list, function definitions and sub-shell execution. The main parsing function is *splitWords*, with the help of *checkMetaCharacters*.

The parser also expands the aliases (and may do that recursively), and parses all the redirection information – and cuts it out of the command to be handled separately. All the functions may throw exceptions, which are caught outside, in *WinShell*. The parser is reset by *WinShell* after that.

## 4. Command objects

All the shell commands, except for command lists (`&&` and `||` - connected commands) and pipelines, are objects of their own. All the types of commands are derived from the *Command* class, which is abstract and contains the member functions and variables common for all kinds of commands – that means pointer to *Env*, redirections, standard handles and execution flags. The command object has also virtual methods to *exec* itself, or to convert itself to string (*getAsString*) predefined. All the commands also are able to create redirection files and open them (and clean that up afterwards) and print debug trace. All the commands may be also executed on background – the *backgroundThread* serves for this.

The simple commands are represented by *PlainCommand*, which is able only to execute the one function call, internal built-in or external program. Other kinds of commands usually contain vectors of commands (or even vectors of vectors of commands, e.g. for `elif` branches, as well as vectors of commands bound to a pattern, for `case` branches) and execute them one-by-one or selectively (see e.g. *CmdVect* and *execAll* method). There's *IfCommand* for all types of `if-then-`

`else` tests, *TestLoopCommand* for `while` and `until` loops, *ForCommand* for `for` and `select` commands, *ArithmCommand* for arithmetical commands and *TestCommand* for test commands, special *ArithmForCommand* class for representing arithmetical for-loops, *CaseCommand* for the case branches, *FunctionCommand*, representing the whole code of a function, and finally *ListCommand* that serves for sub-shell execution and plain command lists. All the commands have their *execSync* method overridden the right way for their execution.

All the commands must have all its arguments (and sub-commands) supplied upon creation, which happens in parser – the subcommands of the incomplete commands are stored in *ctrlStack* and then put together into the constructor for the specific command. The commands may be also copied – using the *clone* method, which is also virtual and always creates the right type for the given command.

## 5. Expansions and execution

For all the shell expansions, the command objects use the *Expandor* class. That is derived from *ArgExpandor* class, which is able to perform the expansions on everything except variable assignations and file redirections. The *ArgExpandor* class is used in some cases where those are not needed. There are also several supportive expansion classes, i.e. for to split test expressions or to create variables from given name and value strings, or used in the auto-completion feature (*AutoCompletor*). All the expansion classes are derived from the *ExpansionBasicFunctions* class that provides several vital functions for any expansion – the most important is *findNestedEnd*, that, given a string and a position, searches for the end of the nesting (e.g. parameter expansion, nested command) using a nest stack. There are also several support functions for the other classes.

The *ArgExpandor* class' main function is *expand*, which uses several other functions, that correspond to all the kinds of expansions shell is supposed to perform, i.e. *expandBraces*, *expandTilde*, *expandVars*, *expandNestedCommands* (which, moreover, use e.g. *performNestedCommand* or *performVarSubstitution* to do the executive work). Because the quotes and backslashes that come as a result of an expansion must be retained, these methods must have an indication of whether they are currently inside of quotes or not (the *inQuotes* parameter). If not, the quotes, apostrophes and backslashes that result from the expansion are all backslashed (*backslashUnquoted*). This holds only for the variable and nested commands substitution, the other types either don't result in any quotes (arithmetical substitution), or are always unquoted (tilde expansion, file name matching) - for such types, *doubleBackslashReplace* is used (and no indication of being in quotes required). For the command substitution, *ArgExpandor* uses a special thread - created from within *performNestedCommand* and represented by the *nestedCmdThread* function.

## 6. Signal handler

The signal handler is represented by the *SigHandler* class. There's only one instance of it in the program, which is shared by all the *Env* instances. Upon startup, the default Ctrl-C event handler is set up – *ctrlCSignalHandler* – which resets the parser and sets a flag indicating that the command has been interrupted in *Env*, it also sends the shell's console input the `Esc` and `Enter` keys to reset the current line. For this, the *SigHandler* class must have a pointer to the instance of *WinShell* that created it (it's given in its constructor). The flag in *Env* is checked for by all the types of commands and upon the execution of a next one, a special exception is raised. However, the Ctrl-C signal in fact doesn't interrupt the innermost plain commands, for the flag is checked only once, upon the launch of a command.

The Ctrl-C signal handler may be changed to perform some given commands using the `trap` built-in, as well as other signals, but there is probably no situation upon which they could be raised by Windows. There is one method – *signalHandler*, that's always set as a handler (except for the default Ctrl-C handler) and performs the commands that it's given upon a signal raise – it decides which commands to launch by the signal number. The trap commands are performed in a dedicated thread, because Windows signal handlers must not contain such things as memory allocation and others.

*SigHandler* also supports the user-preset Bash events, such as `DEBUG` or `ERR` traps. These are triggered with the *triggerEvent* and *triggerDebugEvent* functions, their commands are executed synchronously. They also heed the `errexit` shell setting and may set a flag in the *Env* telling the shell to exit. There's also a function in *SigHandler* that lists all the names of signals (which are stored there in a hard-coded constant) – that is used by *Env* as a support for the `trap -l` command.

## 7. Internal commands

All the internal shell commands' functions are located in the *IntCmds* class. All the members of this class are static, that means no object needs to be created upon the launch of an internal command. Every command has its own main function, always called *cmd<command name>*, e.g. *cmdEcho* or *cmdPwd,* which may then call other supportive functions. The command functions are all declared `private`, so they must not be called directly. For all the commands' use, there is the main function, *exec*, which, given an *ArgVector* that includes a command name, searches for such command (in a list of string constants corresponding to function pointers) and launches the appropriate function – or returns false if no built-in has such name. This may be done because all the commands' main functions have the same set of parameters – the *ArgVector* with the commands, pointer to *Env* and line number and script file name indication (to display error messages correctly).

Most of the commands use the *getOpts* method to parse their options. It is also used by the `getopts` command and has therefore a wide variety of parameter

settings, it also supports the more complicated options of the `set` built-in. Other support functions, such as *declareVars* or *splitToVars* are only used by some specific built-ins. The supportive functions that may be called by more than one internal command have a pointer to its main function as a parameter, to display the error messages right. For the error messages, there's a special exception class – *IntCmdsException*, which has the command function pointer as one of its parameters and when displaying the message, it always prefixes it with the right command name.

## 6. Process communication

The process communication between the GUI and shell instances is done via Windows application messages (so-called "user-defined" message types from the range of constants not used by Microsoft). When the Bash application is run from within the GUI, the GUI associates it with its *Shell* object (by sending a `WM_ASSOC_MASTER` message) and from that time on it may be controlled by it and send back status messages (such as `WM_ASSOC_SLAVE` confirmation). In the association messages, the Window `HWND` of the shell GUI window and a handle to the shared memory area are always sent. The shared memory is a place where some vital informations about the current size and actualities of the Bash console window are stored, as well as a copy of the current shell window view.

### 1. Message thread in Bash

The Bash application has a special thread – encapsulated in a *MsgThread* object, to receive and send the messages. The messages are sent to it from the master application (the GUI) using the `PostThreadMessage` function. There's a usual Windows message loop in its thread function (*threadProc*) that lasts for all the application life. It handles only the messages from the master application – upon start-up, it waits for the `WM_ASSOC_MASTER` with the right parameters (for shells that aren't run under GUI this waiting never ends) – when it's given this message, it creates the shared memory and sends a handle back in `WM_ASSOC_SLAVE` (from *associateMaster*). From that time on, the communication is established and the message thread sends the main application refresh status messages (`WM_UPDATESCREEN`) which must be confirmed by the master (`WM_SCREENOK`). The screen is updated once in a while (pre-set at 50 ms, may take longer if the `WM_SCREENOK` message is not received) – a timer is always set to ensure this. The redrawing messages may be switched on or off on request from the master (`WM_SETTOPMOST`). The application may be also forced to close by the GUI – when it sends a `WM_CLOSEAPP` message, the *MsgThread* calls a function in *WinShell* telling it to exit (by a direct call to `ExitProcess`).

Bash, when being run under GUI, has its console window hidden and does not receive any input from Windows – all the input is sent by the master application either directly to the console window (the `WM_CHAR` messages) or also through the message thread (`WM_KEYUP, WM_KEYDOWN` that are then converted and written as console input by the corresponding *WinShell* object). This thread also ensures

that the console is resized properly, when the user sizes the GUI window (WM_RESIZECONSOLE).

The messages are also used for some less vital functions of the GUI-Bash cooperation, such as console window scrolling. There are four messages, telling the Bash application to scroll the console up or down by one line or whole page – *MsgThread* ensures this in the *scrollConsole* function. For the auto-complete feature of the GUI, the WM_AUTOCOMPLETE message is used. It calls the appropriate function on the *WinShell* object. The same is with WM_HISTORYFW and WM_HISTORYRW messages that provide the Bash history functionality. Only because of these messages, and them being handled from a different thread than the main shell one – asynchronously and instantly, must the input handles have their buffer protected by synchronization functions and CRITICAL_SECTION.

Several messages are also needed to provide the connection of file manager with the shell – the most important of them are WM_RUNCMD that triggers the execution of a command in the shell (the command is passed in as a handle to a shared memory area that contains it) or WM_CHANGEDIR to change the directory (may be sent in both directions, depending on whether the user changes the directory by typing in the console or entering it in file listing). To execute the commands synchronously (and only when the shell is waiting for some input from the user), there's a Windows event object, that is passed to the main input reading procedure as a second HANDLE to wait for. When a WM_RUNCMD or WM_CHANGEDIR message arrived, this event is set, the waiting ends with an exception that is catched by the WinShell object and the command may be run synchronously. Some other messages, such as WM_EXECUTEUNDERCURSOR and WM_ENABLE_PANELS are used to coordinate the functions of shell and file manager.

## 2. Shared memory

The shared memory is used as a storage space for all the GUI-Bash messages that need more than two pointer parameters (except command running and directory messages that use a shared memory of their own, which may be reinitialized every time a message is sent). It's created as a shared file memory mapping in the *associateMaster* function of the Bash message thread. The handle to the mapping is then duplicated and sent to the master application. It contains a buffer to store a copy of the current console view (must be as big as largest console window that may be created in the system), information about the size and position of the console screen buffer (which is updated as the console scrolls or is resized) and the cursor position, as well as a name of the program currently running in Bash (which is refreshed by Bash, and notifications are sent to the master application). The definition of the shared memory "file" format is located in *comm.h* file in the *COMMON* sources folder, along with the definition of all the window messages.

# 7. Doxygen-generated documentation

The BashCommander source codes have their in-line commentaries scheme designed for easy Doxygen documentation generation – the documentation is included with the sources.