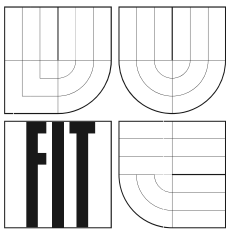
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ



Nástroj pro správu souborů v systému Mac OS X

Diplomová práce

2007 **Ondřej Ševčík**

Nástroj pro správu souborů v systému Mac OS X

Odevzdáno na Fakultě informačních technologií Vysokého učení technického v Brně
dne 24. ledna 2007

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Ing. Martina Hrubého
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

. . . . . .
Ondřej Ševčík
24. ledna 2007

Abstrakt

Diplomová práce popisuje historii operačních systému firmy Apple a vývoj aplikací pro Mac OS X. První část představuje dlouhý vývoj pro tuto platformu od raných začátků v roce 1974 se zaměřením na současný Mac OS X. Druhá část seznamuje čtenáře se základy vytváření aplikací pro Mac OS X s využitím frameworku Cocoa a jazyka Objective-C, který je skutečně objektovou nadmnožinou velmi dobře známého jazyka C. Praktickou částí této práce bylo vytvoření souborového manažera. Způsoby řešení konkrétních částí tohoto manažera jsou použity k vysvětlení základů vývoje. Práce obsahuje přesný návod jak vytvořit svou první aplikaci krok po kroku.

Klíčová slova

Souborový manažer, Apple, Mac OS X, Objective-C, Cocoa, Carbon, FreeBSD, XNU

Abstract

Thesis presents history of Apple's operating system and developing application for Mac OS X. First part introduces long evolution of macintosh's OS since early beginings in 1976 focused on latest Mac OS X. Second part makes reader acquainted with elements of creating applications for Mac OS X using Cocoa framework and Objective-C language which is real objective superset of well known C language. Practical part is developing file manager. Programing patterns from file manager are used for explaining fundamentals of developing. This contains exact directions how to create first application step by step.

Keywords

File manager, Apple, Mac OS X, Objective-C, Cocoa, Carbon, FreeBSD, XNU

Obsah

Kapitola 1

Úvod

Koupíte-li si jakýkoliv počítač z nabídky firmy Apple, dostanete v krabici nejen vytoužený pracovní nastroj, ale také operační systém (v současné době Mac OS X verze 10.4) s širokou paletu programů a nástrojů, které pokrývají většinu každodenních potřeb i velmi náročných uživatelů. Po vybalení počítače z krabice lze prakticky ihned začít surfovat na internetu, číst si poštu, plánovat si čas, chatovat s přáteli, upravovat a organizovat fotografie, stříhat videa a vytvářet z nich DVD. Pro správu souborů slouží integrovaný program nazvaný Finder. Přestože je Finder poměrně jednoduchý, pro většinu uživatelů bude dostačující díky velmi kvalitně vyřešenému drag-and-drop systému. Finder je aplikací podobnou Explorer v MS Windows nebo Konqueror v KDE bez integrovaného internetového prohlížeče. Některým lidem samozřejmě vyhovuje přetahování ikonek mezi dvěmi nebo více okny a to také dodnes praktikují, ale pro práci s větším množstvím souborů, například při programování, je vhodnější použít nějaký souborový manažer.

Nabídka tohoto druhu programů je na poli Mac OS X velmi omezená. Většinou lze použít některý ze souborových manažerů napsaných pro Linux, zde však narazíme na drobný problém a tím je nutnost mít aplikaci spuštěnou pod X11 serverem, což s sebou přináší řadu omezení. Kromě toho pro některé nestačí pouze X11 server, ale je nutná i řada dalších knihoven (pro Krusader je potřeba například Qt od Trolltech apod.). Provozování aplikací pod X11 není zrovna efektivní a práce s nimi může některé uživatele odradit. Velmi špatná je integrace s operačním systémem a nelze využívat jeho vlastnosti.

Další možností je Midnight Commander, který běží pouze pod terminálem a jeho instalace nepatří k těm nejjednodušším, protože mimo jiné je potřeba jej zkompilovat. To může představovat pro většinu uživatelů nepřekonatelnou překážku a myslím, že budou preferovat něco lépe integrované do grafického systému.

Patrně nejkvalitnějším souborovým manažerem pro Mac OS X je MuCommander napsaný v Javě. Java je do Mac OS X zakomponována poměrně velmi dobře. Uživatel na první pohled nepozná, že se jedná o jinou aplikaci, je ale dlužno přiznat, že rozhodně nepatří k nejrychlejším iplementacím. Tato aplikace je zřetelně pomalá i na počítači s procesorem G4 1,5GHz.

Jedinou aplikací, která je napsána v nativním Cocoa frameworku je Disk Order, který je komerční a obsahuje stále velké množství chyb. To je jeden z důvodů proč jsem se rozhodl vytvořit vlastní souborový manažer, který bude šířen jako open source. Druhým důvodem, proč jsem se rozhodl vytvořit tento projekt, je snaha proniknout do tajů programováni pro platformu Apple.

Rád bych vytvořil souborový manažer, který bude uživatelsky přívětivý, zapadne do Mac OS X systému a usnadní přechod uživatelům jiných platforem, kterým by mohl podoný nástroj chybět.

Diplomová práce navazuje na Semestrální projekt, ze kterého čerpá kapitoly 3.1 až 3.4. Na Ročníkový projekt diplomová práce nenavazuje.

Kapitola 2

Historie operačních systému Apple

2.1 Základní údaje o Apple

Steven Gary Wozniak a Steven Paul Jobs byli přátelé již od střední školy. Oba pracovali ve společnostech v Sillion Valley v Kalifornii v USA. S. Wozniak byl zaměstnancem Hewlett-Packard a S. Jobs pracoval zase u Atari. Zajímavý je Wozniakův zaměstnavatel, který mu ukládal podmínku, že veškeré jeho vynálezy musí být předvedeny právě v Hewllet-Packard, která si je mohla nárokovat. Když Wozniak vyvinul Apple I, který byl jen základní deskou v dřevěné krabici připojené k televizi, sklidil v HP jen posměch: „K čemu budou běžným lidem osobní počítače?“ Tím se otevřela brána firmě Apple Computer, která byla založena 1. dubna 1976 doslova v garáži.

Hlavní myšlenkou Apple bylo přivést na svět počítač pro obyčejného člověka. V 70. letech velmi výrazně pomohl nástupu osobních počítačů. Řada Apple byla první uvedená řada a byla zakončena velmi úspěšným modelem Apple II, který se dočkal i několika vylepšení. Těžiště obchodu byl a je vývoj a výroba počítačů, které jsou od počátku částí integrovaných systémů – software a hardware. Apple je většinou vyvíjí buď sám nebo podle svých specifikací zadává ostatním výrobcům hardware.

Apple také vyvíjí a zavádí produkty v jiných oblastech než je hardware a software. Mezi neúspěšné pokusy by se dal zařadit projekt Newton – kapesní počítač s rozpoznáváním psaného písma nebo řada digitálních fotoaparátů a mnoho dalších projektů. Jako konstrast působí projekt iPod, který si dobyl přední pozice na celosvětovém trhu. Myslím si, že tento špičkový mp3 přehrávač není potřeba blíže představovat.

Apple byl průkopníkem v mnoha oblastech, jako první představil světu komerčně úspěšné GUI a myš, které byli převzaty z projektů Xeroxu, kde byli odepsáni jako nepraktické. Svým produktům však od samého začátku Steve Jobs nasadil punc exkluzivity. Kromě toho, že to vedlo k rozkolům mezi ním a Wozniakem a poté i s dalšími, kteří chtěli počítače Apple opravdu pro všechny, způsobil tím paradoxně velmi malé rozšíření Mac počítačů mezi lidi. Schopný obchodník Bill Gates zahltil expandující trh svými produkty, protože se velmi výhodně spojil s tehdejším počítačovým gigantem IBM.

Poslední dominantou na trhu byl zmíněný Apple II, který měl být nahrazen špičkovým Apple III u nějž si Steve Jobs vynutil, aby neobsahoval žádný větrák, a tedy se stal nepoužitelým. Prostě se zavařil. Dalším adeptem na úspěch byla souběžně vyvíjená Lisa, která však byla drahá a také byla odsouzena k záhubě.

Apple se na začal ve velké míře vracet na trh koncem devadesátých let, kdy se do čela společnosti vrátil dříve vyhozený Steve Jobs. V současné době nabízí Apple řešení pro domácnosti, malé a střední kanceláře, pro profesionální fotografy, filmové a hudební tvůrce a další oblasti jako například školství. Za rok 2006 získal cenu za nejlepší marketing roku udělovaným časopisem *Mar-*

keting Daily za explozivní růst průmyslu kolem iPodu, rostoucí tržní podíl Maca a ohromě úspěšný provoz Apple Store obchodů.

2.2 Operační systémy Apple

2.2.1 Začátky Apple

Systém 6

Tato kapitola čerpá z [strf].

Přeskočím první systémy, které provázely počátky počítačů Apple, a budu se věnovat až tomu prvnímu, ve kterém se objevil Finder — souborový manažer. Koncem 80. tých let byl představen a používán operační systém System Software 6 často zkracovaným jen jako System 6. Tento operační systém patří do rodiny Classic MAC OS. Poslední stabilní verze byla 6.0.8 představená koncem roku 1990.

Kooperativní multitasking navržený Macintoshem debutoval v březnu 1985 programem nazvaném Switcher, který umožňoval uživatelům spouštět několik aplikací najednou a přepínat mezi nimi. Bohužel mnoho programů nefungovalo se Switcherem zcela korektně, proto nebyl s operačním systémem dodáván a zájemci jej museli od Applu získat explicitně. System Software 6 přinesl značně vyspělejší přístup nazvaný MultiFinder, který byl poprvé představen v System Software 5.

MultiFinder byl očekávanou odpovědí na požadavky uživatelů. První Macintosh počítače byly limitovány množstvím paměti, které vedlo vývojáře Apple k brzkému opuštění multi-taskingu, který Apple vyvíjel pro operační systém Lisa. Aby byl umožněn alespoň malý stupeň volnosti, byla přidána aplikace Desk Accesories, která byla sice velmi omezená, ale nepotřebovala tolik paměti. Každý nový model Macintosh měl více paměti než předchozí, až se u stroje Mac Plus stalo standardem 1MB RAM. Toto množství paměti bylo již dostatečné pro nějakou formu multitaskingu, který byl poprvé implementován pravě programem Switcher. Switcher vyhradil několik míst v paměti, do kterých se aplikace mohla nahrát. Uživatel mohl pak přepínat mezi těmito aplikacemi kliknutím na malé tlačítko v menu. Aktuální aplikace se horizontálně odsunula mimo obrazovku a nahradila ji nová aplikace. Navzdory této neobratnosti, byl zvolený přístup vyhovující současnému systému správy paměti a aplikace nepotřebovaly žádnou úpravu pro práci se Switcherem.

MultiFinder rozšířil tento systém naznačený Switcherem v mnoha ohledech. Kromě toho, že přiděloval každé aplikaci čas CPU, poskytoval možnost, jak umožnit oknům různých aplikací koexistovat společně s využitím modelu aplikační vrstvy. Pokud byla aplikace aktivní, všechna její okna byla přenesena do popředí jako jedna vrstva. Tento přístup byl potřebný pro zpětnou kompatibilitu s mnoha okenními datovými strukturami, které již byly zdokumentovány. Také poskytoval způsob, jak mohly aplikace získat místo paměti dopředu podle toho, kolik ji budou potřebovat. MultiFinder tak mohl alokovat kus paměti podle potřeb. Tento přístup, přestože byl funkční, způsobil řadu omezení, která ústila v mnoho problémů pro uživatele. Týkala se převážně správy paměti. Integrace MultiFindru do System 7 tyto problémy neodstranila. MultiFinder přežil v útrobách operačního systému až do verze Mac OS 9. Pouze přechod na nový moderní preemptivní multitasking založený na Unixu v Mac OS X odstranil všechny jeho nedostatky.

Multitasking byl pod System 6 volitelný. Uživatel si mohl vybrat zda chce spustit Finder nebo MultiFinder. MultiFinder umožňoval uživatelům vidět několik oken běžících aplikací najednou, vidět ikony Finderu jako byl například Trash (koš) nebo okna aplikací běžících na pozadí.

System 7

Tato kapitola čerpá z [strg].

Apple vydal první verzi System 7 (nesoucí kódové označení Big Bang – Velký třesk) 13. května 1991. System 7 se používal v několika verzích až do roku 1997. Jednalo se o přímého následníka System Software 6, ze kterého přebíral a vylepšoval mnoho funkcí jako kooperativní multitasking, virtuální paměť, sdílení souborů, výrazně vylepšený vzhled uživatelského prostředí (poprvé se objevily barvy), QuickTime a QuickDraw 3D. Apple odstranil z názvu slůvko „software“ a označení System 7 se vžilo pro všechny verze 7.x. S vydáním verze 7.6 v roce 1997 byl odstraněn i termín „System“ a operační systém byl oficiálně přejmenován na Mac OS.

Operační systém nabízel vestavěný kooperativní multitasking, který předchozí systém nabízel jako volitelnou součást. Neodstranil sice jeho nedostatky, ale byl pro uživatele příjemnější. Trash (koš) byl implementován jako adresář, který umožňoval souborům přetrvat i restart systému. System 7 umožnil sdílení souborů přes AppleTalk.

Nově se objevily aliasy, které představovaly obdobu unixovým linkům. Měly typicky velikost od 1-5kb. Mohly ukazovat na libovolný objekt v souborovém systému jako byly například dokumenty, aplikace, složky, diskové oddíly, síťové sdílení, vyměnitelné médium nebo tiskárna. Při přístupu na tento malinký soubor se systém choval stejně jako by uživatel přistupoval přímo na tento soubor. Otevření pomocí standardního dialogu otevřelo samozřejmě originální soubor. Na rozdíl od způsobu založeném na cestě implementované v Microsoft Windows 95, alias ukazoval přímo na soubor a mohl fungovat, i když byl originální soubor přesunut nebo přejmenován.

Některá systémová rozšíření byla přesunuta do zvláštních složek a separována ze složky systémové (tak jako tomu bylo v předchozích verzích) a tím bylo uživateli umožněno přidržením klávesy Shift, během nabíhání systému, jejich zakázání. Pozdější verze System 7 nabízela Manažer rozšíření (Extensions Manager), který zjednodušil proces výběru použitých rozšíření podle požadavků uživatele. Rozšíření byla častým zdrojem nestability systému a jejich vybíráni umožnilo snadnější řešení problémů.

Dalšími vylepšeními, která System 7 přinesl, byly například TrueType fonty (dřívější fonty byly pouze bitmapové nebo bitmapové párované s Postscriptovými fonty), které přinesly kvalitně vypadající fonty jak na obrazovce tak vytištěné. Nově se objevilo plně barevné uživatelské rozhraní, které však zůstalo volitelné, aby uživatelé starších počítačů s monochromatickým displejem mohli využít staré černo-bíle rozhraní. Nové Sound Manager API verze 3.0 nahradilo původní staré proprietární API. Nové rozhraní přineslo znatelně vylepšenou hardwarovou abstrakci a kvalitnější reprodukci.

System 7 přinesl plně 32-bitový adresovatelný prostor proti předchozímu 24-bitovému. To umožnilo všem rutinám operačního systému být plně 32-bitové proti předchozím, u kterých byla horní část slova použita pro rozšíření adresového prostoru. Přestože byl System 7 čistě 32-bitový, mnoho počítačů a aplikací ještě ne a přechod ještě nějaký čas trval. Pro zjednodušení přechodu obsahoval „Memory control panel“ přepínač, který umožňoval zpětnou kompatibilitu se staršími aplikacemi.

System 7.5 a další umožňovaly přístup na CD a diskety formátované jako FAT (MS-DOS) pomocí ovládacího panelu PC Exchange, který byl dodáván zdarma právě s verzí 7.5. V předchozích verzích nebylo možno přistupovat na disky DOS, kromě System 7.1 Pro, který také obsahoval PC Exchange za poplatek.

V době jeho uvedení mnoho uživatelů pocítilo pokles výkonu jako důsledek přechodu z System Software 6 na System 7 a byli nuceni pořídit si nový hardware, aby vyrovnali rozdíl v rychlosti. Dalším problémem byla větší potřeba paměti oproti předchozímu systému. System Software 6 byl schopen běžet na jedné disketě a vyžadoval 600kB paměti, zatímco System 7 potřeboval více než 1MB a nemohl být efektivně provozován na strojích vybavených pouze disketovou mechanikou. (Verze 7.5 byla sice schopna bootovat pouze z diskety, avšak na ní již nezbylo žádné místo pro programy. Ty bylo možno spouštět pomocí AFP serveru přístupného pomocí AppleTalk sítě.) Jakmile

byly počítače Mac dodávány s dostatkem paměti, stalo se používání System 7 opravdu pohodlným.

System 7 byl velmi rychle přijat uživatelskou komunitou Mac. Až do příchodu operačního systému Mac OS X byl System 7 beze sporu největší inovací od počátku

2.2.2 Vývoj systému nové generace

Copland

V roce 1989 plánovali manažeři Apple budoucnost jejich operačního systému. Všechny nápady byly zapsány na kartičky. Ty, které mohly být implementovány v krátkém čase (jako například přidání barev do uživatelského rozhraní), byly zapsány na modré karty, zatímco pokročilé nápady (jako byl objektově orientovaný souborový systém) byly zapsány na růžové karty. Vývoj nápadů na obou sadách kartiček probíhal paralelně na dvou projektech nazvaných prostě „blue“ (modrá) a „pink“ (růžová). Apple naplánoval, aby „modrý“ tým vydal aktualizaci verze existujícího operačního systému Macintosh v letech 1990-1991, zatímco růžový tým měl vydat zcela nový operační systém kolem roku 1993.

„Modrý“ tým přišel se System 7 na konci roku 1991, ale „růžový“ tým se potýkal s velkými problémy a neustále odsouval vydání systému do neurčité budoucnosti. Implementace nového systému byla složitější, než byly původní plány s postupem času se objevovaly nové problémy. Díky stále většímu zpoždění „růžového“ projektu odcházeli někteří vývojáři raději k „modrému“ týmu a „růžový“ tak neustále bojoval s nedostatkem pracovních sil, zatímco vedení tento problém ignorovalo. Časem Apple opustil „růžový“ projekt jeho převedením na projekt Taligent.

Původní záměr podpory jednouživatelského systému, na kterém by běžela jedna aplikace, na bez-síťovém stroji s disketovou mechanikou, nevyhovoval mnoha rostoucím uživatelským potřebám. Architektura QuickDraw byla velmi obtížně implementovatelná v multitaskingovém OS a souborový systém byl velmi neefektivní při použití pevných disků.

Tehdejší vedení, v jehož čele byl John Sculley, tou dobou ignorovalo řízení vývoje a orientovalo především na prodej a marketing. Výsledkem bylo, že vývojová oddělení neměla jasná vedení.

S vydáním System 7.5 na podzim 1994 se Apple rozhodl, že vývoj starých Macintosh operačních systémů by měl skončit. Tehdejší systémy nejen postrádaly spoustu vlastností, které se očekávaly od moderního operačního systému, ale rostla jejich nestabilita díky nedostatečné ochraně paměti. Stále častější pády znepokojovaly uživatele a začaly být vážným problémem. Bylo jasné, že Apple musí vyvinout nový a dokonalejší operační systém, který by byl schopen konkurovat přicházejícím Microsoft Windows.

Výsledkem tohoto snažení bylo operační systém nové generace s kódovým označením Copland [1]. Měl představit chráněnou paměť, multitasking a množství funkcí, které by byly kompatibilní s většinou již existujícího softwaru pro Mac. Copland se spouštěl nad mikrokernelem nazvaným Nukernel, který ošetřoval základní úlohy jako spouštění aplikací a správu paměti, nechávající ostatní úlohy jiným specializovaným programům nazvaných servery. Například síťové a souborové služby jádro neumělo poskytovat, ale využívalo serverů, které tyto úlohy řešily. Copland se skládal z kombinace Nukernelu, několika serverů a balíku aplikačních podpůrných knihoven, známých jako Macintosh programming interface (programovací rozhraní Macintosh).

Nové aplikace napsané přímo pro Copland byly schopny komunikovat přímo s novými servery a tím těžit z výhod jako výkon nebo škálovatelnost. Komunikovaly také přímo s jádrem, pokud vyžadovaly nějaké specifické funkce jako spouštění nových vláken, z nichž každý proces měl vlastní chráněnou paměť, tak jak je zvykem v moderních operačních systémech. Tyto oddělené aplikace však nebyly schopny využívat QuickDraw pro kreslení a nemohly mít tedy uživatelské rozhraní.

[1] systém byl pojmenován po americkém skladateli Aaronu Coplandovi

Apple navrhoval, aby větší programy umístili své uživatelské rozhraní do normální Macintosh aplikace, které potom spustí samostatné vlákno vykonávající práci externě. To bylo poměrně obtížné, neboť vícevláknový kód se složitěji píše a ladí.

Další vlastností Coplandu byla jeho nativní podpora pro procesor PowerPC. Tím bylo dosaženo mnohem vyšší rychlosti, protože se nemusely žádné instrukce emulovat. Výzvou Coplandu bylo, aby se vměstnal do hardwaru, které tehdejší Mac počítače nabízely. System 7.5 tehdy používal kolem 2,5 MB paměti a to byla na většině tehdejších počítačů veškerá dostupná paměť. Aby Copland byl úspornější, využíval systém velmi přísnou správu paměti a hojné využívání sdílených knihoven, s čímž se podařilo udržet Copland o pouhých 50% větší než 7.5.

Za zmínku stojí, že Copland nepodporoval přímý multithreading z původních knihoven Mac OS, a navíc Nukernel nepodporoval symetrický multiprocesing, což znamenalo že systém nemůže nativně využívá více než jeden procesor na víceprocesorovém systému. Obě tyto funkce se však od moderního systému očekávaly, ale Copland si kladl za cíl přinést nové API pro programátory, aby mohli začít přesouvat své aplikace na tento operační systém.

První části Coplandu, mezi nejpozoruhodnější patřila raná verze souborového systému, byly prezentovány na WWDC v květnu 1995. Apple také slíbil první vydání betaverze Coplandu koncem roku a vydaní plné verze počátkem roku 1996. Během roku Apple vydal mnoho prototypů do nejrůznějších časopisů aby představil nový systém, avšak na konci roku byla vývojová verze v nedohlednu.

Brzy projekt vypadal ne jako nový operační systém, ale jako obrovská sbírka nových technologií. QuickDraw GX (náhrada za QuickDraw) a OpenDoc (multiplatformní standardní softwarový framework práci s dokumenty inspirovaný Xerox Star systémem a zamýšlený jako alternativa k Objekt Linking and Embedding - OLE - představený Microsoftem) se staly základními komponentami systému. Seznam nových vlastností rostl rychleji než mohl být tyto vlastnosti vůbec dokončeny, objevily se nová a přepracovaná dialogová okna (například pro otevíraní souborů) a podpora témat. Tak jak se rozrůstalo množství balíčků, jakékoliv jejich kvalitní otestování se stávalo stále těžší a těžší.

Na WWDC v roce 1996 nový generální ředitel Gil Amelio hovořil exkluzivně o Coplandu, který byl představen jako Mac OS 8. Neustále uváděl, že jeho vývoj je jediným cílem inženýrů Apple. Amelio oznámil, že během několika měsíců bude uvolněna verze pro vývojáře a plná verze bude uvedena koncem podzimu. Na konferenci bylo prezentováno, jen velmi málo ukázek běžícího systému. Místo toho byly prezentovány nejrůznější kousky nových technologií. Bylo prezentováno jen velmi málo z funkčnosti jádra a nový souborový systém prezentovaný o rok dříve chyběl úplně.

Amelio slíbil na konci, že do operačního systému bude přidána podpora více procesorů. To mlčky předpovědělo odsunutí systému na neurčito, protože tato funkce je tak významnou součástí jádra, že bylo nemožné ji přidat a odladit do oznámeného data vydání.

V srpnu 1996 byla vydána „Vývojová verze 0“ a byla zaslána vybraným partnerům. Slibovaná stabilita byla velmi vzdálená té skutečné a systém havaroval i při základních operacích a pro účely vývoje aplikací byl zcela nepoužitelný. V říjnu bylo datum vydání systému opět oddáleno někam až do roku 1997. Nejvíce překvapeno bylo hardwarové oddělení Applu, které čekalo na Copland aby mohlo předvést PowerPC v celé své kráse. Objevily se vtipné poznámky oddělení pro dohled nad kvalitou, že vzhledem k množství chyb v systému by mohl být systém nachystán v roce 2030.

Koncem léta se situace nelepšila a Amelio se rozhodl udělat rázný krok. Najal Ellen Hancockovou z National Semiconductor, aby převzala vedení nad Coplandem a vrátila jej k životu. Ta však po několik měsících zjistila, že situace je beznadějná a Copland by neměl být vůbec nikdy vydán. Zjistila také, že se během toho, co se vývojáři zamýšleli nad tím, že Gershwin (následník Coplandu) bude „plně moderní“ systém, nikdo si nebyl zcela jist správnou koncepcí v porovnání s Coplandem a nikdo nebyl na tuto práci přidělen.

Doporučila, že by se mělo pokračovat na zlepšování stability stávajícího Mac OS, zatímco by se hledala technologie mimo firmu Apple pro nový operační systém. Když byla představena „Vývojová verze 1“, zrušil Apple oficiálně vývoj Coplandu.

Dalším plánem Hancockové bylo pokračovat ve vývoji System 7.5 s využitím technologií původně navržených pro Colpand a zapracovat je do systému. Stabilita a výkon byly vylepšeny ve verzi Mac OS 7.6, která odstranila označení „System“. Mnoho vlastností, včetně Platinum GUI navrženého pro Copland nebo podpora témat, bylo přeneseno do Mac OS 8, který byl původně plánován jako Mac OS 7.7. Vývoj pokračoval v této větvi a došel až k Mac OS 9.2.2 vydaný v prosinci 2001, kdy byl vývoj tohoto systému definitivně ukončen.

Plněn byl i druhý plán Hancockové a Apple začal hledat produkt jiné společnosti, který by mohl zakoupit. Po dlouhých diskuzích s Be a některé fámy uvádí možné plány o spojení se Sun Microsystems, bylo pro mnohé překvapením, že Apple zakoupil NeXT Computer v prosinci 1996. Projekt portování OPENSTEP na platformu Macintosh byl pojmenován Rhapsody a byl posléze vydán jako Mac OS X.

Řada vlastností představených v demech Coplandu, jako pokročilý vyhledávací nástroj, vestavěný Internetový prohlížeč, podpora videokonferencí se znovu objevila v pozdějších vydáních Mac OS X.

2.2.3 Classic Mac OS

Mac OS 8

Tato kapitola čerpá z [stra].

Tato verze operačního systému podporovala přechod mezi hardwarovými platformami Macintosh. Nejstarší verze podporovaly Mac počítače s procesorem Motorola 68040, jeho nejmladší verze (Mac OS 8.1 a 8.5) se dodávaly s počítači založených na PowerPC G3, jako byly například první iMac a PowerMac G3. Mac OS 8 využíval až do verze 8.6 monolitické jádro, které bylo v této verzi nahrazeno nanokernelem.

Mac OS 8.0 První verze byla uvedena 26. července 1997 a byla vyvíjena pod kódovým označením Tempo. Vylepšení proti System 7 obsahovala multi-vláknový Finder, třírozměrný vzhled platinum, zlepšení výkonnosti virtuální paměti, spouštění AppleScriptu a spouštění systému. Nápověda byla přístupná pomocí Info Centra (HTML stránek uložených na disku s odkazy do Internetu). Stejně jako předchozí systém nabízel balónovou nápovědu a průvodce (Apple Guide). Byl to první klientský operační systém nabízející AFP přes IP jako rozšíření standardu AppleTalk. Tato verze také představila „Simple Finder“, jako volitelnou konfiguraci Finderu, který redukoval menu na základní operace, aby umožnila jednoduchý přechod novým uživatelům.

Mac OS 8.1 Tato verze byla představena 19. ledna 1998 a byla to poslední verze, která byla schopna běžet, jak na procesorech Motorola 680x0 tak PowerPC. Byl představen nový volitelný souborový systém HFS Plus (známý také jako Mac OS Extended Format), který podporoval velké soubory, dlouhé názvy a lépe využíval místo na velkých discích díky zmenšení velikosti bloků. Mac 8.1 také přinesl vylepšenou verzi PC Exchange, umožňující uživatelům vidět dlouhé názvy souborů vytvořených na PC pod Windows 95. Tato verze byla první, která umožňovala běh aplikací napsaných pro prostředí Carbon.

Mac OS 8.5 Vydání této verze se datuje k 17. říjnu 1998 a byla to první verze Mac OS běžící pouze na Mac vybavených procesory PowerPC. Ne všechen kód pro 680x0 byl nahrazen kódem pro PowerPC, závisel výkon systému na emulaci 680x0.

Mac OS 8.5 nabízel mnoho výkonnostních vylepšení. Kopírování souborů přes síť bylo rychlejší než kdykoliv před tím a Apple prohlašoval, že je „rychlejší než Windows NT“. AppleScript byl také přepracován pro použití pouze PowerPC kódu, čímž se znatelně zvýšila jeho rychlost.

Tato verze operačního systému byla první, která podporovala témata, která umožňovala měnit základní vzhled systému nazvaný Apple Platinum na Gizmo a HiTech. Tato radikální změny počítačového vzhledu byla na poslední chvíli odstraněny a objevila se pouze ve verzi beta (známé jako Mac OS 8.2), nicméně uživatelé si mohli vytvářet svá vlastní témata a používat je ve svém Mac OS. Kromě změny témat byla 8.5 první verzí, která podporovala 32-bitové ikony. Ikony tak měly 16,7mil barev a 8-bitový alfa kanál umožňující jejich průhlednost.

Mac OS 8.6 poslední stabilní verze Mac 8.6 byla vydána 10. května 1999 a přicházela s nanokernelem, který zvládal preemptivní multitasking s rozhraním Multiprocessing Services 2.x a vyšší. Stále však chyběla separace procesů a systém stále používal kooperativní multitasking. Přesto přinášel tento update, který byl pro uživatele systému Mac OS 8.5 zdarma, vyšší stabilitu i rychlost, než předchozí verze a je považován za nejstabilnější Classic OS systém. Mnoho nového hardwaru vyžadovalo jako své minimum Mac OS 8.6.

Mac OS 9

Tato kapitola čerpá z [strb].

Tato verze je poslední verzí známou jako Classic OS, tedy klasický Macintosh operační systém, a byla představena 23. října 1999. Při představování Mac OS 9 bylo uvedeno na 50 nových vlastností. Projekt s kódovým označením Sonata, jež byl původně zamýšlen jako verze Mac OS 8.7, je některými označován jako nejbohatší a nejstabilnější verze originálního Mac OS. Stále však systém neobsahoval vlastnosti moderních operačních systému, jako byla chráněná paměť (která byla údajně implementována v beta verzi Mac OS 9.1, ale na příkaz Steva Jobse odstraněna) a preemptivní multitasking (ten byl experimentálně implementován v nanokernelu, který se objevil ve verzi 8.6). Mac OS 9 si ponechal vzhled Platinum převzatý z verze 8, i když bylo možno jej nahradit tématy třetích stran bez nutného hackování.

Apple označoval Mac OS 9 za „dosud nejlepší Internetový operační systém“, velmi prosazoval svůj software Sherlock 2, který umožňoval vyhledávání jak na disku tak v mnoha on-line zdrojích. Další integrovanou vlastností bylo zavedení podpory Internetových nástrojů nazvaných iTools (nyní .Mac) a dalších vylepšení.

Poslední stabilní verzí byl Mac OS 9.2.2 z prosince 2001 a v květnu 2002 na světové konferenci v San Jose, CA připravil Steve Jobs symbolický smuteční obřad za ukončení vývoje Mac OS 9, čímž skončila éra Classic Mac OS.

PowerPC verze Mac OS X obsahuje kompatibilní vrstvu zvanou Classic, která umožňuje běh kompletní instalace Mac OS 9 v rámci Mac OS X pro aplikace a hardware vyžadující OS 9. Většina aplikací běží v Classic velmi dobře, i když některé z nich mají problém s překreslováním obrazovky a některé řadiče nefungují vůbec.

Nová verze přinesla kromě výše zmíněných vylepšení stability hlavně množství aplikaci. Mimo jiné Network Browser, aplikaci, která umožňuje procházet sítí a připojovat se na jiné počítače. Dříve byly tyto úlohy poskytovány přes aplikaci Chooser. Aplikace Sherlock 2 sloužila pro vyhledávání na Internetu i na disku. PlainTalk 2 vylepšoval syntézu a rozpoznání řeči. Rozšířen byl také Finder, který nyní umožňoval kryptování a vypalování CD. Software Update umožňoval bezpečně aktualizovat systém přímo z Control Panel.

2.2.4 Cesta k Mac OS X

Hledání operačního systému nové generace

Apple se vydal cestou hledání nového operačního systému mimo svou firmu. Některé prameny uvádí, že se krátký čas uvažovalo i o partnerství s Microsoftem a vytvoření Apple OS založeném na

technologii Windows NT. Mezi adepty na nový operační systém byl také Solaris od Sun Microsystems, ale především BeOS od Be. Nejvážněji vypadala jednání právě s Be a vše směřovalo k dohodě. Be bylo založeno Jeanem-Louisem Gasséem, bývalým vedoucím vývojového oddělení, a jeho tým vytvořil impozantní operační systém. BeOS měl ochranu paměti, preemptivní multitasking, symetrický multiprocesoring, tedy všechny vlastnosti po kterých Apple volal, a byl také schopen běžet na PowerPC (později byl portovaný i na x86). BeOS měl velmi dobře navrženou práci s multimédii, ale nebyl dokončený a řádně otestovaný. Kupříkladu chyběla podpora sdílení nebo tisku a také nebylo k dispozici příliš mnoho aplikací pro tento operační systém. Dalo se tedy předpokládat, že Gassée by rád viděl pokračování svého systému v počítačích Apple, a požadoval částku okolo $400-$500 mil. V té době dosahovaly investice do Be 20 milionů USD a maximální nabídka Applu byla 125 milionů USD. Jednání sice probíhala dlouho, ale nakonec se obchod neuskutečnil.

Kromě s Be paralelně probíhala jednání s dalším uchazečem: NeXT Steva Jobse, jehož operační systém byl prověřen trhem na rozdíl od BeOS. Navzdory tomu, že NeXT neměl dostatek prostředků, OPENSTEP byl velmi dobře přijat na firemním trhu. Steve Jobs dohazoval velmi důrazně Applu svou NeXT technologii a zdůrazňoval, že OPENSTEP byl současnému trhu o několik let napřed. Apple firmu zakoupil v únoru 1997 za cenu přes 400 milionů USD. Tehdejší generální ředitel Gil Amelio poznamenal: „Vybrali jsme si plán A namísto plánu B“.

Tou dobou Apple se potýkal s existenční krizí. Za vedení Ameila se ocitly akcie Applu nejníže za posledních 12 let a na konci roku 1997 byla vyhlášena ztráta 708 milionů USD. Amelio byl kritizován za nedostatek vizí a marketingových schopností. Amelio opustil v červenci 1997 společnost a byl dočasně nahrazen Stevem Jobsem.

NeXT Chapter – další kapitola

NeXTSTEP je původní objektově orientovaný, multitaskingový operační systém vyvinutý firmou NeXT Computer k běhu na vlastních proprietárních počítačích NeXT. NeXTSTEP 1.0 byl poprvé představen v září 1989 po několika předváděcích verzích z nichž první se objevila v roce 1986. Poslední stabilní byla verze 3.3 na počátku roku 1995, která byla schopna běžet na procesorech Motorola 68k, x86, Sun SPARC, HP PA-RISC. Od verze 3.2 NeXT začal spolupracovat se Sun Microsystems na vývoji OPENSTEP, jenž by byl multiplatformní implementací založené na NeXTSTEP.

NeXTSTEP systém založený na Unixu s využitím jádra Mach s využitím kódu z BSD Unix. Jádro bylo hybridní a spojovalo výhody jak monolitického jádra tak mikrojádra. Pro zobrazování se využíval Display PostScript. Systém byl objektově orientovaný a byl napsán v Objective-C včetně aplikační vrstvy. Velkou výhodou bylo, že obsahoval vývojové prostředí pro objektově orientovanou vrstvu.

Poskytované prostředky nabízely neuvěřitelnou sílu a byly využity veškerého softwaru. Jazyk Objective-C umožňoval vytvářet aplikace pro NeXTSTEP mnohem jednodušeji než pro ostatní konkurenční systémy. Dokonce i o mnoho let později bylo na tento systém poukazováno jako na vzor.

Grafické rozhraní bylo konzistentní a představilo myšlenku aplikace Dock, která se uchovala i v OPENSTEPu a přešla s ním i do Mac OS X. NeXTSTEP byl mezi prvními systémy, které přinesly koncepty grafického rozhraní, které jsou v dnešních systémech zcela běžné: drag-and-drop v rámci celého systému, rolování a přesouvání oken v reálném čase, dialogové boxy vlastností („inspektory“) a další. System byl také mezi prvními víceúčelovými uživatelskými rozhraními, které měly barevné standardy, průhlednost, pokročilé zvukové a hudební zpracování (pomocí Motorola 56000 DSP), pokročilé grafické primitivy, vícejazyčnost a konzistentní chování napříč všemi aplikacemi.

Hlavním nástrojem uživatele NeXTSTEPu byl beze sporu Workspace Manager sloužící jako grafický správce při interaktivní práci v systémovém prostředí. Workspace Manager přinesl novou techniku využívání aplikací. Často používané aplikace mohl uživatel přemístit na „application dock“, kde mu byly k dispozici po celou dobu práce se systémem. Každý systémový objekt mohl být zobrazen v okně inspektoru.

OPENSTEP

OPENSTEP API bylo vytvořeno během spolupráce společností NeXT Computer a Sun Microsystems, která oddělila objektovou vrstvu systému NeXTSTEP tak, aby byla schopna běžet v operačním systému Solaris (přesněji v Solaris na počítačích založených na SPARC). Hlavním úsilím bylo odstranit ty části NeXTSTEPu, které byly závislé přímo na jádře Mach nebo specifickém hardware využívaném v počítačích NeXT. Výsledkem byl menší systém který se skládal hlavně z Display PostScriptu, Objective-C runtime prostředí, překladačů a většiny knihoven Objective-C z NeXTSTEPu.

První verze API byla prezentována firmou NeXT v létě 1994. Později téhož roku byla vydána verze OPENSTEP jako verze jejich vlajkové lodi operačního systému NeXTSTEP běžící na několika podporovaných platformách a byla přejmenována na OPENSTEP. Tato verze OPENSTEPu byla schopna běžet na hardwaru Sun SPARC nezávisle na operačním systému Solaris. OPENSTEP zůstal hlavním produktem firmy NeXT až do její koupě Applem v roce 1997.

OPENSTEP API se od předchozího NeXTSTEPu lišilo v několika věcech. OPENSTEP popisoval pouze vyšší vrstvu knihoven a služeb (jako například Display PostScript). Dále odstraňoval veškerý kód, který byl závislý na jádře Mach a mohl tak běžet v rámci libovolného operačního systému. Nízkoúrovňové objekty jako například řetězce byly v NeXTSTEPu reprezentovány datovými typy z C, zatímco OPENSTEP přinášel množství nových tříd (`NSString`, `NSNumber` atd.) a odstraňoval tak problémy s interpretací dat v paměti na různých platformách (big-endian/little-endian). Stejně tak další nízkoúrovňové datové typy poskytující standardní funkcionalitu byly přepsány tak, aby byly nezávislé na použitém zařízení (např. třída `NSArray`). Soubor těchto tříd (framework) byl nazván „Foundation Kit“ nebo zkráceně „Foundation“. Správa paměti při programování se vyvinula z prostého alloc/free mechanismu v nový přístup využívající retain/release (zabrat/uvolnit). Pokud některá část kódu potřebuje uchovat objekt pro svoje potřeby, jednoduše ji zabere a jakmile ji přestane využívat oznámí to garbage collectoru. O správné uvolnění z paměti se postará poloautomatický garbage colector. API také obsahovalo kromě výše zmíněného frameworku Foundation Kit ještě jeden, který sloužil pro vytváření GUI a grafického prostředí nazvaný „Application Kit“.

Pro OPENSTEP bylo vyvinuto několik nových balíčků knihoven, které byly později převzaty do platformy OPENSTEP. Na rozdíl od operačního systému jako celku, byly tyto balíčky navrženy tak, aby byly schopny běhu prakticky v libovolném OS. Myšlenkou bylo použít částí OPENSTEP jako základ síťových aplikací běžících na různých platformách.

Nejdůležitější mezi těmito balíčky bylo PDO (Portable Distributed Objects). PDO bylo velmi ořezanou verzí OpenStep obsahující pouze technologie z Foundation Kit, kombinované s novými knihovnami poskytujícími vzdálená volání. Na rozdíl od OPENSTEPu, který definoval operační systém, ve kterém aplikace poběží, byly aplikace s PDO knihovnami zkompilovány do samostatných aplikací, které běžely pod danou platformou nativně. PDO bylo dost malé na to, aby se dalo lehce portovat a byly vydány verze pro všechny hlavní dodavatele serverů.

Jádro Mach

Tato kapitola čerpá z [strd].

Mach je jádro operačního systému vyvinutém na Univerzitě Carnegie Mellon. Bylo určeno především pro distribuované a paralelní výpočty. Je jedním z prvních zástupců skupiny mikrokernel a stále platí za standard mezi obdobnými projekty.

Projekt běžel v Carnegie Mellon mezi lety 1985 a 1994 a končil verzí Mach 3.0. Řada dalších nadšenců pokračovala ve vývoji jádra Mach. Například Univerzita Utah představila Mach 4. Mach byl vyvinut jako náhrada jádra používaného v BSD verzi Unixu a nemusel byt tedy kolem něj vyvíjen další nový operační systém. V dnešní době je vývoj prakticky ukončen a jeho derivát (XNU) užívá právě Mac OS X, který jej převzal z OPENSTEPu.

Mach je následníkem Accent kernelu vyvíjeném také na Carnegie Mellon Univerzitě. Hlavním vývojářem byl Richard Rashid, který od roku 1991 pracoval pro Microsoft na nejvyšších pozicích ve výzkumné divizi Microsoft Research. Další vývojář Avie Tevanian byl kdysi v čele NeXT software a od roku 1997 šéfem softwarového oddělení v Apple Computer až do března roku 2006.

Protože Mach byl navržen jako náhrada za tradiční Unixové jádro, zaměřím se na hlavní rozdíly, které jsou mezi těmito jádry. S postupem času se jevil koncept Unixu, kde všechno bylo jako soubor, na moderních systémech ne tak docela efektivní. Nějaká radikální změna by však mohla poškodit flexibilitu původního konceptu. Kritickou abstrakcí v Unixu byly roury - pipe. Bylo potřeba mít koncept obdobný rourám, který by pracoval více obecněji, dovoloval přenos nejrůznějších informací mezi dvěma procesy. Takovýto systém existoval IPC (inter-process communication). Je to systém obdobný rourám, který umožňuje zasílání libovolných informací mezi dvěma procesy. Je protikladem k předávání informací přes soubory.

Carnegie Mellon Univerzita experimentovala s touto koncepcí na projektu Accent kernel a využívala IPC systém založen na sdílené paměti. Toto jádro bylo čistě experimentální a jeho využití pro vývoj bylo omezeno, protože nebylo kompatibilní s Unixem, který v té době již platil za standard pro většinu výzkumu v oblasti operačních systémů. Navíc Accent bylo silně úzce svázáno s hardwarem, pro který byl vyvinutý, a začátkem 80. let byla nasnadě expanze zcela nových platforem.

Mach vznikl jako snaha vytvořit Unixově orientovaný, čistě navržený a vysoce přenositelný Accent. Základní výčet z konceptu:

- „úloha“ je množina zdrojů umožňující běh „vlákna“
- „vlákno“ je samostatná celek běžící na procesoru
- „port“ definuje zabezpečenou rouru pro IPC mezi úlohami
- „zprávy“ jsou předávány mezi programy pomocí portů

Mach byl vyvinut na konceptu IPC převzatého z Accentu, ale byl navržený tak, aby byl kompatibilní s Unixem a umožňoval běh Unixových programu s žádnými nebo minimálními úpravami. Mach představil koncept portů, reprezentující dvoucestný IPC. Porty byly zabezpečeny a měly práva tak jako soubory pod Unixem, a umožňovaly Unixový model správy ochrany. K tomu Mach umožňovaly aby měl libovolný program práva, která by mělo standardně pouze jádro, aby mohly programy běžící v uživatelském módu přistupovat přímo k hardwaru.

Stejně jako Unix se i operační systém založený na Mach skládal z kolekce základních programů tvořících nejbližší vrstvu kolem jádra. Kromě správy souboru umožňovaly přistupovat k libovolné úloze. Velká část kódu byla přesunuto z jádra mimo - do „uživatelského prostoru“, tedy běžela jako jiné běžné programy. Jádro se tak výrazně zmenšilo a vysloužilo si zařazení do kategorie mikrokernel. Základní myšlenkou mikrokernelu je, že ve striktně chráněném módu jádra běží pouze minimální množství kódu. Všechno ostatní je spuštěno jako běžná úloha - démon. Na rozdíl od tradičních systému se pod Machem mohl proces může skládat z několika vláken. Tento způsob je v dnešních systémech zcela běžný, avšak Mach byl prvním systémem, který představil procesy a

vlákna tímto způsobem. Úloha jádra byla redukována pouze na základní řízení a správu běžících podprogramů a řízení jejich přístupů k hardwaru. Proces může požádat jádro o přístup k portu a pak využít IPC systém k zaslání zprávy na tento port. Existence portů a využívání IPC je možná nejzákladnějším rozdílem mezi Mach a tradičními jádry. Jádro kontroluje validitu zpráv, aby zabránilo nestabilitě jednotlivých programů.

Díky tomu, že se proces skládá z více vláken využívajících IPC mechanizmus, Mach byl schopen pozastavit a znovu spustit libovolné vlákno po dobu zpracování zprávy. To umožnilo jádru distribuci zpráv přes několik procesoru při přímém využití sdílené paměti nebo přidání kódu pro zaslání zprávy na jiný procesor. To je v tradičních jádrech velmi obtížně implementovatelné, protože systém musí mít jistotu, že se dvě různá vlákna běžící na různých procesorech nepokusí zapsat na stejné místo v paměti najednou.

Zpočátku měl IPC systém výkonnostní problémy, protože vznikalo velké množství kolizí. Mach používal sdílenou paměť pro fyzické vyměňování zpráv mezi procesy. Fyzické kopírování zpráv bylo příliš pomalé a Mach se spoléhal na MMU (memory management unit - jednotka správy paměti) pro rychlé mapování dat z jednoho procesu druhému. Data mohou být fyzicky kopírována pouze pokud jsou zapisována. Tento proces je známý pod názvem copy-on-write.

Další výhodou jádra Mach, je že pokud se vyskytne chyba v kódu jádra, nevyžádá si na rozdíl od monokernelu restart celého stroje, ale pouze ukončení problematického procesu, který může být znovu spuštěn, dokud nebylo samo o sobě kompletní. Práce začala převzetím funkčního Accent IPC/port systému a pokračovala zpracováním dalších klíčových částí OS jako byly úlohy, vlákna a virtuální paměť. Jakmile byly tyto části dokončeny, převzali programátoři některé části BSD a přepsali je přímo do Mach. V roce 1986 byl systém kompletně hotov do té fáze, že byl samostatného běhu na stroji DEC VAX. Přestože jeho užitná hodnota byla minimální, cíl vytvořit mikrokernel byl úspěšný. Následovaly další architektury IBM PC/RT a stanice založené na Sun Microsystems 68030 aby se ověřila portabilita jádra. Dále následovaly Encore Multimax a Sequent Balance stroje, aby se prokázala schopnost běžet na multiprocesorovém systému.

Mach byl původně navržen jako náhrada za klasický Unix, a proto implementoval mnoho Unixových myšlenek. Mach využívá chráněný souborový systém obdobný jako v Unixu. Protože jádro bylo privilegováno (běželo v kernel-space), bylo možné, aby mu špatně fungující nebo zlomyslné programy zaslaly příkazy, kterým by mohly poškodit systém, a z tohoto důvodu jádro vždy kontrolovalo validitu každé zprávy. Většina funkcionality byla umístěna jako samostatné procesy do user-space a tak jádro muselo umožnit přidělit těmto programům práva například pro přístup k hardwaru.

Některé nezvyklé funkce jsou také založeny na IPC mechanizmu. Například Mach je schopen běžet sám o sobě na multiprocesorovém počítači. V tradičních jádrech je potřeba vyřešit mnoho problémů, aby mohlo být znovupoužitelné a přerušitelné, tak aby programy běžící na více procesorech byly schopny volat jádro současně. Pod Mach, kde jsou části systému izolovány do serverů, které jsou schopny běžet, jako jiné programy, na libovolném procesoru.

Naneštěstí využití IPC pro většinu úloh způsobuje velmi vážné výkonnostní problémy. Mach 3 systém běžící pod Unixem byl o 50% pomalejší obdobné monolitické jádro. Studie ukázaly, že obrovské množství výkonu spotřebovalo právě řízení IPC. Toto měření bylo prováděno na systému s jedním rozsáhlým serverem poskytujícím služby operačního systému. Rozčlenění na více menších serverů by způsobilo ještě větší režii a ztrátu výkonu.

Vývojáři provedli mnoho pokusů, aby zvýšili výkon jádra Mach a jiných mikrokernelů, ale v polovině 90. let se většina původního zájmu vytratila. Koncept operačního systému založeném na IPC se zdál být mrtvým a myšlenka sama o sobě jako chybná.

Pozdější konkrétní studie výkonnostních problému odhalily množství zajímavých fakt. Jedním z nich bylo, že IPC sám o sobě není problémem, nýbrž režijní náklady spojené s mapováním paměti

pro jeho podporu. To však přidávalo pouze malé množství času potřebného pro zavolání. Zbytek času jádro strávilo dalším zpracováním zprávy. Primárně se jednalo o ověření validity a kontrolu práv. Při testech na 486DX-50 bylo standardní Unixové systémové volání o 80% rychlejší než obdobná operace v Mach. Pouze malá část byla spojena s hardwarem a zbytek Mach strávil zpracováním zprávy.

Když byl Mach 2.x poprvé vážně využit, výkon byl o cca 25% pomalejší než tradiční jádro. Tato cena již nebyla považována za příliš znepokojivou, protože Mach nabízel také multiprocesorovou podporu a byl velmi snadno portovatelný. Faktem je, že systém skrýval vážný výkonnostní problém, který začal být více zřetelný, když se začal používat Mach 3 a vývojáři se pokoušeli vytvořit systém v user-space.

Když se Mach 3 pokusil přesunout celý operační systém do user-space, začala být režie najednou ohromující. Uvažujme jednoduchou úlohu dotazující se systému na aktuální čas. Pod pravým user-space systémem by měl být server zpracovávající tento požadavek. Proces, který požaduje aktuální čas, spustí IPC systém a způsobí přepnutí kontextu a mapování paměti. Jádro by mělo prozkoumat obsah zprávy, aby zjistilo práva odesílatele a provedlo další mapování do paměti serveru a další přepnutí kontextu, pro dokončení úlohy. Tento proces se opakuje při vracení výsledku a vyžaduje si celkové 4 přepnutí kontextu a mapování paměti, stejně jako provedení dvou úloh pro zjištění práv a validity zpráv.

Převedeno do řeči čísel volání této úlohy pod BSD jádrem na 486DX-60 zabere cca 20 ms. Provedení stejné úlohy na Mach 3 si vyžádá 114 ms. V detailních testech publikovaných v roce 1991 Chen a Bershad zjistili, že režie systému degraduje výkon až o 66% v porovnání s tradičními jádry.

To však není jediný zdroj výkonnostních problémů. Dalším zdrojem jsou pokusy o zabrání paměti, když dochází fyzická paměť a musí se swapovat nepotřebné stránky. V tradičních monokernelech autoři měli přímé zkušenosti, které části jádra se navzájem volají, a to jim umožnilo správně odladit jejich stránkovací systém tak, aby neodkládal stránky, které jsou používány. To pod Mach není možné, protože jádro nemá žádné ponětí o tom, z čeho se skládá operační systém. Muselo být proto použito řešení, které vyhoví všem, ale zase způsobí výkonnostní problémy. Mach 3 se pokoušel přímo tento problém řešit pomocí jednoduchého stránkovače spoléhajícího na user-space stránkovače pro lepší specializaci. To mělo však na výkon pouze nepatrný efekt. V praxi byly všechny výhody vymazány drahým IPC potřebným pro volání stránkovačů.

Některé z těchto problémů budou i v jiných systémech pracujících na multiprocesorových strojích a v polovině 80. let se zdálo, že v budoucnu budou na trhnu převážně takovéto počítače. Realita však byla jiná, než se očekávalo, a multiprocesorové počítače byly používány pouze po velmi krátký čas v serverech na počátku 90. let, pak se však vytratily. Mezitím procesorům rostl výkon každým rokem o cca 60%, naproti tomu však rychlost pamětí rostla v průměru jen o 7% za rok, čili jakýkoliv přístup do paměti se stával dražším, a protože Mach byl založen na mapování paměti mezi programy, jakýkoliv cache miss (nenalezení dat v cache paměti) vytvořil velmi pomalé IPC volání.

Bez ohledu na všechny výhody přístupu Mach, nebyly tyto výkonnostní problémy v reálném světě prostě akceptovatelné. Protože všechny týmy došly ke stejným výsledkům, počáteční entuziazmus se rychle vytratil. Po krátkém čase došla většina vývojářské komunity k závěru, že koncept využívající IPC jako základ operačního systému je špatný.

Z předchozí kapitoly je patrné že IPC si vyžaduje obrovskou režii a je základním problémem Mach 3 systému. Ačkoliv je koncept multiserverového systému velmi slibný, vyžadoval si ještě dlouhý výzkum. Důležité bylo izolovat související kód do modulů tak, aby nedocházelo k volání mezi servery. Například většina kódu pro práci se sítí může být umístěna do samostatného serveru a minimalizovat tak IPC pro normální síťové úlohy. V Unixu to není příliš jednoduché, protože systém je založený na využití souborového systému jako základu pro všechno od bezpečnosti po

síťové služby.

Většina vývojářů se raději zabývala originálním konceptem jednoho velkého serveru poskytující kompletní funkcionalitu operačního systému. Pro zjednodušení vývoje však umožňovali operačnímu systému běžet jak v kernel-space tak user-space. Vyvíjeli nové části v user-space a měli k dispozici všechny výhody systému Mach a po odladění je přesunuli z výkonnostních důvodů do kernel-space. Na této metodě bylo představeno několik operačních systému: MkLinux, OSF/1 a hlavně NeXTSTEP/OPENSTEP/Mac OS X.

Mach 4 se pokusil výše zmíněné problémy vyřešit více radikálním způsobem. Především bylo zjištěno, že většina programového kódu většinou nezapisuje a potenciální copy-on-write jsou velmi řídké. Proto vznikla myšlenka nemapovat paměť mezi dvěma programy pro IPC, ale místo toho migrovat programový kód, který se bude používat, do lokálního prostoru programu. To vedlo ke konceptu „shuttles“ (kyvadlová doprava) a došlo ke zvýšení výkonu.

Při všech testech byl výkon IPC označen za hlavní zdroj problémů, a přičítala se mu ztráta 73% cyklů. V polovině 90. let byly práce na mikrokernelových systémech prakticky ukončeny. I když trh věřil, že všechny operační systémy budou založeny v 90letech na mikrokernelech, jediným dnešním rozšířeným desktopem je právě Mac OS X, který využívá co-located serverů běžících na velmi upraveném jádře Mach 3.

Novým místem pro využití systému Mach by mohly být jednouživatelské operační systémy v mobilních telefonech, protože mikrokernel umožní využít jen ty části systému, které jsou opravdu potřeba.

XNU

Tato kapitola čerpá z [strh].

Jádro XNU je jádro použité v Mac OS X a bylo zveřejněno jako open source v operačním systému Darwin. Jedná se o hybridní kernel kombinující Mach 3 jádro s komponentami z jádra Free BSD 5.x a C++ API rozhraním pro psaní ovladačů zvaný I/O Kit. XNU je akronym pro X is Not Unix.

XNU je hybridní kernel, který kombinuje výhody monolitického kernelu a mikrokernelu, a pokouší se využít nejlepší z obou technologií, jako je například schopnost zasílaní zpráv mikrojádra umožňující těžit velkým částem operačního systému z chráněné paměti, stejně jako rychlost monolitických jader pro některé kritické úlohy.

XNU přebírá schopnosti poskytované jádrem Mach. Mezi ně patří správa procesů, vláken, preemptivní multitasking, zasílaní zpráv, chráněná paměť, správa virtuální paměti, jednoduchá podpora práce v reálném čase, podpora debuggingu jádra a konzolové vstupně/výstupní (I/O) operace. Další výhodou převzatou z Mach je možnost hostovat binární kód pro různé CPU architektury v jediném souboru (jako například pro PowerPC a x86). Díky tomu je přechod z architektury PowerPC na x86, který teď u Apple probíhá, pro uživatele prakticky nepostřehnutelný. Většina softwaru nyní vychází v podobně balíčků označených jako „Universal“, které lze spustit na obou procesorech.

Z BSD si XNU přineslo části jádra poskytující POSIX API (systémová volání BSD), Unixový model procesů nad úlohami Mach, základní bezpečnostní politiku, správu id uživatelů a skupin, práva, síťovou vrstvu, virtuální souborový systém (obsahující žurnálovací vrstvu nezávislou na souborovém systému), kryptografický framework, IPC Systému V a další.

Další součástí je framework ovladačů zařízení I/O Kit, který je napsán podmnožinou jazyka C++. Díky objektově-orientovanému návrhu mohou být ovladače psány mnohem rychleji za použití méně kódu. Vlastnosti, které jsou společné libovolné třídě ovladačů, jsou poskytovány přímo frameworkem. I/O Kit je vícevláknový systém, který je bezpečný pro symetricky multiprocesorové systémy, a umožňuje připojení zařízení za běhu a automatickou a dynamickou konfiguraci zařízení.

Mnoho ovladačů může být psáno pro běh v user-space a tím zajistit vyšší stabilitu systému. Pokud havaruje ovladač v user-space, nezpůsobí to havárii samotného kernelu. Ovladač totiž běží jako samostatný proces, který se ukončí a může se spustit znovu.

Současná verze XNU je schopna běžet na procesorech x86 (Intel a AMD) stejně jako na procesorech PowerPC. Umí využít jak jednoduchých procesorů, tak i symetricky multiprocesorových modelů.

Rhapsody

Rhapsody [2] bylo kódové označení vyvíjeného operačního systému nové generace Apple Computer v době mezi zakoupením NeXT koncem roku 1996 a oznámením vydání Mac OS X v roce 1998. Rhapsody byl poprvé prezentován v roce 1997 na světové konferenci vývojářů (WWDC - Worldwide Developers Conference). Objevily se dvě vydání pro vývojáře na strojích Intel x86 a PowerPC procesorech. Vydání plné verze bylo plánováno na jaro 1998. Na výstave MacWorld Expo v New Yorku oznámil Steve Jobs, že Rhapsody bude vydána jako Mac OS X Server. Opravdu vydán byl v roce 1999 jako Mac OS X Sever 1.0 a jeho zdrojové kódy byly vydány v samostatné větvi zvané Darwin, která představuje open source nejnižšího základu Mac OS X.

Základní myšlenkou operačního systému byl mikrokernel Mach, BSD vrstva operačního systému, Yellow Box (později nazván Cocoa) objektově orientované frameworky převzaté z NeXTSTEPu a kompatibilní prostředí s kódovým označením Blue Box, které umožňovalo běh aplikací napsaných pro Mac OS 9.

Darwin

Darwin je open source operační systém podobný Unixu, který je zdarma, a byl poprvé vydán Apple Computer v roce 2000. Obsahuje množinu základních komponent, na kterých byl vyvinut Mac OS X. V dubnu 2002 založil Apple a ISC (Internet Software Consorcium) komunitu nazvanou OpenDarwin, aby koordinovala vývoj Darwinu.

OpenDarwin byla samostatná oficiální vývojová skupina pracující mimo firmu Apple, z části složená právě z jejích zaměstnanců. Vyvíjela Darwin jak pro Intel tak pro PowerPC. Touto skupinou bylo vyvinuto mnoho vlastností, které obsahuje dnešní Mac OS X.

Darwin používá kombinaci jádra Mach nabízejícího preemptivní multitasking, meziprocesovou komunikaci, správu přerušení a další. Jádro přináší také vynikající balík ovladačů zařízení zvaný IOKits (Input-Output kits). Základní rozdíl mezi mikrokernelem použitým v Darwinu a Mac OS X, a monolitickým kernelem, jaký je například použit Linuxu, je ten, že monolitické jádro je jeden kus kódu, zatím co mikrokernel se skládá z několika malých procesů, které běží nezávisle na sobě. To může činit mikrokernel složitějším a snižovat jeho výkonnost. Naproti tomu poskytuje velkou výhodu, že je možno přidat praktický jakýkoliv nový proces a jádro tak v podstatě libovolně rozšířit.

Současná monolitická jádra nabízí také možnosti dynamického přidávání modulů, to však může vést k nestabilitě systému, pokud není modul zcela bezpečný. Také psaní a testování dynamických modulů bývá náročnější, než je tomu v případě mikrokernelu. V něm se části vyvíjí samostatně jako nezávislý program.

Nad mikrokernelem Mach je postavena aplikační vrstva BSD převzatá z Free BSD 5. Tato vrstva zajišťuje kompatibilitu s většinou softwaru napsaného pro UNIX. Kromě toho poskytuje také vynikající síťové služby.

Minulost Darwinu byla velmi světlá a stejně slibná je i budoucnost. Darwin se používá na mnoha univerzitách pro výuku operačních systémů. Zdá se, že je blízko doba, aby byl Darwin scho-

[2] Název byl převzat podle názvu skladby George Gershwina Rhapsody in Blue

pen samostatného života bez Mac OS X. Objevují se obavy, aby se neopakovala situace podobná System V Unix, kdy se v průběhu času vytvořilo velmi mnoho fragmentů. Situace je malinko jiná, protože Darwin je kontrolován a usměrňován firmou Apple, a každý má přístup ke stejným zdrojovým kódům, na rozdíl od Unixu, kde vznikly frakce jako Solaris nebo AIX, které jsou sice založeny na stejném operačním systému, ale jsou mezi nimi značné rozdíly v návrhu. Darwin OS si získává na popularitě a zájmu vývojářů a má tendence konkurovat Linux, rodině BSD a dokonce i Mac OS X. To je však předpověď na mnoho let dopředu a bude jistě zajímavé sledovat, jak se Darwin uchytí na počítačovém trhu.

V červenci 2003 Apple vydal Darwin ve verzi 2.0 pod APSL licencí, kterou Free Software Foundation (FSF) potvrdilo jako licenci volného software. Předchozí vydání patřili pod dřívější verze APSL, které nevyhovovaly FSF definicím o volném software, i když byly splněny požadavky Open Source definice.

2.2.5 Mac OS X

Popis systému

Tato kapitola čerpá z [strc].

Mac OS X se od předchozích Macintosh operačních systému radikálně liší a nemá s nimi prakticky téměř nic společného. Tento operační systém je proti předchozím systémům mnohem stabilnější a spolehlivější. To zajišťuje především pre-emptivní multitasking a ochrana paměti, které také umožňují běh více aplikaci najednou bez možnosti aby jedna při svém pádu ohrozila nebo poškodila aplikaci jinou, tak jako se to mohlo stát u předchozích systémů. Jádro Mac OS X je založeno na dříve existujícím jádře OPENSTEP, které je navrženo jako přenositelné a umožňuje jednoduché přechody od jedné platformy k jiné. Například NeXTSTEP byl portován z originálního 68k NeXT workstation na PA-RISC/SPARC/x86 orientované stroje ještě před tím, než NeXT koupil Apple a OPENSTEP byl postupně portován pro architekturu PowerPC jako součást projektu Rhapsody.

Nejvíce viditelnou změnou pro uživatele je grafické rozhraní Aqua. Používá vyhlazování hran, průhlednosti, více barev a textur pro okna a ovládacích prvků na Desktopu než nabízel vzhled OS 9 Platinum. Přesto nový vzhled Aqua vyvolal mezi některými uživateli Macintosh rozhořčení, neboť jim připadalo rozhraní až příliš hezké a „přeslazené“, které by nemělo mít místo v profesionálním systému. Ostatní chválili Apple za další inovativní revoluci a nový konzistentní vzhled pro takzvaný Human Interface [3].

Základ Mac OS X je open source Unix-like operační systém, postavený okolo jádra Mach se standardními Unix programy přístupnými z příkazové řádky. Krátce před tím než Apple vydal svůj první Mac OS X, uvolnil tento základ jako Darwin. Na tomto základu navrženém a vyvinutém Applem jako řada proprietárních a uzavřených (close source) komponent obsahujících Aqua user interface a Finder shell.

Ke každému prodanému Mac OS X přikládá Apple zdarma vlastní vývojářské nástroje, které nabízí velmi profesionální a kvalitní prostředí pro vývoj aplikací, nazvané Xcode. Ten vychází z původních vývojářských nástrojů ze systému NeXTSTEP. V současné době se nachází ve verzi 2.4.1 a aktualizace jsou zdarma ke stažení na webových stránkách Applu. Pro novou verzi Mac OS X 10.5 nazvanou Leopard bude k dispozici nová verze Xcode 3.0 s řadou vylepšení, která usnadní vývoj. Xcode nabízí prostředí pro vývoj a překlad programů v několika programovacích jazycích jako jsou C, C++, Objective-C a Java. Pozdější verze podporují také kompilaci pro různé procesory, umožňující sestavit aplikaci například buď pouze pro PowerPC nebo x86, ale také pro oba procesory současně tzv. Universal Binary.

[3] Dokument s doporučeními pro vzhled aplikací běžícím v Mac OS X, který je dostupný všem vývojářům.

Vývoj v jednotlivých verzích

Mac OS X Public Beta byla ranou beta verzí operačního systému Mac OS X firmy Apple Computer vydanou 13. září 2000 za cenu 29.95 USD. Umožnila softwarovým vývojářům a lidem se zájmem o novinky se seznámit s přicházejícím operačním systémem a umožnila jim vývoj nového softwaru před jeho finálním vydání. Public Beta bylo první možností, kdy si lidé mohli vyzkoušet nové Aqua rozhraní. Systémové ikony byly větší a detailnější a nový vzhled systému působil velmi příjemné na pohled. Kromě přidání vzhledného rozhraní, přinesla Public Beta také open source Darwin jádro, chráněnou paměť a prostor pro množství dalších technických vymožeností. Vzhledem k tomu, že se jednalo o veřejnou betaverzi, nebyla prosta chyb a neočekávaného chování. Apple využil zpětnou vazbu uživatelů ke kvalitnímu vylepšení prodejní verze. Navzdory všem chybám a časovému omezení licence (do jara 2001) byla Mac OS X Public Beta přijata komunitou velmi dobře a našlo se mnoho nadšenců, kteří testovali nový systém. Vlastnící této verze dostali slevu $29.95 na plnou verzi Mac OS X 10.0.

Mac OS X 10.0 byla představena dne 24. března 2006. Její interní označení bylo Cheetah neboli gepard. Tato první verze byla pomalá, nedokončená a měla velmi málo aplikací, které byly většinou od nezávislých vývojářů. Ačkoliv kritikové označili za chybu, že byl takový nepřipravený systém vydán veřejnosti, museli časem uznat, že to byl velmi dobrý tah, neboť mohl být využitý jako základ pro další vylepšení. Vydání Mac OS X bylo přijato Macintosh komunitou jako velký úspěch, který zastínil nezdary a neustálé odkládání sytému v roce 1996. Následovalo několik minoritních oprav chyb, kernel panic [4] se přestal objevovat méně častěji a systém začal sklízet chválu pro svou stabilitu již na počátku svého vývoje. I přesto byl stále kritizovaný za svou pomalost a neznatelné zlepšení od poslední verze.

Mac OS X 10.1 byl vydán v 21. září 2001 a nesl vnitřní označení Puma. Tato verze výrazně zlepšovala výkon systému a přinášela chybějící vlastnosti, jako například přehrávání DVD. Apple vydal verzi 10.0 jako aktualizační CD zdarma nebo jako krabicovou verzi za poplatek 129 USD pro uživatele Mac OS 9. V lednu 2002 Apple oznámil, že Mac OS X bude od konce měsíce výchozím operačním systémem pro všechny Macintosh produkty.

Mac OS X 10.2 Jaguar byla vydána 24. srpna 2002 a nesla ve svém názvu slovo „Jaguar“, které se stalo součásti značky [5]. Tato verze přinesla další důrazné zvýšení výkonu, nové uhlazenější vzhled a mnoho dalších vylepšení (Apple uváděl číslo přes 150). Tato vylepšení obsahovala mimo jiné:

- zlepšená podpora pro MS Windows sítě
- Quartz Extreme pro vykreslování grafiky přímo na grafické kartě
- adaptivní spam filtr pro aplikaci Mail, která byla založena na sémantickém indexování
- systémově dostupný framework poskytující informace z adresáře - aplikace Address Book
- Rendezvous sítě (Apple implementace Zeroconf [6]), ve verzi 10.4 přejmenované na Bonjour

[4] Zpráva zobrazená operačním systémem na základě detekování vnitřní chyby systému, ze které se nelze zotavit. Tento výraz je znám především v Unixových systémech.

[5] Toto označení nebylo nikdy použito ve Velké Británii na základě dohody s automobilkou Jaguar.

[6] Technika umožňující automatické sestavování IP sítí bez konfigurace nebo speciálních serverů. Umožňuje připojování vzdálených tiskáren, FTP serverů, bez znalosti toho, kde se na síti nachází. První počátky Zeroconf pochází právě z Apple Computer

- vylepšený Finder, který umožňoval mimo jiné vyhledávání v každém okně
- nové Apple Universal Access, které zpřístupňují počítač postiženým lidem

Verze 10.2 byla označována jako první kvalitní vydání Mac OS X. Díky výrazným změnám v API od předchozí verze, je tato verze většinou označována jako minimální požadavek pro běh většiny software. Právě nejvýraznější změnou je přesunutí záležitosti vykreslování oken na grafickou kartu, která v současné době poskytuje obrovský potenciál, který je při běžné práci zcela nevyužitý. V předchozích verzích se o správu oken staral Quartz Compositor, který byl odpovědný za uživatelské rozhraní. Jeho dvě základní funkce jsou správa a vykreslování oken a zpracování událostí. Každé okno v Mac OS X je uloženo jako bitmapa s údaji o pozici (včetně z-indexu), průhlednosti a anti-aliasingu. Aplikace, která je vlastníkem takového okna kreslí přímo do této bitmapy využitím některého z kreslících módů (jako jsou Quartz 2D, QuickDraw nebo OpenGL). Quartz Compositor využíval tyto bitmapy k tomu aby z nich sestavil okna tak, jak mají být zobrazena na monitoru. To umožňuje odstínit správce oken a vykreslovací model a poskytuje tak možnosti k vytváření zajímavých efektů, jako například *genie effect* při minimalizaci do Docku. Ve své roli správce oken Quartz Compositor také zpracovává události vstupu jako jsou stisknutí klávesy a kliknutí myši. Quartz Compositor načítá tyto události z fronty, zjišťuje kterému procesu okno patří a potom předá tuto událost správnému procesu. Quartz Extreme, který se objevil právě ve verzi 10.2 je rozšířením Quartz Compositoru, který využívá OpenGL k vykreslování na display rychleji jako textury v 3D OpenGL kontextu. To umožňuje mnohem rychlejší vykreslování díky 3D akceleraci a využití potenciálu, který skýtají současné grafické karty. Díky tomu musí mít všechny počítače Macintosh grafickou kartu s podporou vykreslování textur a minimálně 16MB VRAM. To znamená, že minimální grafickou kartou je NVIDIA GeForce 2 MX nebo ATI Radeon v AGP nebo PCI Express slotu. S příchodem Intel Macs, které využívají Intel GMA950 grafický procesor, jež je použit v Mac mini, byla přidána podpora také tohoto procesoru. Díky tomu, že o zobrazení okna včetně průhlednosti a anti-aliasingu se stará grafická karta, je celé rozhraní velmi rychlé a umožňuje například přehrávání videa za poloprůhledným oknem bez zatížení procesoru.

Mac OS X 10.3 Panther byl představen 24. října 2003. Kromě dalšího vylepšení výkonu, byl proveden také zatím největší zásah do uživatelského rozhraní. Aktualizace přinesla ještě více vylepšení než předchozí verze. Na druhou stranu bylo upuštěno od podpory počítačů založených na raných procesorech G3. Mezi tato vylepšení patří:

- vylepšený Finder v novém vzhledu, nastavitelný Sidebar a rychlé vyhledávání
- Expose - nový systém pro manipulaci s okny
- rychlé přepínání uživatelů, které umožnilo aby zůstal přihlášený stávající uživatel
- iChat AV, který přidal video konference
- vylepšené renderování PDF, které umožnilo rychlejší prohlíženi PDF souborů
- vylepšená interoperabilita s MS Windows
- FileVault: zašifrování domovské složky za běhu pomocí AES algoritmu
- vylepšená podpora PowerPC G5 procesorů
- webový prohlížeč Safari

Mac OS X 10.4 Tiger přišla na svět 29. dubna 2005. Apple představil přes 200 nových vlastností a vylepšení. Stejně jako u Pantera bylo upuštěno od podpory některých starších strojů. Tiger

vyžadoval počítač s vestavěným FireWire a DVD mechanikou. Mezi nejvýznamnější vylepšení patří:

- Spotlight - vyhledávání založené na meta informacích o souborech a indexované databázi podporovaných souborů
- Dashboard - virtuální plocha obsahující malé aplikace nazvané *Widgets*, která je přístupna po stisknutí klávesy.
- Smart Folders (chytré složky), které ukládají vyhledávací parametry Spotlightu a umožňují se tak kdykoliv vracet k dřívějšímu vyhledávání s aktuálními výsledky
- aktualizovaný program Mail, který obsahoval Smart Mailboxes (chytré schránky), které umožňují vyhledávání pomocí spotlightu
- nová verze programu iChat, která podporuje H.264 kompresi a umožňuje kvalitnější videopřenosy, a podpora Jabber protokolu
- QuickTime 7 přinesl také podporu H.264 komprese a zcela přepracované rozhraní
- aplikace umožňující vytváření skriptů pomocí grafického rozhraní nazvaná Automator
- VoiceOver pro čtení obsahu obrazovky pro zrakově postižené
- Core Image a Core Video pro nové efekty pro práci s obrázky a videem v reálném čase
- 64bitová podpora pro procesory G5 pro programy nebo programové části bez uživatelského rozhraní, které muselo být stále 32bitové
- aktualizované Unixové programy jako cp, mv, rsync které umí pracovat s HFS Plus metadaty
- rozšířená práva systému využívající ACL [7]
- zcela nové aplikační rozhraní nazvané Core Data, které výrazným způsobem vylepšilo správu aplikačních dat v Cocoa programech Intel x86 verze Mac OS X Tiger unikla do Internetu, načež Apple oznámil přechod na Intel platformu. To bylo oficiálně oznámeno v červnu na Worldwide Developers Conference, kde se ukázalo, že Mac OS X byl interně kompilován pro Intel platformu v každé verzi, aby byla udržena parita mezi Intel a PowerPC verzí. Vývojáři si mohli zakoupit počítač s procesorem Intel Pentium 4 s předinstalovaným systémem 10.4.1 - 10.4.3 (jak postupně vycházely aktualizace), aby mohli začít vyvíjet pro platformu Intel. První verzí, která byla pro Intel oficiálně představena byla 10.4.4. Všechny nové Intel Mac počítače mají předinstalovaný systém Tiger. Ve verzi 10.4.7 byla přidána podpora 64bitových procesorů Intel Xeon a Intel Core Duo. Od doby vydání verze pro Intel se na Internetu objevila komunita OSx86 snažící se provozovat Mac OS X na jiném hardware než od Apple. S každou aktualizací byly vydány záplaty, které více znemožňoval běh operačního systému na jiném než Apple hardwaru.

Mac OS X 10.5 Leopard byl oznámen na Worldwide Developers Conference 6. července 2005 a byla poprvé představena vývojářům na Worldwide Developers Conference 7. srpna 2006. Steve Jobs oznámil, že OS X Leopard bude k dispozici na jaře 2007 a bude podporovat jak PowerPC tak Intel x86 Macintosh počítače. Bylo představeno několik nových a zajímavých technologií, ale stejně tak bylo řečeno, že tyto jsou předmětem změn a proto se o nich nebudu šířeji rozepisovat.

[7] Access control lists

2.3 Finder

2.3.1 Co je Finder?

Tato kapitola čerpá z [stre]

Finder je jedna ze základních komponent Mac OS a provází operační systémy Apple již od jejich raných počátků. Jedná se o základní aplikaci používanou pro správu souborů, pevných i výměnných disků, síťových disků a spouštění jiných aplikací. Byl poprvé představen na jednom z prvních počítačů Macintosh a byl součástí operačního systému GS/OS na Apple IIgs. S přechodem na unixový Mac OS X aplikace podstoupila kompletní přepsání.

Finder je první aplikace, se kterou přijde uživatel po zapnutí počítače do styku, protože je zodpovědný za vykreslování pracovní plochy. Pracovní plocha je složka na disku jako každá jiná, ale její obsah je vždy zobrazen na pozadí.

Stejně tak zajišťuje zobrazení koše, kam jsou přemisťovány soubory při mazání. Ikona koše je od verze OS X umístěná a přístupná v Dock (také samostatná aplikace zobrazující spuštěné programy). V předchozích verzích byla umístěna přímo na ploše.

2.3.2 Finder 1.0 - 4.1

Původní Finder pracoval s MFS (Macintosh File System), který vždy obsahoval prázdnou složku v kořenovém adresáři na každém disku. Při každém přejmenování a použití této složky byla vytvořena nová. Složky nebylo umisťovat do jiných složek a musely být spravovány pouze pomocí aplikace Finder a fyzicky na disku neexistovaly. To mělo za následek, že na jednom disku nemohly být umístěny dva soubory se stejným názvem. V otevíracím dialogu také chyběl výpis složek a byl zobrazen vždy jen seznam všech souborů.

Finder poskytoval složku odpadkového koše. Jedinou možností jak vymazat soubory bylo přesunout tyto soubory do složky odpadkového koše a tu pak vyprázdnit. Ve skutečnosti byla tato složka virtuální, nebyla na disku fyzicky přítomná a seznam souborů pro smazání byl uložen v paměti. Finder tak vždy před svým ukončením složku vyprázdnil a tím fyzicky vymazal soubory. Při havárii systému se soubory vrátily na své původní místo nesmazané.

Původní Finder frustroval uživatele svou velmi malou rychlostí kopírování souborů, protože ty byly mezi kopírováním odkládány na disk, neboť tehdejší počítače měly velmi malou paměť.

Apple se snažil tento problém řešit, ale až Finder 4.1 vydaný v dubnu 1985 opravdu vylepšil rychlost a přidal řadu funkcí. Mezi ně patřil například příkaz „New Folder“ (Nová složka) nebo „Shut down“ (Vypnout) ve speciálním menu, které také zpřístupňovalo „MiniFinder“. MiniFinder měl zjednodušené rozhraní a obsahoval často používané aplikace a dokumenty, které spouštěl mnohem rychleji.

2.3.3 Finder 5.x

Apple nahradil v září 1985 původní MFS novým souborovým systémem HFS (Hierarchical File System) jako součást nového Finder 5. 0, který byl představen spolu s prvním pevným diskem použitým v počítači Mac. Vnořené složky již přestaly být iluzí vytvářenou Finderem a zobrazovaly skutečné rozmístění souborů na disku. Finder 5.0 také přinesl příkaz „Eject“ (Vysunout) a drobné změny vzhledu systémových ikon.

2.3.4 Finder Software 6.x

Původní verze aplikace Finder byla ukončena vždy, když uživatel spustil jinou aplikaci, protože dřívější operační systémy Apple byl jednoúlohové. System 5.x přišel s verzí Finder 6.0 a novým MultiFinder, který umožňoval kooperativní multitasking. MultiFinder byl aktivován v nastavení a projevil se po restartování systému. Operační System Software 6.0 přišel s vylepšeným Finder 6.1 a nabízel také inovovanou verzi MultiFinder.

Mac OS Finder nabízel funkci „Universal Desktop“, která zobrazovala sjednocení všech neviditelných složek „Desktop Folder“ v kořenovém adresáři každého připojeného disku. To znamenalo, že pokud byl soubor přesunut z disku na plochu, nebyl vždy fyzicky překopírován na správný disk a při odpojen tyto soubory zase z plochy zmizely. Odpojení disku se provádělo přetažením ikony disku na koš.

2.3.5 Finder 7.0 - 9.2

V roce 1991 vydal Apple výrazně přepsaný operační systém System 7. Tak jako většina částí nového OS se i Finder dočkal výrazných renovací. MultiFinder přestal být volitelnou součástí a byl napevno zabudován. Celý operační systém získal barevné rozhraní a pozadu nezůstal ani Finder. Jeho novou vlastností bylo zobrazovat soubory v seznamech, kde každá složka získala tlačítko pro rozbalení nového podseznamu se svým obsahem, a umožnit tak pracovat s více složkami bez otevření více oken.

Vylepšena byla také funkce „Labels“ (Štítky) známá ze System Software 6 umožňující uživatelům definovat si několik typů souborů a byla rozšířena o možnost definovat si vlastní názvy a barvy. Označené soubory měly obarvené ikonky a štítky byly využitelné při hledání. Složka odpadkového koše již byla skutečnou složkou a soubory v ní zůstávaly i po restartu.

Finder 7.0 také přinesl možnost vytváření zástupců, které ukazovaly přímo na soubor (nikoliv na jeho cestu) a zůstávaly platné i při přesunutí nebo přejmenování originálního souboru.

Přestože v průběhu času systémy Macintosh procházely mnoha výraznými úpravami, zůstal Finder prakticky v nezměněné podobě až do vydání Mac OS 8 v roce 1997. Finder 8.0 byl první verzí, která byla plně vícevláknová. Poprvé v historii nezablokovalo kopírování souborů nebo vyprazdňování složky odpadkového koše celý Finder dalšímu používání.

Finder se, stejně jako zbytek operačního systému, přizpůsobil novému vzhledu Platinum. Přinesl také několik dalších funkcí jako vyskakovací okna, která se zobrazovala vespod obrazovky a jejich obsah musel uživatel zobrazit až kliknutím na ně. Také byly představeny automaticky se otevírající složky „Spring-loaded Folders“, které umožnili přesouvání pomocí drag & drop do nižších hierarchických vrstev. Pokud se při přesouvání zastavila myš nad složkou, došlo k jejímu otevřeni a zpřístupnění obsahu.

Na počátku roku 1998 byl vydán Finder 8.1, který představil podporu vylepšeného souborového systému HSF+ a byl posledním výrazným vylepšením klasického Mac OS Finderu.

2.3.6 Finder 10.0 - 10.2.1

Mac OS X Finder nebyl aktualizací předchozích verzí, ale byl kompletně přepsán s využitím konceptu souborového manažera z NeXTSTEP. Tím se velmi vzdálil původnímu Finderu a byl velmi špatně přijat mnoha dlouholetými uživateli Macintosh, protože ve svých prvních verzích nedosahoval kvalit Finderu z Mac OS 9. Apple vychvaloval nové principy: Finder přestal být celým GUI, ale pouze jednou z aplikací v systému, i když zůstal základním programem, který stále slouží pro práci s plochou. Mac OS X Finder je Carbon aplikace napsaná nad frameworkem Powerplant firmy Metrowerks. Apple si vybral Carbon namísto elegantnějšího Cocoa API, aby prokázal jeho validitu.

Označení čísla verze Finderu se neshoduje s číslem verze operačního systému. Například ve verzi Mac OS X 10.4.8 je Finder 10.4.6.

Finder 10.0 postrádal mnoho funkcí svých Classic předchůdců. Univerzální Desktop byl odstraněn a nahrazen Desktopem (pracovní plochou) reprezentovanou jednou složkou Desktop pro každého uživatele. Pryč byly i štítky (Labels) a většina souborových metadat stejně jako vyskakovací okna nebo automaticky otvírající se složky. Odstraněn byl i koš, který přestal být součástí Finderu a byl přesunut do aplikace Dock.

Finder 10.0 představil uživatelsky přizpůsobitelný panel nástrojů, jež mohl být zobrazen nad každým oknem Finderu, a také převzal z NeXT způsob zobrazení souborů souborů po sloupcích. Soubory se zobrazují ve sloupcích hierarchicky zleva – doprava a vybráním složky se přidá další sloupec s jejím obsahem. Uživatelé si také mohli vybrat, které připojené disky se zobrazí na ploše.

Mac OS X 10.1, který byl uživatelům 10.0 k dispozici jako aktualizace zdarma, také přinesl možnost vypalováni CD. Tato vlastnost byla původně přidána již v Mac OS 9.1.

Finder 10.2 již znovu disponoval automaticky otevírajícími se složkami (spring - loaded folder)[8], i když stále chyběly některé funkce jeho předchůdce z Mac 8.0. Tato verze byla schopna procházet a stahovat soubory z FTP bez možnosti uploadovat soubor zpět na server. To je způsobeno tím, že Finder využívá nástroje `mount_ftp`, který zajistí transparentní přístup na server, ale neřeší otevírání souborů s možností jejich zápisu při změně. Je to způsobeno povahou FTP protokolu, nedovoluje otevírat soubory. Slouží pouze k přenosu.

Tak jako Finder 1.0 zobrazoval Mac OS X Finder některé věci virtuálně. Například v Unixovém shellu byly názvy souborů zobrazovány POSIXovým stylem, přestože souborový systém byl ve skutečnosti HFS. Unixové soubory nesmí obsahovat znak „/“ zatímco uživatelé Macintoshe byli z historických důvodů schopni tento znak (na rozdíl od „:“) používat. Finder jednoduše tyto dva znaky vzájemně prohazoval. Jediným zakázaným znakem je dvojtečka. Kromě toho nenechal Finder zadat některé další znaky (jako například zalomení řádku), přestože je souborový systém podporoval. Finder i shell podporovaly názvy souborů v celém rozsahu Unicode.

2.3.7 Finder 10.3

Mac OS X 10.3 představila aktualizovanou verzi Finderu, která obsahovala několik klasických vlastností, s mírně upraveným vzhledem GUI.

Finder 10.3 získal „brushed-metal“ vzhled obdobný aplikaci iTunes. Stejně jako předchozí verze v Mac OS X umožňoval využití panelu nástrojů, ve kterém přibylo editační políčko pro rychlé vyhledávání souborů. Nově byl implementován panel nazvaný „Sidebar“, který byl schopen přijmout libovolný objekt (soubor, složku, aplikaci ...) a velmi snadno je zpřístupnit. Velmi důležitou vlastností tohoto vylepšení je, že se tento panel zobrazoval ve všech dialogových oknech pro otevření nebo uložení vytvořených ostatními aplikacemi. Sidebar také umožnil rychlé odebrání všech připojených zařízení. Štítky a schopnost podle nich hledat, která k nespokojenosti uživatelů stále chyběla, byly v této verzi opět obnoveny.

Finder také získal speciální mód zpřístupnitelný malým tlačítkem v horní liště (to u většiny aplikací skrývá/zobrazuje panel nástrojů). V tomto módu je skryt panel nástrojů, ale také Sidebar a je zobrazeno jenom jednoduché okno s obsahem složky. V tomto módu se pro každý obsah složky otevře nové okno. Obsah jedné složky může být zobrazen pouze v jednom okně.

[8] Pokud se při tažení souboru zastaví myš nad složkou, ta se po nastaveném čase otevře.

2.3.8 Finder 10.4

Prozatím poslední verze Mac OS X 10.4 představila další vylepšení aplikace Finder, jakým je například funkce „Slideshow“. Ta umožňuje zobrazit nalezené obrázky přímo z Finderu. Lze tak například ukázat velmi rychle fotky z dovolené, protože leze vyhledat všechny obrázky ze zadaného rozmezí a pak je jedním kliknutím spustit. Další novinkou je tedy integrování vyhledávání pomocí Spotlight, které kompletně nahradilo původní způsob hledání. Hledání je velmi rychlé, protože využívá indexované databáze souborů. Hledat lze pomocí mnoha kritérií (typ souboru, datum vytvoření nebo posledního prohlédnutí, štítků ...). Výsledky hledání lze umístit jako „Smart Folders“, které zobrazují aktualizované hledání podle zadaných parametrů. Tyto složky jsou jen XML soubor, který obsahuje definovaná kritéria hledání a při otevření tohoto souboru, se spustí hledání podle těchto parametrů. Malou nevýhodou je, že tyto složky lze uložit pouze do domovské složky, „Saved Search“ nebo na plochu, případně je umístit rovnou do „Sidebaru“. Po uložení je možno tyto složky kamkoliv přesunout.

Spotlight je přístupný definovatelnou klávesovou zkratkou kdekoliv v systému a umožňuje velmi rychlé hledání souborů podle zadaného názvu. Stejný způsob hledání ve Finderu je tedy redundantní a některým lidem chybí původní implementace vyhledávání, neboť Spotlight jim z různých důvodů nevyhovuje.

2.3.9 Výhody a nevýhody současné verze Finder

Finder v poslední verzi Mac OS X 10.4 Tiger je velmi pokročilou aplikací, jež nabízí poměrně pohodlný přístup k souborům na disku. Finder nabízí prakticky neomezený počet otevřených oken, ve kterých je zobrazen obsah některé ze složek na disku.

Tyto výpisy mohou být ve třech formátech: Ikony (Icons), Seznam (List), Sloupce (Columns). Při prvním způsobu zobrazení je každý soubor, složka nebo aplikace zobrazena jako ikona zvolené velikosti. Tyto ikony lze řadit, přichytit do mřížky a podobně. Je to mé základní nastavení Finderu. Druhý způsob zobrazí obsah složky jako seznam položek s podrobným popisem vlastností (datum vytvoření, typ, velikost ...). Každou složku lze buď otevřít jako nový seznam nebo otevřít jako další větev stromu. Tak Finder umožní zobrazit více složek aniž by bylo potřeba mít otevřeno více oken. Posledním způsobem zobrazení je sloupcové, které je známe ze všech dialogových oken pro ukládání a otevírání souborů. Obsahy složek se otevírají ve sloupci napravo a celý obsah okna při hierarchickém postupu složkami horizontálně roluje. Tento způsob je převzat z NeXTSTEP a umožňuje velmi rychlou navigaci, protože je vidět celá struktura od kořenové složky až po aktuálně vybraný soubor.

Velmi užitečnou vlastností Finderu je, že si pamatuje nastavení zobrazení pro jednotlivé složky. To se vztahuje na způsoby řazení i zobrazení. Například složka, do které se ukládají stažené věci, může být řazena podle data vytvoření souboru. Při otevření této složky budou nejnovější soubory vždy první a není potřeba měnit způsob řazení, protože toto nastavení je platné pouze pro tuto složku. Otevře-li uživatel složku v novém okně, pamatuje si i poslední velikost okna. Nové okno se otevře stisknutím klávesy `Command` při poklepání myší na ikonu nebo přidržení klávesy `Shift` při použiti klávesové zkratky.

Prakticky veškeré operace práce se soubory se provádí přetahováním myší (drag & drop) mezi složkami, případě okny. Výchozí akcí je přesun vybraných objektů. Chce-li uživatel objekty kopírovat, přidrží při uvolnění myši klávesu `Option`. U kurzoru myši se zobrazí malé plus, které znázorňuje, že objekty budou kopírovány. Kombinací kláves `Option` a `Command` dojde k vytvoření odkazů na objekty. Odkazy na objekty mají svou ikonu doplněnou malou šipkou, která se zobrazí u kurzoru myši při vytváření. Mazání lze provést buď přetažením objektů na ikonu odpadkového koše nebo rychleji pomocí klávesové zkratky.

Díky automaticky se otevírajícím složkám, které zobrazují svůj obsah, podrží-li uživatel při tažení nad nimi kurzor, a při vhodném využití funkce implementované v poslední verzi Mac OS X nazvané Exposé, je tento způsob práce se soubory velmi intuitivní a pohodlný, které zvládne i počítačový laik. Při přetahování se nelze bohužel jednoduše dostat v adresářové struktuře o úroveň výš a nelze využít jak kolečko myši pro rolování obsahu okna, tak ani pohyb po složkách psaním jejich názvu. Je-li nějaká složka, do které je potřeba se vnořit mimo aktuální obsah okna, musí se rolovat pouze umístěním kurzoru na horní nebo dolní okraj okna.

Na druhou stranu Finder nabízí množství klávesových zkratek, které zefektivňují některé často používané akce. Znalost klávesových zkratek však vyžaduje pokročilého uživatele, který se s aplikací Finder již dostatečně seznámil.

2.3.10 Přechod uživatelů z jiných OS

Uživatelé můžeme v zásadě rozdělit na dvě hlavní skupiny podle způsobu práce se soubory. V jedné skupině budou ti, kteří pracují s vestavěnými programy operačního systému. Uvážíme-li nejrozšířenější operační systémy, kterými beze sporu jsou MS Windows a Linux či některá z implementací BSD, kde hraje roli použité prostředí pro práci s okny (hlavní role hrají role Gnome a KDE), dostaneme velmi podobné nástroje: MS Windows - Explorer (Průzkumník), Gnome - Nautilus, KDE - Konqueror. Do druhé skupiny zahrneme uživatele, kteří používají , jejichž základním prvkem jsou 2 panely, zobrazující nezávisle na sobě obsahy dvou složek. Typickými představiteli těchto aplikací jsou Total Commander, Servant Salamander, konzolový Midnight Commander, Krusader a další.

První skupina uživatelů nebude muset při přechodu na Mac OS X měnit své návyky, neboť Finder je výše zmíněným aplikacím koncepčně velmi podobný ba dokonce díky některým vlastnostem je jeho používaní snazší a pohodlnější. Pro Explorer i Finder existují alternativní náhrady vyvinuté cizími společnostmi, které rozšiřují základní funkce ale ponechávají stejnou koncepci.

Druhá skupina však při přechodu na Mac OS X bude postrádat souborový manažer takových kvalit, na jaké je zvyklá z jiného OS. Vzhledem k tomu, že je Mac OS X poměrně mladým operačním systémem s velmi dlouhou historií svých předchůdců mající svůj zaběhnutý systém práce se soubory, který většině uživatelů vyhovuje, je pochopitelný nedostatek správců souborů pracující s dvěma panely. Jeho potřeba se objevuje až v současně době, kdy si Mac OS X získává uživatele jiných operačních systémů.

Pravě pro uživatele patřící do druhé skupiny, jsem se rozhodl vytvořit souborový manažer, který bude šířen zdarma.

2.3.11 Ostatní správci souborů

Jak jsem zmínil v předchozí kapitole, potřeba souborových manažerů, které pracují se dvěma panely v jednom okně, se objevuje teprve v poslední době s přicházejícími novými uživateli. Pro dlouholeté zákazníky je, podle ohlasů na diskuzních fórech, Finder vyhovující a dostatečně efektivní. Pro ty ostatní je potřeba nástroje, který jim umožní pohodlnou práci se soubory tak, jak jsou na ni zvyklí.

Současný výběr je velmi skromný. MuCommander je volně dostupnou, multiplatformní aplikací napsanou v Javě. Odtud pramení jeho hlavní nevýhoda a tou je jeho malá rychlost a poměrně velká paměťová náročnost. Umožňuje standardní práci se soubory, připojení na FTP, SFTP a SMB servery, které podle mých zkušeností nejsou zcela stabilní a způsobují zatuhnutí celé aplikace. Práce s ním není příliš pohodlná, ale jeho hlavní předností je jeho nulová cena a možnost jej používat na libovolné platformě (Mac OS, Linux, Solaris, MS Windows). Jeho vývoj příliš nepokračuje.

Disk Order je naopak aplikace komerční (stojí 22.57USD), je napsána v Cocoa a poskytuje komfort a rychlost, na kterou jsou uživatelé Mac OS X zvyklí. Umožňuje například procházení archivů

jako by to byly obyčejné složky, připojování FTP serverů, připojení k .Mac a další. Disk Order je kvalitní nástroj velmi srovnatelný se svým protějškem Total Commander nebo Salamendrem, kteří hrají první housle na platformě MS Windows. Jeho 30ti denní verze je k dispozici zdarma.

Za zmínku stojí také Midnight Commander, který je také zdarma, a existuje pouze v textové verzi spustitelné v terminálu. Jeho instalace je však záležitostí pro pokročilé uživatele. Je potřeba zkompilovat nejen samotnou aplikaci, ale také několik dalších, které využívá (např. `ncurses`). Vzhledem k tomu, že běží pouze v Terminálu, neumožňuje žádné operace s myší.

Kapitola 3

Vývoj aplikací v Mac OS X

Tato kapitola čerpá z [Dav02].

3.1 Popis programovacích prostředí v Mac OS X

3.1.1 Carbon

Procedurální API pracující s Mac OS X. Toto rozhraní je původně převzato z dřívější verze Mac OS Toolbox API a je upraveno, aby bylo schopno pracovat pod Mac OS X s využitím chráněného prostředí paměti a preemptivním plánování úloh. Jako přechodné API, Carbon umožňuje vývojářům přímou cestu k migraci starších aplikací do Mac OS X bez potřeby kompletního přepsání. Adobe Photoshop 7.0 a Microsoft Office X jsou příkladem „Karbonizovaných“ aplikací. To, že je prostředí označováno jako přechodné, neznamená, že by mělo s postupem času vymizet. Zůstane kvůli podpoře starších aplikací a softwarový inženýři firmy Apple usilují o lepší integraci mezi prostředími Carbon a Cocoa. Velmi důležitou aplikací je Finder, pro který byl Carbon vybrán, aby Apple dokázal jeho kvality.

3.1.2 Cocoa

Objektově-orientované API vycházející z technologií operačního systému NeXT a přebírá mnoho výhod z Carbon. Velké množství aplikací dodávaných s operačním systémem Mac OS X, jako například Mail, Stickies, iCal, jsou napsány právě v Cocoa. Programování s Cocoa je primárním zaměřením této diplomové práce a bude podrobně rozebráno v kapitole 3.2 na straně 3.2.

3.1.3 Java

Robustní a relativně rychlý virtuální stroj pro běh aplikací, které jsou vyvinuty v JDK (Java Development Kit). Java aplikace jsou typické svou přenositelností a možností, aby běžely v různých prostředích bez rekompilace. V současně době je implementována Java 1.5, která je plně integrována do systému. Aplikace napsané v Javě jsou znatelně pomalejší než nativní aplikace, ale mají řadu předností. Lze vytvořit vše kromě rozhraní v Javě, které je použitelné na všech platformách, které pro které je Java dostupná a grafické rozhraní, které je nejpomalejší napsat v některém z nativních prostředí. V Mac OS X má Java vytvořené propojení s Cocoa.

3.1.4 BSD Unix

BSD vrstva v Mac OS X poskytuje bohatou, robustní a vyspělou skupinu nástrojů a systémových volání. Součástí této vrstvy je je prostředí příkazové řádky. Programy z této vrstvy lze velmi pohodlně volat přímo z Cocoa aplikací a proto tato vrstva poskytuje velmi solidní základ pro tvorbu nejrůznějších aplikací.

3.1.5 Classic

Prostředí, které je implementováno z důvodu kompatibility s předchozími systémy, umožňuje běh aplikací, které byly napsány pro MAC OS 8 a MAC OS 9 a nebyly přepsány aby získaly všechny výhody systému Mac OS X. Prostřední Classic je mírně modifikovaná verze MAC OS 9, která běží uvnitř systému jako speciální proces. Význam tohoto prostředí klesá s klesajícím množstvím aplikací, které nemají svůj protějšek naprogramovaný s využitím jiného moderního aplikačního prostředí. Ve verzi Mac OS X pro procesory Intel není již vrstva Classic podporována a bylo od ní upuštěno.

3.2 Cocoa

3.2.1 Historie Cocoa

Cocoa není ve světe počítačů žádným nováčkem. Jeho historie sahá až k počátkům Macintoshe samotného. Vyplývá to z toho, že je založen na systému NeXTSTEP, který byl světu představen v roce 1987. Ten prošel mnoha vydáními, byl převzat mnoha společnostmi jako jejich vývojové prostředí a opěvován mnoha články v odborných časopisech. Byla to, a bude, spolehlivá technologie založená na designu, který byl tehdy několik let před čímkoliv na trhu.

NeXTSTEP byl postaven na základech BSD Unixu z UC Berkley s využitím mikrojádra Mach z Carnegie-Mellon University. Využíval PostScriptu od firmy Adobe, který umožňuje použití stejného kódu pro zobrazení dokumentu na obrazovku nebo vytištění na papír. NeXTSTEP přišel s bohatou sadou knihoven nazývaných „frameworky“ a nástroji umožňujícími programátorům vytvářet aplikace s použitím jazyka Objective-C.

V roce 1993 NeXT opustil svět hardware a začal se koncentrovat pouze na software. NeXTSTEP byl portován na architekturu Intel x86 a vypuštěn. Další portace byly představeny na architekturách SPARC, Alpha a PA-RISC. Později byly tyto frameworky a nástroje revidovány a portovány pro běh na systémech Windows a Solaris. Tyto revidované frameworky byly známy pod názvem OpenStep.

3.2.2 Popis Cocoa

Cocoa je pokročilý objektově-orientovaný framework určený pro vytváření aplikacích běžících na Mac OS X. Skládá se ze skupiny knihoven sdílených objektů, runtime systému a vývojového prostředí. Cocoa poskytuje většinu infrastruktury, kterou grafické uživatelské aplikace potřebují, a odstiňují aplikaci od vnitřních záležitostí operačního systému.

Představme si Cocoa jako vrstvu objektů, jež představují zprostředkovatele a pomocníka mezi programy, které vytváříme, a operačním systémem. Tyto objekty pokrývají velmi široké spektrum od zapouzdření základních typů, jako jsou řetězce a pole, ke složitým problémům, jako je distribuované výpočty nebo garbage collector. Jsou navrženy tak, aby umožnily jednoduché vytvoření grafického uživatelského rozhraní (GUI) aplikace, a jsou založeny na důmyslném základě, a tak zjednodušují programátorské úlohy.

Aplikace založeny na Cocoa nejsou omezenou pouze na vlastnosti tohoto frameworku. Mohou také využívat veškerou funkcionalitu ostatních frameworků, jež jsou součástí Mac OS X. Sluší se vyjmenovat alespoň tyto Quartz, QuickTime, OpenGL, ColorSync ale také mnoho dalších. A také díky tomu, že je Mac OS X postaven na Darwinovi (spolehlivý BSD-orientovaný systém), mohou Cocoa aplikace využívat velmi širokou škálu funkcí Unixového jádra a dostat se tak až na úroveň souborového systému, síťových služeb nebo zaražení tak, jak potřebují.

Cocoa se skládá ze dvou objektově-orientovaných frameworků: Foundation (neplést s Core Foundation, které je frameworkem nezávislým na Cocoa) a Application Kit. Třídy z části Foundation framework poskytují objekty a funkce které jsou základními kameny Cocoa, a které nemají co do činění s uživatelským rozhraním. Naproti tomu třídy AppKit jsou postavený právě na třídách z Foundation a poskytují objekty a vlastnosti, které váš uživatel uvidí v uživatelském rozhraní, jako jsou okna a tlačítka; tyto třídy také zajišťují události jako kliknutí myší a stisknutí kláves. Stručně řečeno Cocoa Foundation poskytuje funkcionalitu, která pracuje pod povrchem aplikace, zatím co AppKit poskytuje funkce pro uživatelské rozhraní, které uživatelé vidí.

3.2.3 Foundation Framework

Foundation framework je skupinou čítající přes 80 tříd, které vytváří vrstvu základních funkcí pro aplikace Cocoa. K tomu tento framework přináší několik postupů definujících konvence správy paměti a objektů. Tyto konvence umožňují programátorům vytvářet kód mnohem rychleji a efektivněji používáním stejných mechanizmů na nejrůznější typy objektů. Obrázek 3.1 ukazuje jak spolu jednotlivé skupiny objektů ve frameworku souvisí.

Foundation framework obsahuje:

- prapředka všech tříd, `NSObject`
- třídy reprezentující základní datové typy, např. řetězce nebo bytová pole
- třídy kolekcí pro uchování jiných objektů
- třídy reprezentující systémové informace a služby

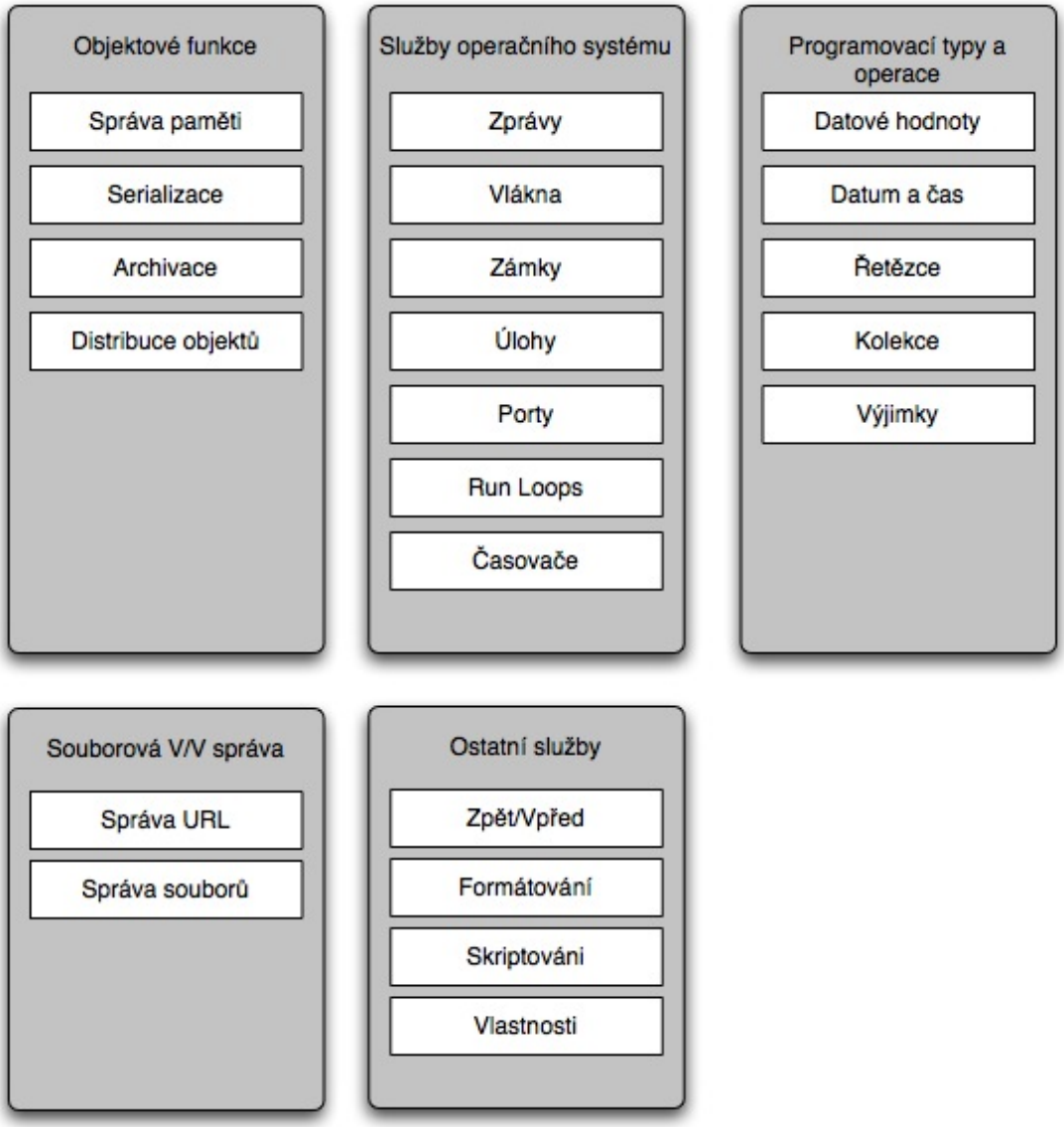
Programovací typy a operace

Foundation framework poskytuje mnoho základních typů včetně řetězců a čísel, ale také několik tříd, jejichž účel je shromažďování jiných objektů - pole a slovníky (dictionary).

Řetězce (Strings) nabízené frameworkem Cocoa, jsou prezentovány třídou `NSString` a nahrazují datový typ známy `char *` z jazyka C. Tyto objekty obsahuji znaky v kódování Unicode a mohou tak obsahovat text v libovolném jazyce včetně čínštiny, arabštiny nebo hebrejštiny. API poskytuje třídy pro vytváření měnitelných a neměnitelných objektů, včetně metod pro práci s řetězci jako je vytváření podřetězců, hledání, porovnávaní nebo spojování.

Kolekce (Collections) slouží pro uchování dat v logické struktuře podle určitých pravidel. Data-objekty se mohou nacházet v poli s indexováním od 0, slovníku uchovávající data v párech klíč-hodnota nebo třeba množině, v níž jsou objekty neuspořádané a mohou se vyskytovat pouze v jednom exempláři (množinu definuje například jazyk Pascal; v pascalovské množině mohou být uchovány pouze ordinární typy, v Cocoa lze vložit do množiny libovolný objekt).

Kolekce mají dynamickou velikost a opět existují ve dvou variantách - měnitelné a neměnné. Jak už názvy napovídají měnitelné kolekce mohou po svém vzniku přijímat další objekty a



Obrázek 3.1: Foundation Framework

programátor může tyto kolekce měnit podle svých potřeb. Naproti tomu neměnné kolekce nemohou byt po svém vytvoření změněny. Tyto dvě varianty existují z pochopitelných důvodů kladených na rychlost.

Data a hodnoty (Data and value) jednoduše alokují vyrovnávací paměti, skalární typy a ukazatele. Datové objekty (data object) jsou objektově orientovaná zapouzdření pro vyrovnávací paměti a mohou uchovávat data libovolné velikosti. Datové objekty neobsahují žádné informace o uchovaných datech (např. jejich typ). Veškerá zodpovědnost, jak bude s daty naloženo, je pouze na programátorovi.

Pro data nějakého typu jsou hodnotové objekty (value objects). To jsou jednoduché kontejnery pro jednotlivé datové položky. Mohou obsahovat skalární typy jako např. integer, float, znaky, ale také ukazatele nebo adresy na objekt.

Datum a čas - třídy poskytující datum a čas nabízí metody pro výpočet časových rozdílů, zobrazení data a času v požadovaném formátu a přizpůsobení data a času na základě lokalizace (např. časové zóny).

Ošetření výjimek (Exception handling) je speciální podmínkou, která přeruší normální provádění programu. Výjimky umožňují ošetření chybových stavů jemným způsobem. Například pokud program potřebuje uložit soubor a zápis do chráněného adresáře se mu nezdaří, oznámí to uživateli dialogovým oknem.

3.2.4 Služby operačního systému

Foundation framework poskytuje třídy pro přístup k funkcím jádra operačního systému jako jsou zámky, vlákna a časovače. Spolupráce těchto služeb poskytuje robustní prostředí pro běh vytvářené aplikace.

Run loops je programové rozhraní, které umožňuje objektům spravovat vstupní zdroje. Run loops přijímají vstupy z různých zdrojů jako myš, klávesnice, události systémových oken, z portů, časovačů a dalších připojení. Každé vlákno má svůj run loop vytvořený automaticky. Automaticky je vytvořen i pro celou aplikaci v době jejího spuštění. Run loops vláken, která jsou vytvořena ručně, musí být ručně také nastartovány.

Zprávy (Notifications) slouží v systému k zasílání o změnách v rámci aplikace. Objekt může specifikovat a odeslat zprávu a jakýkoliv jiný objekt může zaregistrovat sebe jako příjemce této zprávy.

Vlákna (Threads) je spustitelná jednotka, která má svůj vlastní zásobník a je schopna mít nezávislý vstup/výstup (V/V). Všechna vlákna sdílí virtuální adresový prostor. Ve chvíli kdy je vlákno spuštěno je odpoutáno od svého mateřského vlákna a běží dále nezávisle. Různá vlákna stejné aplikace mohou běžet ve stejnou chvíli na různých procesorech, jsou-li v systému k dispozici.

Zámky (Locks) slouží ke koordinaci operací několika vláken spuštěných v rámci jedné aplikace. Zámek může být použit jako prostředník k přístupu ke globálním datům aplikace nebo také jako ochrana kritické sekce kódu a umožnit tak aby mohla běžet atomicky - takto označenou část kódu může provádět současně pouze jedno vlákno a přistupovat tak ke chráněným prostředkům.

Úlohy (Tasks) umožňují stručně řečeno spustit v rámci jednoho programu jiný program jako jeho podproces a sledovat jeho provádění. Od vláken se liší zásadně tím, že běží samostatně a nesdílí paměťový prostor.

Porty (Ports) představují komunikační kanály do nebo z jiného portu který bývá typicky v jiném vláknu nebo úloze. Tyto komunikační kanály nejsou omezeny pouze v rámci jednoho počítače, ale mohou být rozšířeny po celé síti.

Časovače (Timers) jsou objekty, které zasílají zprávy jiným objektům ve specifických intervalech. Tyto časovače můžeme přirovnat k budíku, který „zazvoní“ v předem určený čas.

Objektové funkce (Object Functionality)

Foundation framework poskytuje služby pro správu vašich objektů-od vytváření a ničení po ukládání a sdílení v distribučním prostředí.

Správa paměti (Memory management) zajišťuje, že objekty jsou správně odstraněny ve chvíli, kdy již nejsou potřeba. Tento mechanismus, který závisí na obecném souhlasu majitelů objektů, automaticky vystopuje objekty, které jsou označeny „k odstranění“ a odstraní je v nejbližším možném čase. Správné pochopení správy paměti je důležité pro vytváření korektních Cocoa aplikací.

Serializace a archivace (Serialization and archiving) umožňují reprezentovat data, které objekt obsahuje, v nezávislém formátu, které pak lze sdílet napříč mezi aplikacemi. Speciální sterilizátor, známy jako Coder, rozšiřuje tento krok o ukládání informací o třídě spolu s daty objektu. Archivace ukládá kódované objekty a data do souborů, které lze použít pro pozdější běh aplikace nebo pro distribuci.

Distribuované objekty (Distributed objects) je skupina tříd postavených nad porty a umožnující meziprocesorové zasílání zpráv. Tento mechanizmus umožňuje aplikacím zpřístupnit jeden nebo více objektů dalším aplikacím na stejném stroji nebo na vzdáleném počítači. Distribuované objekty jsou velmi široce zpracovány a nabízí velké možnosti.

Soubory a V/V správa (File and I/O Management)

Souborový systém a vstupně/výstupní (V/V) operace zahrnují správu URL, souborový management, dynamické načítání kódu a lokalizované zdroje.

Souborová správa (File management) je skupinou utilit pro práci se soubory, které umožňují vytvářet adresáře a soubory, získávat obsah souborů jako data objektů, měnit pracovní pozici na souborovém systému a mnohé další. Vedle nabízené široké funkcionality dokonale odstiňují aplikaci od vrstvy souborového systému a umožňují používat stejné operace pro práci na lokálním disku, CD-ROMu nebo napříč sítí.

Správa URL (URL Handling) je poskytována speciálně vytvořenými objekty pro práci s URL. Objekty Cocoa, které umí číst nebo zapisovat do nějakého soubory, jsou schopný většinou akceptovat URL namísto klasické cesty k souboru.

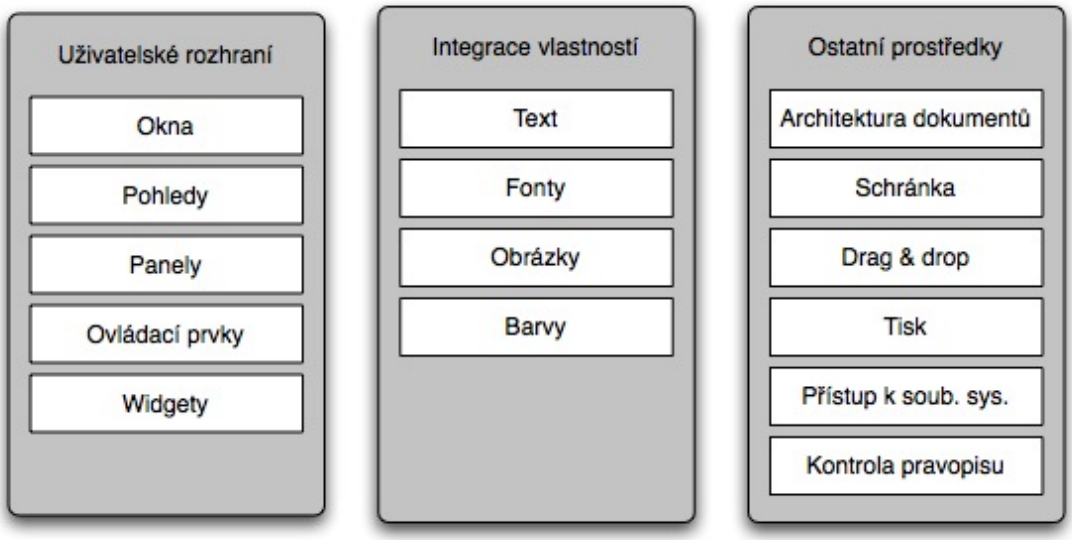
Další služby

Foundation framework poskytuje možnosti pro správu uživatelských nastavení, akcí zpět/vpřed, formátování dat a lokalizace do mnoha jazyků. Aplikace Cocoa mohou také reagovat odpovídajícím způsobem na příkazy AppleScriptu.

3.2.5 Application Kit framework

Application kit framework (nebo také častější zkrácený název AppKit) obsahuje skupinu čítající přes 120 tříd a příbuzných funkcí, které jsou potřeba pro implementaci grafického, událostmi řízeného uživatelského rozhraní. Tyto třídy implementují potřebnou funkcionalitu k pohotovému kreslení uživatelských prvků na obrazovku, komunikaci s video kartou a obrazovou vyrovnávací pamětí a zpracování událostí z klávesnice a myši.

Na první pohled se může zdát množství tříd v AppKitu velmi nepřehledným a od práce s ním odrazujícím, ale pro práci není potřeba znát vlastnosti každé třídy. Většina tříd AppKitu jsou pomocné třídy, které pracují v zákulisí a pomáhají ostatním třídám pracovat programátor k nim nikdy nepřistupuje přímo. Obrázek 3.2 ukazuje jak jsou třídy uspořádány do skupin a jaké jsou jejich vzájemné vztahy.



Obrázek 3.2: Application Kit

Uživatelské rozhraní

Okna (Windows) mají dvě základní funkce. První z nich je poskytnutí ohraničeného prostoru na obrazovce, do kterých jsou umísťovány pohledy a druhou je odchytávání událostí z myši a klávesnice. Oknům můžeme měnit velikost, minimalizovat je do Docku nebo zavírat. Všechny tyto akce generují události, které mohou být odchytávány v aplikaci.

Pohledy (Views) je abstraktní reprezentace pro všechny objekty zobrazené v okně. Pohledy poskytují plochu pro kreslení, tisk a odchytávání událostí. Pohledy jsou sestavovány uvnitř okna a jsou seskupovány hierarchicky do podpohledů (subviews).

Panely (Panels) jsou speciální typy oken, které jsou používány pro zobrazování dočasných, globálních nebo důležitých informací. Panely by mely být používány raději než okna k zobrazení chybových hlášek nebo dotazů k uživateli na významné nebo neobvyklé okolnosti.

Application Kit implementuje některé obecné panely, jako například dialogy otevření, uložení nebo tisku. Tyto obecné panely zajišťují uživatelům konzistentní vzhled a chování napříč všemi aplikacemi.

Ovládací prvky a widgety (Controls and Widgets) jsou množinou obecných objektů, které tvoří uživatelské rozhraní, jako jsou tlačítka, posuvníky nebo prohlížeče. Těmi se aplikace graficky ovládá. Co přesně bude který prvek dělat záleží na programátorovi. Cocoa nabízí velkou škálu tlačítek, tabulek, kurzorů, posuvníků, „šuplíků“ - drawers (speciální okno, které se vysouvá jako šuplík po straně jiného okna), hotová menu a mnoho dalších widgetů.

Integrace vlastností (Feature Integration)

Application Kit dává vašim aplikacím možnost integrovat správu barev, fontů a tisku a umožňuje vytváření dialogových oken pro tyto účely.

Text a fonty (Text and fonts) - text může být vkládán buď do jednořádkových nebo velkých textových polí. Ty se používají podle typu vstupního prvku. Textům může být přidána schopnost formátování. Možnosti práce s texty jsou velmi vyčerpávající. Základní textová pole nabízí možnosti srovnatelné se schopnostmi WordPadu. Vytvoření obdobného programu vyžaduje napsání několika řádků kódu.

Obrázky (Images) - třídy pracující s obrázky zapouzdřují grafická data, která lze jednoduše a efektivně z dat uložených na disku zobrazit na obrazovce. Cocoa umí bez problémů pracovat s mnoha standardními typy jako JPG, TIFF, GIF, PNG, PICT a mnoha dalšími.

Barvy (Colors) - práci s barvami je umožněna rozmanitými třídami reprezentujícími přímo barvy nebo barevné náhledy. Práci zjednodušuje velmi rozmanitá podpora barevných formátů a reprezentací. Vytvořené jsou i panely zobrazující barevné palety, které umožňují uživatelům vybrat barvu přímo v programu.

Ostatní prostředky (Other Facilities)

AppKit poskytuje množství prostředků, které umožňují vytvořit robustní aplikaci nesoucí všechny vymoženosti, které uživatelé očekávají od aplikace běžící na Mac OS X.

Architektura dokumentů slouží pro vytváření dokumentově orientované aplikace, jako jsou například textové procesory, jsou jedním z obecných typů aplikací, které jsou vyvíjeny. Jsou kontrastem k aplikacím typu iTunes, které ke svému běhu potřebují jen jedno okno. Dokumentově orientované aplikace potřebují mít sofistikovanou správu oken. To zajišťují rozličné třídy AppKitu a vytvářejí tak architekturu pro tyto typy aplikací, zjednodušují práci, kterou by měl udělat programátor. Tyto třídy si rozdělují činnosti jako vytváření, ukládání, otevírání a spravování dokumentů v aplikaci.

Tisk (Printing) zajišťují třídy vykonávající činnosti tisku a poskytují všechny potřebné prostředky k tomu, aby bylo možno vytisknout informace v oknech a pohledech vaší aplikace. Mimo to umožňují vytváření PDF dokumentů.

Schránka (Pasteboard) je úložiště pro data, která se kopírují z vaší aplikace a umožní tak zpřístupnění dat všem aplikacím, které to zajímá. Schránka implementuje oblíbené operace pro práci se schránkou: vyjmout-kopírovat-vložit.

Drag-and-drop lze vytvořit s velmi malým programátorským úsilím a mohou být objekty přenášeny odkudkoliv kamkoliv. Application Kit ošetřuje všechny detaily pohybu myši a zajišťuje grafickou reprezentaci přenášených dat.

Přístup k souborovému systému (Accessing the filesystem) - souborová reprezentace přímo odpovídá souborům a adresářům na disku. Tyto třídy uchovávají obsah souboru přímo v paměti a umožňují tak jeho zobrazení, změny nebo přenosy do jiných aplikací. Poskytují také ikony pro přesouvané soubory nebo jejich reprezentaci jako přílohy. Panely pro otevírání a zavírání souborů rovněž přináší vhodné a důvěrně známé rozhraní k souborovému systému.

Kontrola pravopisu (Spellchecking) je zabudovaná pro jakoukoliv aplikaci napsanou v Cocoa jako jsou textové procesory, textové editory, emailové aplikace a další aplikace pracující s texty. Nad libovolným textovým polem lze provést kontrolu pravopisu. Kontrola pravopisu je omezena pouze na jazyky oficiálně podporované operačním systémem Mac OS X. Pro ostatní jazyky lze využít `aspell` a jím podporované slovníky.

3.3 Model-View-Controller Design Pattern

Designový návrh Model-View-Controller je již poměrně starý. Objevil se někdy v dobách prvních dnů Smalltalku. Jedná se o vysokoúrovňový návrh se zaměřením na globální architekturu programu a pokouší se klasifikovat objekty podle zaměření a sestavit z nich aplikaci.

Návrh vytváří tři abstraktní vrstvy: datová vrstva *Model*, prezenční vrstva *View*, řídící vrstva Controller. tyto vrstvy mají své definované role, které určují jaký typ funkcí budou objekty v aplikaci zastávat. Objekty se vždy vytváří jako příslušné do jedné konkrétní vrstvy. Tyto vrstvy jsou odděleny pomyslnými hranicemi a objekty komunikují navzájem s ostatními přes tyto hranice pomocí speciálních prostředků. Striktní rozdělení do tří vrstev není ve všech případech aplikací ideální a ve speciálních případech se používá kombinace dvou vrstev jako jedné.

3.3.1 Datová vrstva - uchování dat

Objekty datové vrstvy *Model* reprezentují a popisují jak název napovídá data. Uchovávají aplikační data a definují logiku pro práci s nimi. Dobře navržená MVC aplikace má všechna důležitá data zapouzdřena v objektech právě této vrstvy. Jakákoliv data, která jsou perzistentní částí aplikace (jedno jakým způsobem je persistence provedena, zda jsou data uloženy na disku, v databázi nebo děrných štítcích) jsou zase uložena po načtení do aplikace v objektech datové vrstvy.

V ideálním případě nemají objekty datové vrstvy žádné propojení na uživatelské rozhraní, které je použito pro jejich reprezentaci a editaci. Pokud máme například objekty vrstvy *Model* reprezentující osobu (vytváříme-li třeba Adresář), budeme chtít uložit například jméno a datum narození. Je dobré uložit tyto údaje do objektu Osoba, který bude patřit do nejnižší vrstvy. Uložení formátovacích řetězců nebo dalších informací, jak má být datum prezentováno, by mělo být provedeno na jiném místě.

V praxi není úplné rozdělení vždy tím nejlepším řešením a nabízí se velká flexibilita a prostor pro vlastní řešení. Obecně řečeno tyto objekty by neměly mít nic do činění s uživatelským rozhraním a prezentačními záležitostmi. Příkladem výjimky jsou třeba kreslící aplikace, ve kterých se budou nacházet objekty reprezentující nějaký grafický prvek určený pro zobrazení. V jeho případě bude

určitě vhodné aby věděl jakým způsobem má sám sebe vykreslit, protože důvod jeho existence je právě definice něčeho vizuálního. Ani v tomto případě by neměly být tyto objekty závislé na existenci konkrétní objektu z prezenční vrstvy a ani žádného jiného objektu a také by neměly být zodpovědné za to, aby věděly kdy se mají vykreslit. Vždy by měly být o vykreslení požádány nějakým jiným objektem (např plátno) o to, aby se zobrazily.

3.3.2 Prezenční vrstva - uživatelské rozhraní

Objekty z prezenční vrstvy *View* ví jak zobrazovat a případně editovat data získaná od objektů vrstvy datové. Naopak by tyto objekty neměly být zodpovědné za jejich ukládání. (To samozřejmě neznamená, že by nikdy za žádných okolností nemohly uložit data, které zobrazují. Tato vrstva může data ukládat do vyrovnávacích pamětí nebo provádět podobné triky z výkonových důvodů.) Po těchto objektech lze vyžadovat zobrazování pouze části datového modelu, celého modelu nebo kombinace mnoha různých a nezávislých modelů.

Objekty z prezenční vrstvy se snaží být znovupoužitelné a poskytovat konzistenci mezi aplikacemi. V Cocoa definuje Application Kit velké množství objektů z prezenční vrstvy. Používáním těchto objektů z Application Kit, jako jsou například `NSButton`, máme garantováno, že všechny tlačítka v našich aplikacích budou vždy stejná jako tlačítka v jiných Cocoa aplikacích, a poskytují tak uživatelům velkou míru předvídatelnosti chování v různých aplikacích.

Prezenční vrstva by měla zajistit správné zobrazování datového modelu. Důsledkem toho potřebuje vědět o změnách v datovém modelu. A protože datový model nemá být těsně propojen s prezenčním modelem, potřebují všeobecně použitelnou cestu jak si dát vědět, že došlo ke změně. Tento problém řeší zasílání objektů `NSNotifications` pokud u nich dojde k pozměnění nebo si definují jiný způsob pro zvládnutí oznámení změn prezenční vrstvy, což se děje zpravidla přes řídící vrstvu (Controller layer).

3.3.3 Řídící vrstva

Řídící vrstva svazuje dohromady datovou a prezenční a je jakým si prostředníkem v aplikaci. V této vrstvě bývá největší část logiky aplikace. Po objektech řídící vrstvy se velmi často požaduje aby zajistily prezenční vrstvě přístup k datům, kterou má zobrazit, a často jsou právě tou rourou, přes kterou získávají potřebné znalosti o těchto datech.

Díky tomu, že řídící vrstva bývá velmi striktně definována přímo pro potřeby aplikace, je umožněno aby zbylé dvě vrstvy byly obecné a využitelné v jiných aplikacích. Objekty řídící vrstvy bývají zpravidla nejméně znovupoužitelnými objekty v aplikaci, ale to je přijatelné. Není udržitelné mít všechny objekty nezávislé a čím více závislého kódu je v řídící vrstvě, tím snadněji můžeme znovu používat ostatní objekty.

Nejvíce řádků kódu bývá napsáno řídící vrstvě. Aby bylo možno toto velké množství kódu lépe spravovat, bývá občas vhodné rozdělit tuto vrstvu na dvě podvrstvy datově-řídící a prezenčně-řídící.

Datově-řídící vrstva je orientována na komunikaci s datovou vrstvou. Je primárně zodpovědná za práci s modelem a komunikuje s prezenčně-řídící vrstvou. Všechny operace týkající se práce s daty, výpočty, ukládání atd by mělo být implementováno v této vrstvě.

Prezenčně-řídící vrstva se koncentruje na komunikaci s uživatelským rozhraním. Zasílá mu zprávy o změnách v datech a upravuje výsledná data pro jejich zobrazení. Dále také přijímá události vyvolané uživatelem a zachycené v prezenční vrstvě.

3.4 Objective-C

3.4.1 Úvod do Objective-C

Jazyk Objective-C je jednoduchý počítačový jazyk navržený pro pokročilé objektově orientované programování. Tento způsob programování staví na starších myšlenkách, které rozšiřuje a spojuje je v nových a neobvyklých způsobech. Výsledkem je rozmanitý a čistý přístup k umění programování. Objektově orientovaný přístup umožňuje navrhovat programy mnohem intuitivněji, vyvíjet rychleji a dělá je přístupnější k úpravám a jednodušším k pochopení. To vede nejen k alternativním cestám výstavby programů, ale také k novátorským cestám k pochopení programovacích úloh.

3.4.2 Proč Objective-C

Jazyk Objective-C byl vybrán pro Cocoa framework z několika rozličných důvodů. Prvním nejdůležitějším je, že je to objektově-orientovaný jazyk. Obrovské množství funkcí, které jsou zabaleny v Cocoa frameworku, mohou být možné jenom díky objektově-orientovaným technikám. Za druhé, protože Objective-C je rozšířením ANSI C, může existující program v C být adaptován pro použití softwarových frameworků bez toho, aby se ztratilo cokoliv z práce, která byla věnována jeho vývoji. Díky tomu, že je Objective-C založen právě na C, můžeme čerpat všechny výhody, které jazyk C nabízí. Lze si vybrat kdy využijeme objektově orientované způsoby řešení (např. definování nových tříd) nebo kdy se budeme držet procedurálních technik (definování struktur a několika funkcí místo použití tříd).

Mimo to je Objective-C jednoduchý jazyk. Jeho syntaxe je velmi stručná, přímočará a jednoduchá k pochopení. Pro většinu nově začínajících programátoru, díky důrazu na abstraktním návrhu, nepředstavuje jeho pochopení žádné zásadní problémy.

V porovnání s ostatními objektově orientovanými jazyky založenými na C je Objective-C velmi dynamické. Překladač zachovává velké množství informací o objektech, které lze využít v době běhu programu. Rozhodnutí, která by jinak měla být provedena v době překladu, mohou být přenechána až kdy je program spuštěn. Například dynamika Objective-C přináší dvě výhody, které by byly velmi těžko získatelné v jiných objektově-orientovaných jazycích.

- Objective-C podporuje otevřený způsob dynamického bindování, způsob který vytváří prostor jednoduché architektuře pro interaktivní uživatelská rozhraní. Zprávy nemusí být nutně zpracovány třídou, která je přijala nebo metodou selektoru, a tedy softwarový framework může programátorovi dovolit vybrat si až za běhu a otevírá tak možnosti svobodného návrhu. Výrazy dynamické bindování, zprávy, třída, příjemce a selektor budou vysvětleny na příslušném jiném místě.

- Dynamika Objective-C umožňuje konstrukce pokročilých vývojových nástrojů. Rozhraní k běhovému systému poskytuje přístup k informacím o běžících aplikacích, a je tedy možné vyvinout nástroje k monitorování a odkrývání informací, e strukturám a aktivitám ležícím pod aplikacemi v Objective-C.

3.4.3 Objekty

Jak již název napovídá, objektově orientované programy jsou postaveny okolo objektů. Objekty spojují data s příslušnými operacemi, které mohou využívat nebo ovlivňovat tato data. V Objective-C jsou tyto operace známé pod názvem metody; data, která ovlivňují, se nazývají instanční proměnné.

V Objective-C jsou instanční proměnné schovány ve svém objektu. Přistup k nim je možný pouze přes metody daného objektu. Aby ostatní mohli získat informace týkající se objektu, musí existovat takové metody, které tyto informace poskytnou.

Kromě toho, ostatní objekty vidí pouze ty metody, které byly k tomuto účelu určeny. Je to stejné jako funkce v C chrání své lokální proměnné a skrývají je před zbytkem programu, tak objekty skrývají jak své instanční proměnné tak i implementační metody.

V Objective-C jsou identifikátory objektů datovým typem: id. Tento typ je definován jako ukazatel na daný objekt. Ve skutečnosti se jedná o ukazatel na instanční proměnné objektu, jako unikátní data objektu. Je to stejné jako v jazyce C, kde je funkce nebo pole identifikováno svou adresou. Všechny objekty bez ohledu na to jaká data nebo metody obsahují jsou datového typu id.

id anObject;

Pro objektově orientované konstrukce jazyka Objective-C, je jako základní návratová hodnota typu id a nahrazuje tak int. Pro striktní C konstrukce, jako jsou funkce, zůstává int základním návratovým typem.

Typ id je zcela nerestriktivní. Sám o sobě nenese žádnou informaci o objektu kromě toho, že se jedná o objekt. Objekty však nejsou vždy stejné. Metody, které má jeden objekt, samozřejmě nemusí mít objekt jiný. V těchto případech musí mít program více specifických informací o objektech, které obsahuje: jaké má tento objekt instanční proměnné, jaké metody nabízí a podobně. Díky tomu, že id tyto informace nemůže překladači poskytnout, každý objekt bude tyto informace dodat v době běhu.

To je možné díky tomu, že každý objekt nese s sebou instanční proměnnou isa, která identifikuje třídu objektu - jakého typu objekt je. Každý objekt je tak v době běhu schopen říci, které třídy je instancí.

Objekty jsou takto typovány až za běhu. Kdykoliv potřebuje běhový systém znát přesnou třídu, které objekt náleží, prostě se jej zeptá. Dynamické typování v Objective-C slouží jako základ pro dynamické bindování. O něm se zmíním později.

Ukazatel isa obsahuje informace překladače o definici třídy v datové struktuře pro použití běhového systému. Použitím běhového systému lze například zjistit zda nějaký objekt implementuje určitou metodu nebo získat jméno jeho nadtřídy.

I přes tyto výhody leze použít statické typování, kdy překladači oznámíme typ třídy již při překladu prostým nahrazením id názvem třídy jejíž instanci vytváříme. Lze tak odhalit již některé chyby, kdy budeme posílat objektu zprávu, které nerozumí. Překladač nevyhlásí chybu, ale pouze varovaní. Díky dynamice, kterou Objective-C přináší, překladač neví zda nebude této zprávě objekt rozumět v budoucnosti. Díky varování lze samozřejmě chyby také vysledovat, a proto se doporučuje používat spíše statické typování objektů.

3.4.4 Zasílání zpráv

Chceme-li, aby objekt vykonal nějakou činnost, zašleme tomuto objektu zprávu. V Objective-C jsou výrazy zpráv uzavřeny do hranatých závorek

```
[prijemce zprava]
```

Příjemce je objekt a zpráva mu říká co má dělat. Ve zdrojovém kódu je pak zpráva názvem metody s patřičnými parametry, které je potřeba předat objektu. Jakmile je zpráva odeslána, vybere běhový systém příslušnou metodu z repertoáru příjemce a provede ji.

```
[objekt zobrazit];
```

Metoda s argumentem bude vypadat takto:

```
[objekt zobrazitXY:20 :50];
```

Zde jsou za názvem metody uvedeny dvě dvojtečky, každá pro jeden argument. Argumenty jsou uváděny vždy za dvojtečkou. Dvojtečky nemusí být nutně seskupeny až za názvem metody jako zde. Obvykle se přidávají ještě popisy argumentů, které dvojtečku předchází. Pro přehlednost a lepší pochopení znovu čteného kódu by zpráva vypadala takto:

```
[objekt zobrazitX:20 Y:50];
```

Metody, které přijímají proměnné množství parametrů, jsou sice vzácné, ale ze je taktéž používat. Extra argumenty jsou odděleny čárkou za koncem jména metody. V následujícím příkladě máme smyšlenou metodu vytvorSkupinu: s jedním povinným argumentem a třemi volitelnými:

```
[prijemce vytvorSkupiny:skupina, clenJedna, clenDva, clenTri];
```

Tak jako standardní funkce v C mohou metody vracet nějakou hodnotu.

```
BOOL isFilled;
isFilled = [objekt isFilled];
```

Za povšimnutí stojí, že název proměnné i metody jsou stejné.
Výraz odeslání zprávy může sloužit jako parametr pro další zprávu:

```
[ctverec setPrimaryColor:[jinyCtverec] primaryColor]];
```

Odeslání zprávy do `nil` (neexistujícího objektu) je v pořádku, pokud odesílaná zpráva vrací objekt. Jestliže ano, odeslání zprávy do `nil` vrátí `nil`. Jestliže zpráva odeslaná `nil` má vracet cokoliv jiného, bude návratová hodnota nedefinována.

3.4.5 Dynamické bindování

Důležitým rozdílem mezi funkcemi a zprávami je ten, že funkce a jejich argumenty jsou propojeny dohromady již v kompilovaném kódu, zatím co zprávy a projímající objekty nejsou spojeny do té doby, dokud program neběží a zpráva není odeslána. To znamená, že konkrétní metoda, která bude vyvolána jako odpověď na zprávu, může být zjištěna pouze v době běhu, ale nikoliv v době kompilace. Přesná metoda, kterou zpráva spustí závisí na až příjemci. Rozdílní příjemci mohou mít rozdílné implementace pro pro stejný název metody. Aby překladač našel správnou implementaci metody pro konkrétní zprávu, musí znát jaký druh objektu příjemce je - do jaké třídy patří. Tuto informaci příjemce zpřístupní v době běhu, kdy obdrží zprávu (dynamické typování), ale není přístupná z typové deklarace nalezené ve zdrojovém kódu.

Výběr metody probíhá v době běhu. Když je zpráva poslána, rutina starající se o odesílání zpráv nalezne příjemce a metodu pojmenovanou ve zprávě. Nalezne také příjemcovu implementaci metody souhlasící s názvem volané metody a nasměruje ukazatel na příjemcovy instanční proměnné.

Dynamické bindování metod na zprávy jde ruku v ruce s polymorfismem a dává objektově orientovanému programování mnohem více flexibility a síly. A protože každý objekt může mít svou vlastní verzi metody, program může docílit rozmanitých výsledků nejenom díky mnoha druhům zpráv, ale také díky tomu, že tyto zprávy může odesílat rozdílným objektům, které mnoho tyto

zprávy přijímat. Příjemce může být vybrán až za běhu a volba může být ovlivněna externími faktory jako jsou například uživatelské akce.

Když se spouští kód založený na Application Kit frameworku uživatel například rozhoduje, který z objektů příjme zprávy z menu jako jsou `cut` (Vyjmout),`copy` (Kopírovat) a `paste` (Vložit). Zpráva je odeslána jakémukoliv objektu, který byl vybrán. Objekt, který zobrazuje editovatelný text by měl reagovat na zprávu `copy` jinak, než objekt, který zobrazuje naskenovaný obrázek. Kód, který odesílá zprávu, by se tím neměl vůbec zabývat nebo zvažovat všechny možnosti které mohou nastat. Každá aplikace vytváří vlastní objekt, který odpovídá svým vlastním způsobem na zprávy `copy`.

3.5 Programovací nástroje Xcode

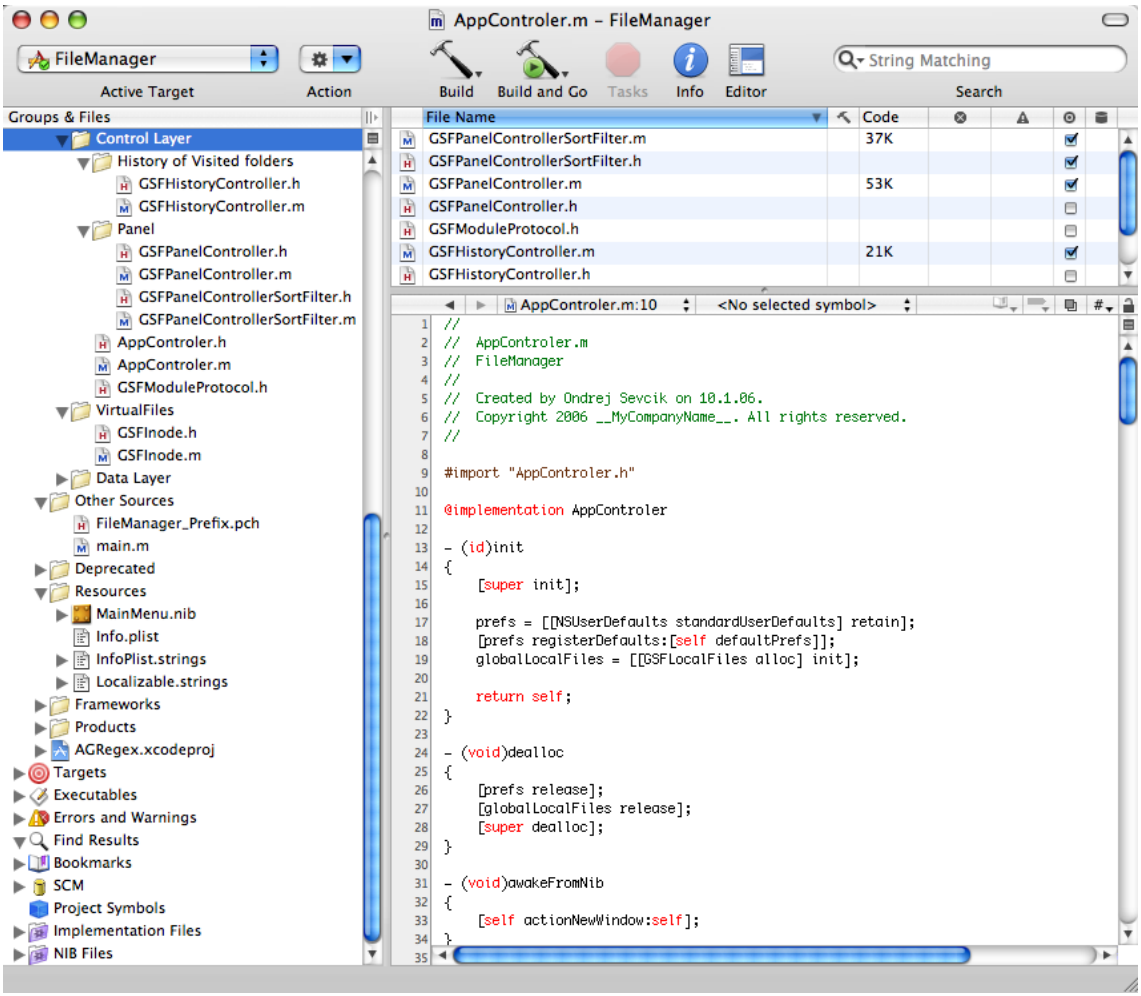
3.5.1 Project Builder

Xcode je vývojové prostředí složené ze dvou hlavních programů Project Builder a Interface Builder, které se používá pro vývoj aplikací v Mac OS X. Balík obsahuje také množství menších aplikací, které pomáhají ve vývoji. Lze v něm tvořit jak celé aplikace, tak nejrůznější doplňky jako jsou moduly pro ostatní aplikace, spořiče obrazovky, systémová nastavení, rozšíření jádra nebo nástroje pro příkazovou řádku. Při vytváření nového projektu se Xcode postará o vše potřebné. Založí adresářovou strukturu, vytvoří základní kostru projektu, připojí potřebné základní frameworky, nastaví překladač a výchozí `target`, což je vlastně požadovaný výstup, a otevře nachystaný nový projekt. Xcode skýtá vše co může vývojář potřebovat: správce souborů, ve kterém lze velmi rychle vyhledávat, samozřejmostí je editor souborů se zvýrazněnou syntaxí, velmi kvalitní a mocný debugger, který je založený na GNU gdb, integrovaný nástroj pro správu verzí a práci v týmu a mnoho dalších nástrojů, které pomáhají vyvářet robustní aplikace.

Zaměřím se pouze na vytváření aplikací založených na Cocoa a Objective-C. Hlavní okno projektu viz obrázek 3.3 zobrazuje v levém panelu editovatelnou strukturu souborů. Tato struktura je pouze virtuální a neodpovídá té, která je na disku. To je velmi výhodné, lze soubory uspořádat tak, aby se mohly velmi efektivně vyhledávat. V hlavním okně lze jak editovat tak zobrazovat soubory z vybrané větve virtuálního stromu. Na obrázku lze vidět zobrazení obojího. Xcode neumí zobrazit více souborů v jednom okně tak aby byly přímo přístupné přes záložky, tak jako to je například v Delphi, ale tyto záložky jsou přístupné přes menu v toolbaru, což při práci malinko zdržuje. Také se dá mezi těmito soubory přepínat pomocí klávesových zkratek. Editor je poměrně jednoduchý, ale umí vše co je potřeba pro psaní zdrojových kódů. Umí zvýrazňovat syntaxi, doplňovat kód a velmi dobře zvládá odsazování textu. Na jeho používání si programátor velmi rychle zvykne.

Velkou výhodou je integrovaný SCM systém na správu verzí s využitím některého s běžně používaných nástrojů: CVS, Subversion nebo Perforce. Vyzkoušel jsem a aktivně používal právě Subversion. Všechny potřebné funkce jsou přístupné buď přes hlavní menu nebo přímo v kontextové nabídce konkrétního souboru. Používání samotnému vývojáři lze SCM jen doporučit, protože se lze jednoduše vracet k předchozím verzím, porovnávat co se změnilo v souboru, když něco před chvílí ještě fungovalo a nyní aplikace „bezdůvodně“ padá. Při práci v týmu je SCM prakticky nutností, protože jednak předchází tomu, aby si programátoři navzájem přepsali svou práci, a také umožňuje jednoduchý přístup ke sdílenému úložišti projektu kdekoliv na síti.

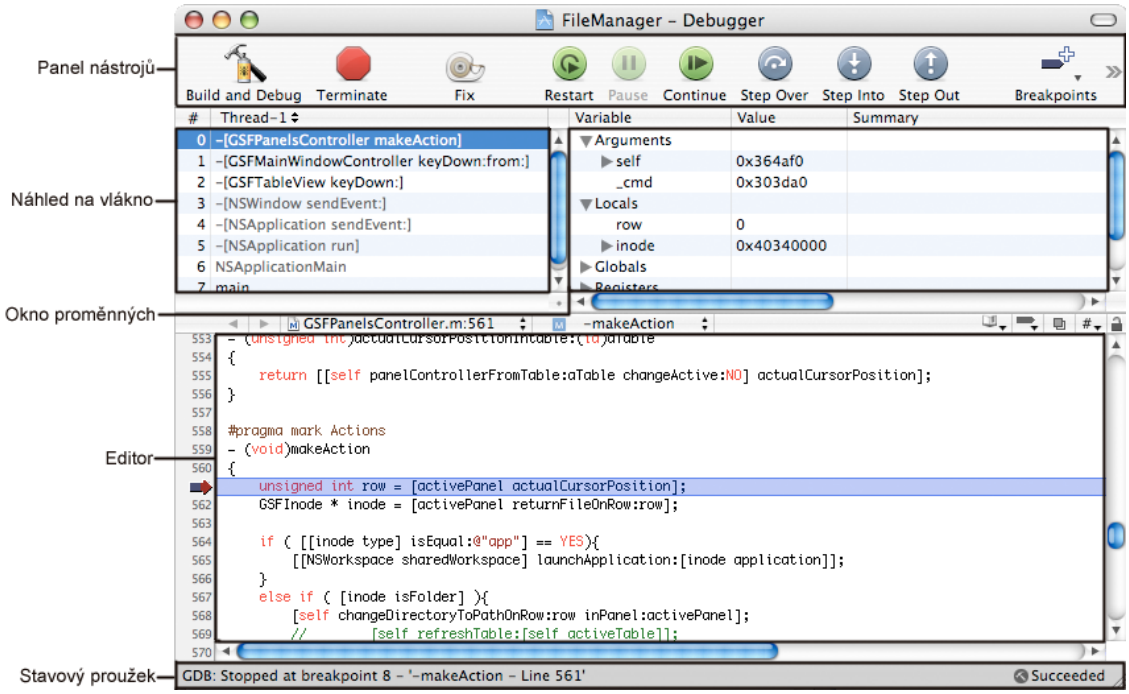
Velmi důležitou součástí je debugger. Pro správné debuggování je potřeba mít zapnutý přepínač překladače Generate Debug Symbols (GCC_GENERATE_DEBUGGING_SYMBOLS) a nesmí být zapnuta optimalizace zdrojového kódu. Já jsem si tím na dlouhou dobu znefunkčnil debugger, protože nebyl schopen správně procházet instrukce tak jak jsou ve zdrojovém kódu. Díky tomu, že Xcode umožňuje více konfigurací mezi nimiž lze jednoduše přepínat, není problém mít



Obrázek 3.3: Project Builder

pro vývoj zapnuto generování těchto symbolů a vypnutou optimalizaci a pro finální sestavování jen v menu vybrat jinou konfiguraci překladače.

Na obrázku 3.4 lze vidět okno debugger při zastavení na breakpointu, který lze vytvořit kdekoliv v kódu kliknutím na číslo řádku vlevo. Základní ovládání se neliší od jiných vizuálních debuggerů. Kód lze krokovat, zastavovat a znovu spouštět, zobrazovat hodnoty výrazů, vytvářet nové breakpointy atd. Užitečným ovládacím prvkem, ze kterým jsem se v jiných debuggerech nesetkal je „Step out“, který provede vyskočení z aktuálně prováděné metody a zastaví provádění. Náhled na vlákno ukazuje jak byly v daném vláknu předávány zprávy a je vidět odkud byla provedena jednotlivá odeslání zpráv. Okno proměnných zobrazuje rychlý náhled na dostupné proměnné hierarchicky rozčleněné. Záložka „Locals“ obsahuje lokálně vytvářené proměnné v rámci daného segmentu, část „self“ zase obsahuje proměnné aktuálního objektu apod.

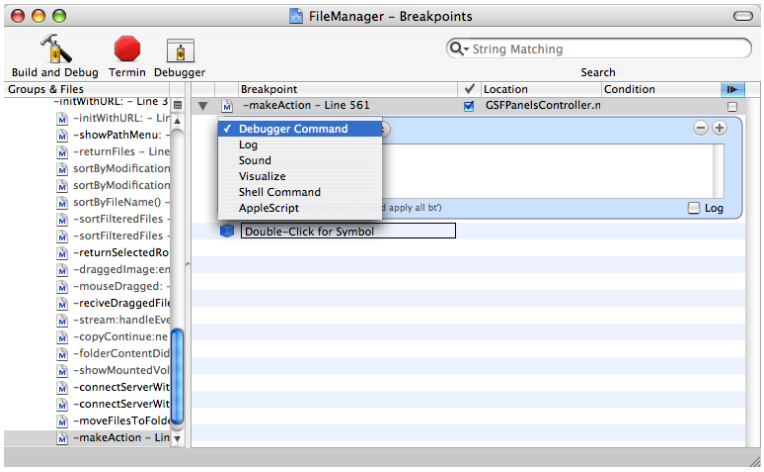


Obrázek 3.4: Debugger v Xcode založený na gdb

Breakpointy lze editovat ve správci (obrázek 3.5), kde jsou přístupné najednou všechny breakpointy projektu. Tady je možné nastavit breakpointy velmi podrobně. Například co se má při průchodu přes daný řádek provést. Může to být klasické zastavení provádění kódu, zapsání na standardní výstup, zvukové oznámení nebo co je velmi zajímavé, tak provedení shellového příkazu nebo AppleScriptu. Díky tomu lze vznikající aplikaci ladit velmi pohodlně.

Obrovskou předností integrovaného debuggeru je možnost jej ovládat pomocí konzole. Tuto vlastnost dědí ze svého předka „gdb“. Konzole se otevře pomocí ikony, která není na obrázku 3.5 vidět, protože se již nevlezla a je skryta v rozbalovacím menu přístupném přes poslední ikonu šipek. Konzole debuggeru se chová velmi podobně jako práce s Terminálem. Po zadání příkazu se odešle pomocí Return a výsledek se zobrazí hned na další řádce. Tak jako základní shell v Terminálu uchovává debugger historii zadaných příkazů, které jsou přístupné pomocí šípek nahoru a dolů. Provedení jednoho kroku se spustí napsáním `step`. První výstup se získá příkazem `print`.

```
(dbg) print row
```



Obrázek 3.5: Správce breakpointů

```
$2 = 5
```

Výstupem je obsah proměnné `row`, která obsahuje číslo 5. Proměnná je typu `unsigned int`, takže není potřeba uvádět typ. Ve všech ostatních případech je potřeba uvést správný typ vypisované hodnoty.

```
(dbg) print [inode isFolder]
Unable to call function at 0x90a3f0e0: no return type information available.
To call this function anyway, you can cast the return type explicitly
(e.g. 'print (float) fabs (3.0)')
```

Výsledkem je pouze chyba, protože návratovou hodnotou je typ `BOOL`. Správné znění příkazu je tedy:

```
(dbg) print (BOOL)[inode isFolder]
$4 = 1 '\001'
```

Tato hodnota vrací správně `YES`. Typ musí být uvedený u všech typů, které jsou jiné než `int` Speciálním případem je typ ukazatel na objekt. Příkaz `print` provede pouze vypsání ukazatele na daný objekt:

```
(gdb) print (GSFInode *)inode
$10 = (class GSFInode *) 0x37b890
(gdb) print (NSString *)[inode name]
$11 = (class NSString *) 0x3e6f80
```

Pro získání obsahu cílového objektu slouží příkaz `print-object`. Ten odešle zprávu `describe` a vypíše její návratovou hodnotu:

```
(dbg) print-object inode
<GSFInode: 0x37b890>
(dbg) print-object [inode path]
/Users/Ondra/Desktop
(dbg) print-object [inode name]
Desktop
```

Typ objektu nemusí být znám, protože se dynamicky přiřadí při odesílání zprávy a prakticky není důležitý, neboť pro odeslání zprávy jej není potřeba znát. Neumí-li objekt zpracovat zprávu `describe` [1] vrací jako v prvním případě, kdy byla odeslána objektu `inode`, je vypsán pouze ukazatel. Druhé dva řádky vrací hodnotu `NSString`, která umí metodu `describe` zpracovat.

Ovládání debuggeru pomocí příkazů však nabízí mnohem více. Nejzajímavější je možnost provádět libovolné příkazy, které nejsou vůbec nejsou v pořadí. K tomu slouží příkaz `call`:

```
(gdb) call (void)[inode setName:@"zkouška"]
(gdb) print-object [inode name]
zkouška
```

Jak je z ukázky patrné, došlo v objektu `inode` k nahrazení původního řetězce „Desktop“ novým řetězcem „zkouška“ zavoláním metody `setName:`. U příkazu `call` musí být uvedený typ návratové hodnoty, v tomto případě se jedná o `void`, protože metoda nevrací žádnou hodnotu. Tímto způsobem lze provádět velmi pokročilé a složité operace, které však mohou velmi usnadnit ladění některých problémů. V tabulce 3.1 jsou vypsány mnou nejpoužívanější příkazy.

`step`	provede jeden krok
`call [výraz]`	provede zadaný výraz, lze tak provádět vyhodnocování složitějších výrazů nebo třeba měnit celé objekty
`print [výraz]`	vytiskne primitivní hodnotu zadaného výrazu
`print-object [výraz]`	vytiskne hodnotu objektu zadaného výrazu
`set [proměnná] = [výraz]`	přiřadí hodnotu výrazu do proměnné. Například lze nastavit hodnotu objektu třídy `NSString` pomocí `set string = @"nová hodnota"`
`whatis [proměnná]`	vytiskne třídu nebo hodnotu předané proměnné
`help`	zobrazí seznam použitelných příkazů

Tabulka 3.1: Přehled nejpoužívanějších příkazů debuggeru

Podrobný vyčerpávající popis debuggeru by vystačil na celou knihu, a proto zůstanu pouze u tohoto stručného přehledu.

3.5.2 Interface Builder

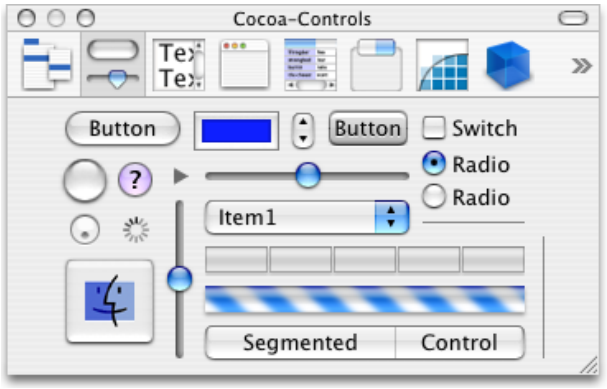
Interface Builder je vizuální editor pro tvorbu grafického rozhraní a vytvoření propojení s vrstvou kontroléru. Jeho ovládání je velmi intuitivní, ale vyžaduje znalosti způsobu návrhu a vytváření aplikací pomocí *Model – Control – View*. Umožňuje rozmisťovat okna a jejich prvky a ukládat tyto objekty do souborů. Lze také vytvářet instance vlastních tříd a vytvářet spojení mezi těmito instancemi a instancemi tříd uživatelského rozhraní. Když uživatel pracuje s objekty uživatelského rozhraní, spojení, která byla vytvořena během návrhu, umožní spuštění správného kódu.

V Interface Builderu jsou vytvářený takzvané *nib* soubory. Ty obsahují uložené instance objektů, které jsou vytvořeny v době návrhu. Při vytváření nového okna, se tak nevytváří nová instance, ale „probouzí“ se již vytvořené objekty z *nib* souboru. To je potřeba mít na paměti zvláště při inicializaci. Je-li potřeba inicializovat nějakou hodnotu až když se okno skutečně za běhu vytváří, neděje se to v konstruktoru, ale kontroléru okna se odesílá zpráva `awakeFromNib` a všechny

[1] Tuto třídu by měl implementovat každý objekt, ale pro účely ukázky jsem ji u třídy `GSFInode` odstranil.

inicializace závislé na okolnostech při běhu aplikace je potřeba umístit do této metody. Chybné umístění do metody `init` je zdrojem častých začátečnických chyb, které se velmi špatně hledají, zvlášť pokud o tomto problému programátor neví, protože při překladu i při krokování se vše zdá být v pořádku, ale objekt je špatně inicializovaný.

Interface builder poskytuje základní sadu běžně používaných prvků, které se nazývají `widgets`. Všechny jsou umístěny na paletě (obrázek 3.6), a do okna se umísťují přetažením pomocí myši. Vlastní komponenty se vytváří přímo v Xcode a zpravidla bývají potomkem třídy `NSView`. V Interface Builderu se tedy na plochu umístí `NSView` a nastaví se mu typ vytvořeného objektu. Interface Builder umožňuje také vytvořit těmto vlastním ovládacím prvkům palety a ty pak přímo využívat. Lze tak přímo v době designu pracovat s novou komponentou.



Obrázek 3.6: Okno palety ovládacích prvků

Jak již bylo zmíněno, je vhodné dodržovat Apple Human Interface Guidelines aby vznikající aplikace byla uživatelsky příjemná. V tom je Interface Builder maximálně nápomocný a poskytuje potřebné nástroje pro snadné dodržování těchto doporučení. Při umisťování prvků na plochu okna jsou automaticky přichytávány od ostatních prvků ve vzdálenostech, které odpovídají definicím Apple Human Interface Guidelines. Interface Builder dále nabízí také automatické změny velikosti všech vybraných prvků, zarovnání podle jednoho okraje apod. Práce s IB, jak bývá velmi často zkráceně označován, je velmi pohodlná a rychlá.

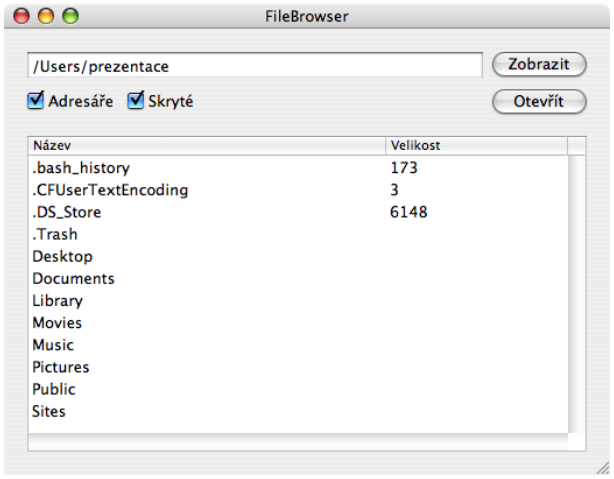
Kromě návrhu grafického rozhraní je důležitým posláním IB také napojení komponent na vrstvu kontroléru. Právě toto propojování bývá velmi častým zdrojem chyb, protože pokud není například nějaké tlačítko napojeno na konkrétní akci, nemá prakticky žádnou funkci a jeho stisknutím se nevyvolá žádná akce, tedy ani chybová. Pokud například aplikace nevykazuje chování takové, jaké programátor očekává, je vhodné znovu v IB zkontrolovat propojení komponenty, která nereaguje, se správnou akcí.

3.6 Ukázka vývoje aplikace - FileBrowser

3.6.1 Charakteristika vyvíjené aplikace

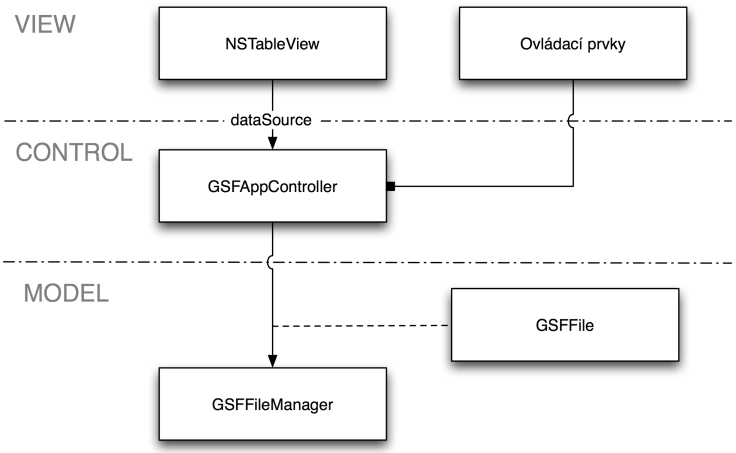
V této kapitole popíši vznik aplikace krok za krokem. Bude to rozšířený "Hello World!" ukazující třívrstvou aplikaci. FileBrowser je velmi zjednodušený prohlížeč obsahu disku, na kterém lze ukázat základní programovací principy, z nichž některé jsou využity přímo ve FileManageru. Výsledek, kterého chci dosáhnout je na obrázku 3.7. Do textového políčka se zadá cesta k adresáři a po stisknutí tlačítka *Zobrazit* dojde k zobrazení obsahu adresáře do tabulky. V tabulce je v levém

sloupci název souboru nebo složky, v pravém sloupci je velikost souboru. V tabulce se vybírá jeden řádek, jedná-li se o složku, pomocí tlačítka *Otevřít* ji lze otevřít, v případě souboru se otevře tento v příslušné aplikaci. Přepínač *Adresáře* respektive *Skryté* povolí zobrazení složek respektive skrytých souborů.



Obrázek 3.7: Aplikace FileBrowser

Aplikace je navržena do tří vrstev podle konceptu *Model – View – Control* , který je typicky pro Cocoa aplikace. Jak vrstvy mezi sebou spolupracují ukazuje obrázek 3.8. Vytvořeny jsou celkem tři třídy, které mají prefix `GSF`. [2]



Obrázek 3.8: Rozdělení vrstev v aplikaci

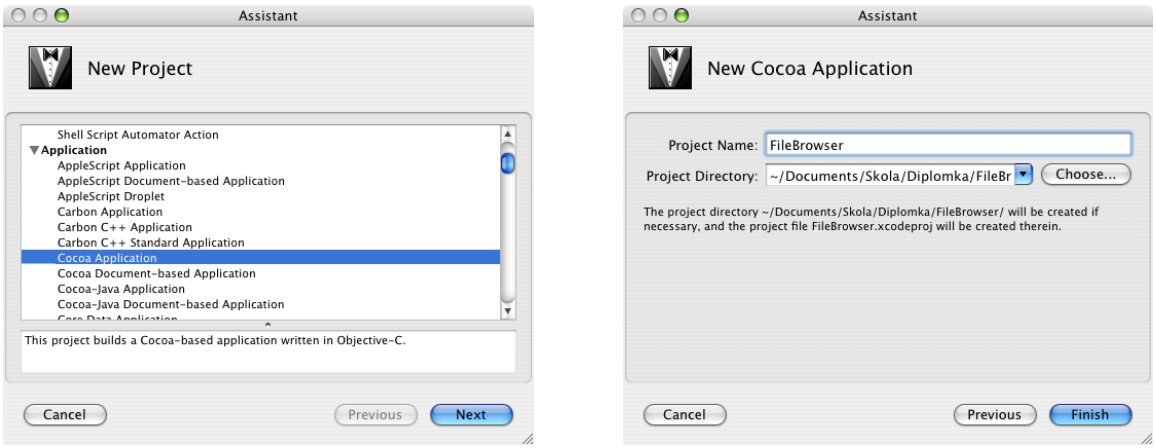
Vrstvu *View* tvoří samozřejmě ovládací prvky a tabulka pro zobrazení výsledků. (Popis po-

[2]Protože Objective-C nepoužívá namespaces, které můžeme znát z Javy nebo C++, musí být všechny názvy. Z tohoto důvodu se používají prefixy, které pak daný vývojář používá pro všechny své třídy a funkce. Téměř všechny funkce i třídy z frameworku Cocoa začínají prefixem `NS`, který představuje zkratku pro NeXTSTEP, a chrání tak tak svůj namespace proti kolizím. Používání `NS` i v Cocoa frameworku je důkazem dědictví z NeXTSTEPu. Pro své třídy a funkce používám prefix `GSF`.

užití tabulky viz strana 88.) Tato nejvyšší vrstva má na starosti interakci s uživatelem a zobrazení požadovaných výstupů. Je přímo napojena na třídu z prostřední vrstvy `Control` nazvanou `GSFAppController`, která zprostředkovává komunikaci ovládacích prvků s vrstvou nejnižší, která je zastoupena třídou `GSFFileManager`. Ta zpracovává data z reálného světa - obsah vybraného adresáře, a poskytuje třídu `GSFFile`, se kterou je schopný pracovat kontrolér aplikace. Na první pohled může toto rozdělení vypadat zbytečně, a zpracování obsahu adresáře by bylo možno zapracovat přímo do třídy `GSFAppController`, ale tím by došlo k omezení znovupoužitelnosti tříd. Třídu `GSFFileManager` lze totiž nyní nahradit třídou jinou, která by mohla poskytovat například obsah cesty na nějakém FTP serveru, aniž by bylo potřeba zasáhnout do zbytku aplikace.

3.6.2 Vytvoření nového projektu

Otevřu si Xcode a z menu File vyberu New Projekt. Xcode nabídné průvodce viz obrázek 3.9. Na první stránce „New Project“ si vyberu „Cocoa Application“ a stisknu tlačítko *Next*. Do názvu projektu zadám FileBrowser. Projekt si vytvoří složku, která je stejná jako jeho název, a do ní se ukládají všechny související soubory. Do textového pole lze vyplnit cestu umístění projektu. Projekt lze kdykoliv premístit tím, že se přesune celá složka na nové místo.



Obrázek 3.9: Průvodce novým projektem

3.6.3 Vytvoření kontroléru GSFAppController

Vývoj zahájím vytvořením hlavičkového souboru třídy `GSFAppController` (viz ukázka 1), abych si nachystal napojení na vrstvu *View*. Z menu `File` vyberu `New File...`, čímž otevřu dialog, kterým vytvořím novou třídu nazvanou `GSFAppController`. Dialog je velmi podobný tomu pro vytvoření projektu. Z nabídky vyberu z sekce Cocoa položku Objective-C class a v dalším kroku zadám její název. Soubor by měl mít koncovku .m, která se používá pro soubory jazyka Objective-C. Nechám zapnutou defaultní volbu „Also create GSFAppController.h“ a potvrdím tlačítkem „Finish“. Otevře se mi nové okno ve kterém je předvyplněna základní struktura hlavičkového souboru pro třídu jazyka Objective-C a naprogramuji zdrojový kód uvedený v ukázce 1.

Prvních šest položek začíná klíčovým slovem `IBOutlet`. To je ve skutečnosti pouhé makro, které se nahradí prázdným řetězcem, a slouží pouze pro Interface Builder, který podle něj pozná, že se jedná o Outlet pro napojení ovládacího prvku vytvořeném v Interface Builderu. Pro ovládací

Ukázka zdrojového kódu 1 Hlavičkový soubor GSFAppController.h

```
@interface GSFAppController : NSObject {
    IBOutlet NSTextField * path_textField;
    IBOutlet NSButton * run_button;
    IBOutlet NSButton * open_button;
    IBOutlet NSButton * folders_button;
    IBOutlet NSButton * hidden_button;
    IBOutlet NSTableView * content_tableView;

    GSFFileManager *fileManager;
    NSString *actualPath;
    BOOL showFolders;
    BOOL showHidden;
}

- (IBAction)showContent:(id)sender;
- (IBAction)open:(id)sender;

- (int)numberOfRowsInTableView:(NSTableView *)aTableView;
- (id)tableView:(NSTableView *)aTableView
     objectValueForTableColumn:(NSTableColumn *)aTableColumn
            row:(int)rowIndex;
@end
```

prvky je potřeba jednoho textového pole `NSTextField`, čtyř tlačítek `NSButton` a pro zobrazení jedena `NSTableView`. Pro úplnost uvedu, že všechny tyto ukazatele na objekt lze nahradit prostým `id`, které znamená ukazatel na libovolný objekt. Objective-C totiž nepotřebuje mít přesně uvedený typ a až za běhu se rozhodne zda daný objekt je schopen přijímat doručené zprávy. Přesné uvedení typu však zjednodušuje práci, protože pak při překladu dochází ke kontrole zda daný objekt akceptuje dané zprávy a pokud je jiného typu, oznámí překladač varování. Stejně tak Interface Builder při propojování nabízí jen relevantní objekty.

Proměnná `fileManager` je typu `GSFFileManager` a představuje propojení vrstev *Control* a *Model*. Proměnné `actualPath`, `showFolders` a `showHidden` jsou vnitřní proměnné řídící třídy, a budou vysvětleny později.

Třídá `GSFAppController` implementuje čtyři metody. První dvě metody mají jako návratový typ uvedeno `IBAction`. To je opět makro, které se nyní rozvine do typu `void`, a představuje pomocný text pro Interface Builder, který podle něj pozná že se jedná o Target/Action pro vyvoření propojení. Většina ovládacích prvků vyvolává nejakou akci, která způsobí spuštění napojené metody. Na příklad stisknutí tlačítka nebo změna polohy posuvníku atd.

Metoda přijímající akci musí mít jeden parametr typu `id`, do kterého je předán odesílatel zprávy. V aplikaci FileBrowser ji ale nebudu využívat.

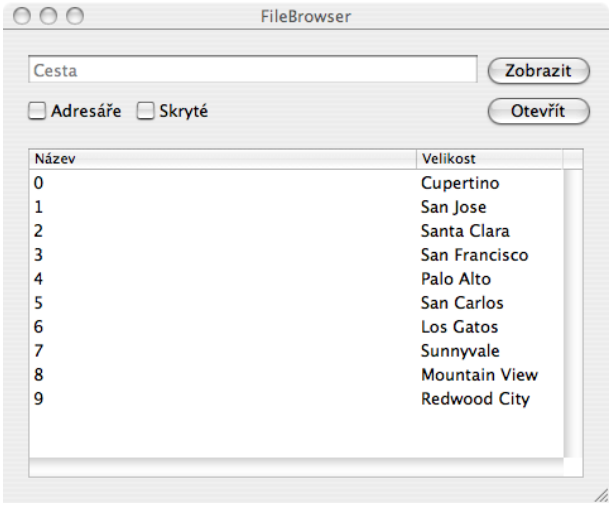
Protože hlavní kontroler aplikace je zároveň zdrojem dat pro tabulku, musí implementovat dvě metody:

```
- (int)numberOfRowsInTableView:(NSTableView *)aTableView;
- (id)tableView:(NSTableView *)aTableView
     objectValueForTableColumn:(NSTableColumn *)aTableColumn
            row:(int)rowIndex;
```

Metody, které musí programátor implementovat, pokud vytváří zdroj nebo delegáta pro již hovovou třídu, by měl zjistit z dokumentace.

3.6.4 Vytvoření okna aplikace

Hlavní okno aplikace je již vyžvořeno, ale neobsahuje žádné komponenty kromě menu. Poklepáním v hlavním okně Xcode na soubor MainMenu.nib [3] se otevře Interface Builder a v něm hlavní okno vytvářené aplikace. Hlavní okno je prázdné prázdné a v samostatném okně by melo zobrazit menu obsahující základní operace automaticky napojené na objekt First Responder a tím je zajištěno, že je programátor nemusí programovat. Mezi ně patří vypnutí aplikace, zobrazení okna o aplikaci, cut copy paste operace, kontrola pravopisu a další funkce, které Cocoa nabízí.



Obrázek 3.10: Rozmístění komponent aplikace

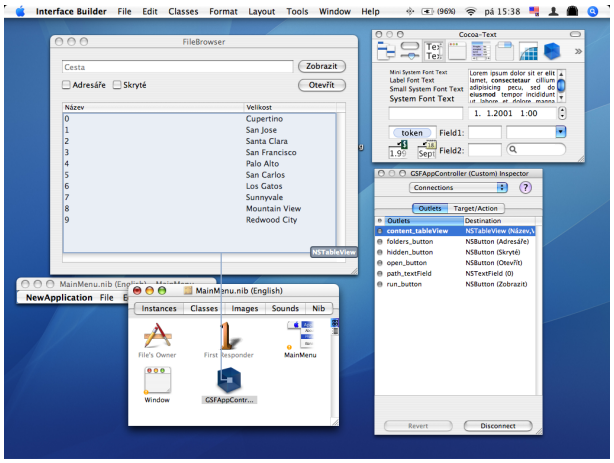
Z palety komponent, umístím na plochu ovládací prvky podle obrázku 3.10. Interface Builder umisťované komponenty přichytává ve vzdálenostech od ostatních tak, jak definují Apple Human Interface Guidelines. Během několika chvil tak lze vytvořit uživatelské rozhraní, které se neliší od ostatních aplikací.

Aby se komponenty chovaly korektně i při změně velikosti okna, je potřeba jim definovat takzvaný *Autosizing*, který se nachází ná záložce Size v Inspektoru. Na této záložce se nachází kříž, jehož ramena určují svou vnější částí, který pozice je vůči okraji okna pohyvlivý, a svou vnitřní částí označí, který rozměr komponenty je relativní vůči rozměru okna. Rovná část kříže znamená, že vzdálenost respektive rozměr, je v daném směru neměnná. Grafická reprezentace zase značí dynamiku, vzhledem k vybranému rozměru. Pokud chci, aby tlačítko *Zobrazit* bylo vždy stejně vzdáleno od pravého okraje, a textové pole se drželo okraje levého a měnilo svou velikost stejně jako okno, „vytvořím pružinu“ ve vodorovné a spodní části kříže u textového pole a v levé a dolní části u tlačítka *Zobrazit*. U ostatních komponent je chování analogické. Pro správné pochopení funkčnosti je nejlepší si vyzkoušet chování přímo v náhledu vznikajícího okna v Interface Builderu. V menu `File` vyberu `Test Interface` a změnou velikosti okna si ověřím zda jsem nastavil *Autosizing* správně u všech komponent. Zpět do Interface Builderu se vrátím pomocí položky `Quit NewApplication` v menu `Interface Builder`. Samozřejmě lze využít i jejich klávesové zkratky.

Dalším krokem je propojení komponent na kontrolér aplikace. Nejprve musím Interface Builderu oznámit existenci třídy `GSFAppController`. Z Xcode přetáhnu hlavičkový soubor této třídy do Interface Builderu. V okně nazvaném *MainMenu.nib* by s měla zobrazit záložka Classes a vybraná právě vložená třída. V kontextovém menu této třídy vyberu Instantiate a přepnu na záložku

[3] nachází se v části stromu Resources

Instances. Tam by se měla zobrazit modrá kostka reprezentující instanci mého kontroléru.



Obrázek 3.11: Propojení komponent s objektem kontroléru GSFAppController

Nyní propojím Outlety kontroléru s komponentami grafického rozhraní. Za držení klávesy *ctrl* táhnu myší od instance `GSFAppController` postupně jednotlivým komponentám. Měla by se objevit modrá linka (viz obrázek 3.11, která znázorňuje projojení. Na záložce inspektoru objektu `GSFAppController` vyberu odpovídající Outlet dané komponenty a kliknu na tlačítko „Connect“. Tím se propojí proměnná třídy s konkrétním objektem.

Stejným postupem vytvořím propojení akcí. Od tlačítka táhnu myší ke kontroléru, v inspektoru vyberu záložku Target/Action a zvolím akci, se kterou chci danou komponentu propojit. Pro tlačítko „Zobrazit“ to je akce `showContent:` a pro „Otevřít“ vyberu `open:`. Tak se určí, jaká zpráva se má kontroléru odeslat při stisknutí tlačítka.

Tímto práce v Interface Builderu nekončí, protože ještě musím nastavit zdroj dat pro tabulku zobrazující soubory a složky. To se dělá úplně stejným způsobem jako propojení Outletů. Vyberu `NSTableView` několikerým kliknutím na tabulku. Kliknout se musí vícekrát, protože nejprve se vybere `NSScrollView`, který automaticky vytváří horizontální i vertikální posuvníky. Pak propojím tuto tabulku s instancí objektu `GSFAppController` a v inspektoru na záložce Outlets vyberu `dataSource` a kliknu na tlačítko „Connect“. Druhým cílem zpráv, který může tabulka vyžadovat je `delegate`, ale v tomto příkladě není zapotřebí. Pro určení sloupců při vyžádání dat musím definovat jejich názvy. Tyto názvy jsou textové a definují se na první záložce inspektoru nazvané Attributes. Stejným způsobem jako k tabulce se musím doklikat až na `NSTableColumn`, který se zvýrazní zešednutím. Na záložce vyplním položku `Identifier` a první sloupec nazvu `column_name` a druhý `column_size`. Tyto názvy jsou důležité, protože podle nich kontrolér zjišťuje sloupec, pro který tabulka požaduje data. U každého sloupce vyplním jeho text v hlavičce `Header Title` a ručně nastavím jeho šířku.

Lze také vypnout možnost editace buď pro jednotlivé sloupce nebo pro celou tabulku. Vzhledem k tomu, že není implementovaná metoda pro editaci tabulky, nebude mít případné editování za běhu žádný význam. Tím prozatím končí práce v IB a přesunu se zpět do Project Builderu.

3.6.5 Vývoj vrstvy Model

Třída GSFInode

Nyní si vytvořím dvě třídy, které patří do vrstvy Model. První z nich je `GSFFile`, se kterým se v aplikaci pracuje. Místo samotné třídy by stačilo vytvořit pouze strukturu, která bude efektivnější, rychlejší a úspornější, ale pro účely ukázky použiji třídu. Hlavičkový soubor je v ukázce 2.

Ukázka zdrojového kódu 2 GSFFile.h

```
#import <Cocoa/Cocoa.h>


@interface GSFFile : NSObject {
    NSString *name;
    unsigned long long size;
    BOOL isHidden;
    BOOL isFolder;
}

- (void)setName:(NSString *)aName;
- (unsigned long long)size;
- (void)setSize:(unsigned long long)aSize;
- (BOOL)isHidden;
- (void)setHidden:(BOOL)hidden;
- (BOOL)isFolder;
- (void)setFolder:(BOOL)folder;
@end
```

Proměnné uchovávají pouze obsah, který se předává na výstup. Pravě z tohoto důvodu by mohla být výhodněji použita nějaká struktura, ale na této třídě ukáži později velmi silný nástroj nazvaný Key-Value coding (KVC). Vnímavému čtenáři určitě neuniklo, že jsou uvedeny všechny metody pro přístup k proměnným kromě čtení `name`. K této proměnné se bude přistupovat právě pomocí KVC.

Implementace třídy je velmi jednoduchá viz ukázka 3. Pouze u metody `setName:` stojí za zmínku, přestože řetězce `NSString` bývají zpravidla v autoreleasepool a není potřeba se o ně starat, při implementaci metody je vhodné si zajistit, aby je garbage collector nevhodně nezrušil. To se zajistí odesláním metody `retain` řetězci, který se přiřazuje. Nahrazovanému řetězci se odešle metoda `release` a tím se mu oznámí, že již není potřeba (sníží se mu retainCount o jedna). Nakonec se provede předání ukazatele na původní řetězec, který se v paměti nekopíruje. Ostatní metody jsou pouhé přiřazení hodnot, které se od klasického C nijak neliší, protože se jedná o skalární proměnné.

Názvy proměnných odpovídají parametrům souboru. Name, size, isHidden obsahují přesně to co se očekává od jejich názvu. Proměnná isFolder nabývá hodnoty YES v případě, že se jedná o složku, v opačném případě má hodnotu NO. Velikost souboru musí byt `unsigned long long`, aby byla zajištěna dostatečně velká celočíselná reprezentace pro soubory, které lze v současné době vytvářet.

Třída GSFFileManager

Druhou třídou z vrstvy *Model* je `GSFFileManager`, který zajišťuje přístup aplikace do vňejšího světa souborů. Přesněji řečeno pro tento přístup využívá frameworku Cocoa, který aplikaci odstiňuje přímo od souborového systému a zajišťuje tak přenositelnost aplikace. Tato přenositelnost je sice omezená frameworkem Cocoa, ale odstiňuje programátora od znalostí souborového systému.

Ukázka zdrojového kódu 3 GSFFile.m

```
#import "GSFFile.h"

@implementation GSFFile

- (void)setName:(NSString *)aName
{
    [aName retain];
    [name release];
    name = aName;
}

- (unsigned long long)size
{
    return size;
}
- (void)setSize:(unsigned long long)aSize
{
    size = aSize;
}
- (BOOL)isHidden
{
    return isHidden;
}
- (void)setHidden:(BOOL)hidden
{
    isHidden = hidden;
}
- (BOOL)isFolder
{
    return isFolder;
}
- (void)setFolder:(BOOL)folder
{
    isFolder = folder;
}
@end
```

Stejným způsobem jako třídu `GSFAppController` vytvořím nový soubor typu Objective-C class s názvem `GSFFileManager.m` včetně hlavičkového souboru, který je zobrazen v ukázce 4.

Ukázka zdrojového kódu 4 GSFFileManager.h

```
#import <Cocoa/Cocoa.h>
#import "GSFFile.h"


@interface GSFFileManager : NSObject {
    NSFileManager *fileManager;
    NSMutableArray *fileCache;

    NSString *path;
    BOOL hidden;
    BOOL folders;
}

-(id)init;
-(NSArray *)filesOnPath:(NSString *)aPath
             showHidden:(BOOL)showHidden
            showFolders:(BOOL)showFolders;

@end
```

Tato třída obsahuje pouze instanci třídy `NSFileManager` s názvem `fileManager`, která zajistí přístup k souborům, a důležitý objekt `fileCache`, který je instancí třídy `NSMutableArray`, a tři pomocné proměnné, jejichž význam vysvetlím záhy. Tato třída umožňuje načtení obsahu adresáře, jehož obsah filtruje podle zadaných parametrů (požadovaná cesta, zobrazovat také složky, zobrazovat skryté položky). Při každé změně vstupních parametrů dochází k načtění souborů a složek na zadané cestě a pokud vyhoví podmínkám filtru, jsou uloženy do dynamického pole `fileCache`. V každém dalším přístupu s nezměněnými parametry je vrácen pouze ukazatel na již vytvořené pole a nemusí se znovu data načítat. Abych mohl evidovat zda se parametry změnily, musím si uchovat jejich původní hodnotu, která se právě ukládá do posledních tří proměnných. Při jejich změně se hodnota aktualizuje.

Třída `GSFFileManager` implementuje (viz ukázka 5) jedinou vlastní metodu

`filesOnPath:showHidden:showFolders`

a přetěžuje konstruktor `init` a metodu `dealloc`, která se volá při odstraňování objektu z paměti. V kontruktoru je vytvořena instance pole pro ukládání souborů a také `NSFileManager`, který se používá pro práci se soubory. Zajimavější je metoda `dealloc`, ve které dochází k uvolnění prostředku, které byly zabrány. Zpráva `release` musí být polána všem objektům, kterým předtím posláno `retain`. Zpráva `retain` se odesílá také při inicializaci pomocí metody `init` nebo při vytvoření zasláním zprávy `copy`. V ostatních případech je objekt umístěn do autorelease pool a nemusí být ručně odstraňován.

Nakonec destruktoru se musí odeslat zpráva rodiči (`[super dealloc]`), aby také dokončil uvolňování prostředků z paměti. Pokud programátor zapomene tuto zprávu odeslat, překladač ho na to upozorní varováním.

Třída implementuje jednu metodu, která vykonává veškerý kód. Jejími parametry jsou cesta a zda jsou požadovány složky a skryté položky. Nejprve se provede kontrola, zda došlo od posledního požadavku ke změně. Pokud ano, uloží se nové nastavení a provede se načtení souborů z adresáře. K tomu slouží `NSFileManager`, kterému se odešle zpráva `directoryContentsAtPath:` a vrácenou hodnotou je pole souborů a složek.

Ukázka zdrojového kódu 5 GSFFileManager.m

```
@implementation GSFFileManager
- (id)init
{
    fileManager = [NSFileManager defaultManager];
    fileCache = [[NSMutableArray alloc] init];
    return self;
}

- (void)dealloc
{
    [fileCache release];
    [path release];
    [super dealloc];
}

- (NSArray *)filesOnPath:(NSString *)aPath showHidden:(BOOL)showHidden showFolders:(BOOL)showFolders
{
    if( aPath == nil ) return nil;

    if( path == nil || showHidden != hidden || showFolders != folders || ![aPath isEqual:path] ){
        //nahrazeni novou cestou
        [path release];
        path = [aPath copy];
        hidden = showHidden;
        folders = showFolders;

        [fileCache release]; // odstraneni cache
        fileCache = [[NSMutableArray alloc] init]; // vytvoreni nove

        // nacteni obsahu slozky do pole
        NSArray *tempArray = [fileManager directoryContentsAtPath:path];
        // enumerator slouzi k~prochazeni dat
        NSEnumerator *enumerator = [tempArray objectEnumerator];
        NSString *fileName; // aktualne prochazene jmeno souboru
        NSString *filePath; // cesta prochazeneho souboru
        NSDictionary *attributes; // atributy prochazeneho souboru
        GSFFile *file; // objekt souboru ukladany do cache

        while( fileName = [enumerator nextObject] ){
            // vytvoreni cesty z~puvodni cesty a nazvu
            filePath = [path stringByAppendingPathComponent:fileName];
            // nacteni atributu souboru
            attributes = [fileManager fileAttributesAtPath:filePath
                                              traverseLink:NO];

            file = [[GSFFile alloc] init]; // vytvoreni objektu virtualniho souboru
            [file setName:fileName];
            [file setSize:[[attributes valueForKey:NSFileSize] unsignedLongLongValue]];
            [file setHidden:([fileName characterAtIndex:0] == '.')];
            [file setFolder:([attributes valueForKey:NSFileType] == NSFileTypeDirectory)];

            // vlozeni obejktu do pole pokud vyhovi podminkam
            if( (showHidden || ![file isHidden]) &&
                (showFolders || ![file isFolder]) ){
                [fileCache addObject:file]; // timto se zvysi retainCount
            }
            [file release]; // odstraneni puvodniho objektu (zustane uchovan v~poli)
        }
    }
    return fileCache; // predani cache na vystup
}
@end
```

Toto pole se prochází pomocí enumerátoru. Každá datová struktura je schopna vrátit enumerátor pomocí zprávy `objectEnumerator`, který usnadňuje procházení. Každý další objekt datové struktury (v mém případě `NSArray`) se získá odesláním zprávy `nextObject`. Pokud v datové struktuře již žádné objekty nejsou, je vrácena hodnota `nil` a podmínka v cyklu while je NO.

V tělě cyklu se provede nastavení hodnot `GSFFile` a pokud tento objekt vyhoví nastaveným podmínkám, je vložen do pole pro uchování předávaných souborů. Objekty do pole se vkládají pomocí zprávy `addObject:`. Pokud je objekt takto předán, je mu automaticky odeslána zpráva `retain`. Po vložení do pole je tedy retainCount na pomocný objekt `file` roven dvěma. Pokud vytvořil novou proměnnou pomocí kontruktoru `init` po odstranění pole by tento objekt zůstal v paměti. Protože pomocnou proměnnou již nepotřebuji, odešlu ji po vložení do pole zprávu `release` a tím vrátím retainCount na jedna. Až se bude rušit celé pole, budou s ním odstraněny také všechny objekty.

Po ukončení cyklu se ukazatel na pole předá na výstup a dále se z něj čte. Není předpoklad, že by se toto pole dále upravovalo, proto je jako návratový typ uveden `NSArray` a programátora, který využíva této třídy nemusí zajímat jak pole vzniklo. Protože je `NSMutableArray` potomkem třídy `NSArray` může být předáno tam, kde se očekává jeho rodič. Pokud bych chtěl zajistit jeho neměnnost i dále, mohu vytvořit nové pole pomocí `[NSArray arrayWithArray:]` a tím získat nové statické pole.

3.6.6 Implementace třídy GSFAppController

Nejvíce programování si vyžádá kontrolér celé aplikace. Základní kostra objektu je v ukázce 6. K ní budou později přidány dvě metody, které musí obsahovat napojený dataSource pro tabulku třídy `NSTableView`, a implementovány dvě metody pro akce (jejich návratová hodnota je `IBAction`. Konstruktor i destruktor byly popsany v předchozí kapitole. V této třídě se objevuje nová metoda `awakeFromNib`. Jak jsem uvedl, instance kontroléru je vytvořena v době návrhu a je uložena do souboru s koncovkou nib. Po načtení ze souboru je tomuto objektu odeslána zpráva `awakeFromNib`. Velmi častým zdrojem chyb je napsání kódu do konstruktoru místo do metody `awakeFromNib`.

Kdybych v ukázce 6 vložil odeslání zprávy

```
[path_textField setStringValue:[@"~" stringByExpandingTildeInPath]];
```

do konstruktoru, došlo by při způštění aplikace k chybě, protože konstruktor by se zavolal ještě před načtením z nib souboru, kdy ještě není známé propojení s grafickým rozhraním.

Nyní implementuji metody, které vyžaduje `NSTableView` od svého dataSource (viz 7). První metoda vrací počet řádku, které má tabulka zobrazit. Tento počet je roven počtu zobrazovaných souborů z adresáře, proto vrátím přesně velikost pole souborů. Zajímavější je metoda pro získání dat. Tabulka odesílá zprávu, ve které je obsažen ukazatel na tuto tabulku (to lze využít pokud jeden kontrolér obsluhuje více tabulek), ukazatel na sloupec a nakonec číslo řádku. Úkolem přogramátora je z této zprávy analyzovat, kterou buňku tabulka požaduje, obstarat pro ni data a předat na ně ukazatel. Zde je opět vidět síla Objective-C umožňující předávat ukazatel na obecný objekt, který může být obrázek, řetězec, formátovaný řetězec nebo libovolná jina data.

V kapitole 3.6.4 jsem si v IB pojmenoval sloupce abych je mohl nyní jednoduše identifikovat. Předanému sloupci odešlu zprávu `identifier` vracející jeho název, který porovnám s textovým řetězcem, abych zjistil, o který sloupec se jedná. V případě, že se jedná o první sloupec vrátím jen ukazatel na název souboru, který se nachází na pozici požadovaného řádku. K získání ukazatele na objekt na konkrétní pozici slouží zpráva `objectAtIndex:`.

Abych zjistil název souboru využili silného nástroje Cocoa nazvaného *Key-Value coding*. To

Ukázka zdrojového kódu 6 GSFAppController.m

```
@implementation GSFAppController

- (id)init
{
    fileManager = [[GSFFileManager alloc] init];

    return self;
}

- (void)dealloc
{
    [fileManager release];
    [super dealloc];
}

- (void)awakeFromNib
{
    [path_textField setStringValue:[@"~" stringByExpandingTildeInPath]];
    actualPath = [NSString stringWithString:[path_textField stringValue]];
}

- (IBAction)showContent:(id)sender
{
}

- (IBAction)open:(id)sender
{
}
@end
```

umožňuje získat přístup k proměnným, aniž by pro ně existovala veřejná metoda [4]. `NSObject` definuje dve velmi užitečné metody pro každý objekt - metodu pro čtení a metodu pro nastavení proměnné pomocí jejího jména:

```
- (id)valueForKey:(NSString *)attrName;
- (void)setValue:(id)newValue forKey:(NSString *)attrName;
```

Metoda `valueForKey:` umožňuje číst hodnotu proměnné pomocí jména. Samozřejmě lze přistoupit přes přístupovou metodu nebo, pokud neexistuje, číst proměnnou přímo. Řekněme, že odešlu zprávu `valueForKey:@"name"` objektu `GSFFile`. Pokud by měl metodu `name`, bude spustěna tato metoda a vrácena její hodnota. Protože však metodu `name` nemá, hledá se po instanční proměnné s názvem name. Pokud proměnná existuje, je vrácena její hodnota. Není-li nalezena ani metoda ani promnná, bude ohlášena chyba. Metoda `setValue:forKey` funguje podobně.

Tento mechanizmus umožňuje přístup ke všem atributům objektu. Díky této technice lze plikace jednoduče zpřístupnit AppleScriptu. Key-value coding také umožňuje existenci `NSControlleru` a bindovacího mechanizmu. Jejich popis by byl nad rámec diplomové práce a zájemce odkazuji na literaturu [Hil05].

Nyní zbývá implementovat metody, které se volají při stisknutí tlačítek „Zobrazit“ a „Otevřít“. Metoda `showContent:` se volá při stisknutí tlačítka prvního a nastaví aktuální cestu z textového pole, přečte zadání filtru a oznámí tabulce, že je potřeba aktualizovat data. Programátor nikdy ne-

[4] Objective-C nerozlisuje privátní a veřejné metody. Všechhny metody uvedené v hlavičkovém souboru jsou veřejné - public a jsou uvedeny v generované dokumentaci. Pokud chce programátor metodu skrýt, neuvede ji do hlavičkového souboru. Aby se vyhnul varování při překladu, vytvoří kategorii pro danou třídu a v ní uveden metody, které mají být skryté

Ukázka zdrojového kódu 7 Implementace metod pro NSTableView ve třídě GSFAppController.m

```
- (int)numberOfRowsInTableView:(NSTableView *)aTableView
{

    return [[fileManager filesOnPath:actualPath
                          showHidden:showHidden
                         showFolders:showFolders] count];
}

- (id)tableView:(NSTableView *)aTableView
     objectValueForTableColumn:(NSTableColumn *)aTableColumn
            row:(int)rowIndex
{
    id object;
    GSFFile *file = [[fileManager filesOnPath:actualPath
                                   showHidden:showHidden
                                  showFolders:showFolders] objectAtIndex:rowIndex];

    if( [[aTableColumn identifier] isEqual:@"column_name"] ){
        object = [file valueForKey:@"name"];
    }
    else if( [[aTableColumn identifier] isEqual:@"column_size"] ){
        if( [file isFolder] ){
            object = @"";
        }
        else {
            object = [[NSNumber numberWithUnsignedLongLong:[file size]] stringValue];
        }
    }
    return object;
}
```

posílá nová data, ale pouze oznámí jejich změnu a vyčkává až si o ně tabulka řekne. Ta totiž nejlépe ví, které řádky potřebuje překreslit.

Druhá metoda se volá při stisknutí druhého tlačítka a způsobí otevření aktuálně vybrané položky. Pokud není žádná položka vybrána, má proměnná `row` hodnotu -1 a v metodě se nepokračuje. V opačném případě dojde k načtení aktuálního souboru a na základě zda se jedná o soubor nebo složku se vykoná akce.

Jedná-li se o složku, připojí se její název na konec aktuální cesty a zavolá se stejná akce jako v případě kliknutí na tlačítko „Zobrazit“. Otevířá-li uživatel soubor, využije se služeb `NSWorkspace` a odešle se zpráva `openFile` s cestou k souboru a Cocoa zajistí otevření souboru ve správné aplikaci.

Ukázka zdrojového kódu 8 Implementace akcí IBAction ve třídě GSFAppController.m

```
- (IBAction)showContent:(id)sender
{
    actualPath = [path_textField stringValue];
    showHidden = [hidden_button state] == NSOnState;
    showFolders = [folders_button state] == NSOnState;
    [content_tableView reloadData];
}
- (IBAction)open:(id)sender
{
    int row = [content_tableView selectedRow];
    if( row >= 0 ){
        GSFFile *file = [[fileManager filesOnPath:actualPath
                                       showHidden:showHidden
                                      showFolders:showFolders] objectAtIndex:row];
        NSString *path = [actualPath stringByAppendingPathComponent:
            [file valueForKey:@"name"]];

        if( [file isFolder] ){
            [path_textField setStringValue:path];
            [path_textField setNeedsDisplay];
            [self showContent:nil];
        }
        else {
            [[NSWorkspace sharedWorkspace] openFile:path];
        }
    }
}
```

3.6.7 Dokončení aplikace

Pokud nyní aplikaci zkompiluji a spustím, bude vybrána domovská složka, protože při spučtění metody `awakeFromNib` dojde k nastavení na a s využitím třídy `NSString` k rozvinutí na plnou cestu. Změnit adresář o úroveň níž lze vybráním požadovaného adresáře a kliknutím na tlačítko „Otevřít“. Nahoru lze přejít odmazáním cesty v textovém poli.

Chce-li uživatel změnit nastavení filtru, musí nejprve provést změnu zaškrtávacích tlačítek a pak potvrdit zobrazení. Na to nejsou lidé z mac komunity zvyklí, proto je potřeba ještě aplikaci dotvořit. Přejdu do Interface Builderu a tlačítkům „Adresře“ a „Skryté“ nastavím akci stejnou jako pro „Zobrazit“. Akce se provede při změně stavu zaškrtávacího pole. Stačí uložit nib soubor a v Project Builderu spustit aplikaci [5]. Teď dochází k aktualizaci obsahu adresáře i při změně nastavení filtru.

Ještě je vhodné aby aplikace začínala s aktivním vstupním polem pro zadání cesty. Opět v Interface Builderu vyberu v okne nib souboru objekt Window a propojím ho s textovým polem a

[5] Za povšimnutí stojí, že nedochází ke kompilování, protože změny se ukládají do nib souboru.

na záložce Oulet propojím s `initialFirstResponder`. Tím se zajistí, že textové pole bude při spuštění aplikace jako první očekávat vstup od uživatele. Pokud chci aby mohl uživatel pomocí klávasy `tab` skákat pouze po textovém poli a tlačítkách „Adresře“ a „Skryté“, navzájem je propojím jako Outlet nazvaný `nextKeyView`. Při stisknutí `tab` dojde k přeskočení na objekt, který odpovída propojení s `nextKeyView`.

Tato ukázka je velmi vzdálená některé z použitelných aplikací, ale dobře posloužila k ukázání některých programovacích technik. Podrobný popis a výklad programování aplikací pro Mac OS X vydá na nejednu obsáhlou knihu a takový prostor mi tato práce nenabízí. Výše popsaný program je velmi odlehčenou verzí souborového manažera, který je hlavní náplní mé diplomové práce a poskytuje tak názornější výklad dané problematiky.

Kapitola 4

Implementace souborového manažera

4.1 Koncepce a myšlenky

Cílem této práce je vytvořit souborový manažer určený výhradně pro operační systém Mac OS X, aby svým ovládáním i vzhledem odpovídal ostatním aplikacím určeným pro tento systém. Správce souborů má dva vertikálně oddělené panely, které zobrazují nezávisle na sobě obsah složek. Aktivní je vždy jeden panel, který slouží jako zdrojový, zatímco druhý je zpravidla cíl operací. Měnit zvolenou složku lze vždy jen v aktivním panelu. Panel se aktivuje buď kliknutím myši nebo stisknutím tlačítka *tab*, kterým se pokaždé vybere druhý panel. Akce mezi panely se provádí buď pomocí operací v menu, z nichž mají ty nejpoužívanější své ekvivalenty jako klávesové zkratky, nebo tažením myši. Aplikace umožňuje základní souborové operace a slouží jako základ pro pokročilý souborový manažer.

Souborový manažer je navržen tak, aby všechny operace byly odstíněny od fyzické práce se soubory. Tím je umožněno implementovat libovolnou abstraktní vrstvu, která poskytuje přístup k určitým datům, které mohou být reprezentovány jako soubory. Tyto data mohou představovat soubory na FTP serveru, knihovnu hudebních souborů na iPodu, SVN repository na vzdáleném počítači a další. Pro tyto účely je navržen protokol, který popisuje jaké metody musí objekt poskytující soubory implementovat a je na programátorovi tohoto modulu, aby zajistil potřebná data pro reprezentaci souborů. Pro účely diplomové práce jsem vytvořil modul, který poskytuje přístup k souborům na lokálním disku.

4.1.1 Rozdělení do vrstev

Tato aplikace je třívrstvá a využívá konceptu *Model – View – Control* viz strana 42. Části aplikace jsou rozděleny do tří logických celků, které jsou na sobě nezávislé. V aplikacích, které vznikají s využitím frameworku Cocoa, je tento koncept nejpoužívanější, protože umožňuje rozdělení do tří nezávislých částí, které lze udržovat i vyvíjet samostatně, které lépe odpovídají lidskému chápání aplikace.

4.1.2 Vrstva Model

Nejnižší vrstvu, která obstarává data pro aplikaci, zastupují třídy poskytující abstraktní souborovou reprezentaci, a tedy implementují protokol `GSFModuleProtocol` viz ukázka 9 na straně 68. Tento protokol definuje všechny potřebné metody pro přístup k souborům a základním operacím s nimi.

Protokol slouží jako předloha pro nově vznikající třídu, aby měl programátor usnadněnou práci a nezapomněl implementovat důležitou metodu, která poskytuje potřebná data. Pokud není některá

Ukázka zdrojového kódu 9 Ukázka použití protokolu ve třídě pro práci se soubory

```
@protocol GSFModuleProtocol
- (NSMutableArray *)filesOnPathByString:(NSString *)aPath
                     ignoreUpDirectory:(BOOL)ignoreUpDirectory;
- (NSMutableArray *)fileTreeOnPathByString:(NSString *)aPath;
- (GSFInode *)getFileFromPath:(NSString *)aPath;

- (BOOL)createFolder:(NSString *)aFolder
             onPath:(NSString*)aPath;
- (BOOL)renameOnPath:(NSString *)oldFile
             toPath:(NSString *)newFile;
- (int)filePermission:(NSString *)aFile;
- (BOOL)setFilePermission:(NSString *)aFile
             permissions:(int)aPerm;

- (BOOL)deleteFile:(NSString *)filePath;
- (BOOL)canMoveFileToTrash;
- (BOOL)moveFileToTrash:(NSString *)filePath;

- (unsigned long long)availableSpaceAtPath:(NSString *)filePath;

- (NSInputStream *)loadFileStreamFromPath:(NSString *)
                    aPath WithController:(id) controller
                                InRunLoop:(NSRunLoop *)aLoop;
- (NSOutputStream *)storeFileStreamToPath:(NSString *) aPath
                           WithController:(id) controller
                                InRunLoop:(NSRunLoop *)aLoop;
- (NSString *) lastErrorMessage;

- (BOOL)fileExistsAtPath:(NSString *)path;
- (BOOL)folderExistsAtPath:(NSString *)path;
- (BOOL)fileExistsAtPath:(NSString *)path
           canOverwrite:(BOOL *)canOverwrite;

- (BOOL) preparePathForFile:(NSString *) aPath;

- (NSArray *)pathCompletition:(NSString *)path;
- (NSArray *)pathCompletition:(NSString *)path
               caseSensitive:(BOOL)flag
                 filterTypes:(NSArray *)filterTypes;
@end
```

metoda definovaná v protokolu implementována, překladač na to upozorní, ale překlad se provede. Protokol tedy slouží jako pomůcka, která je užitečná obzvlášť při práci v týmu. Použití ve třídě se definuje uvedením v ostrých závorkách viz ukázka 10 na straně 69.

Ukázka zdrojového kódu 10 Definice protokolu pro práci se soubory

```
@interface GSFLocalFiles : NSObject < GSFModuleProtocol > {
.
.
.
}
@end
```

Nejčastěji používanou metodou je `filesOnPathByString:ignoreUpDirectory:`, která poskytuje seznam souborů na aktuální cestě uvedené v prvním parametru. Tato cesta je virtuální pro vrstvu kontrolérů a musí být přeložena. To je záležitostí třídy, která soubory poskytuje, a pro třídu, která je používá to musí být zcela transparentní. Cesta je vnitřně přeložena ve všech metodách, které požadují nějakou akci na zadané cestě. Třída, která poskytuje soubory z fyzického disku, na druhou stranu nic nepřekládá, protože tyto cesty jsou ekvivalentní. Druhým parametrem je přepínač, kterým se oznámí, zda je potřebná virtuální složka .. pro přechod do složky, která se nachází o úroveň výš. Návratovou hodnotou této metody je `NSMutableArray`, které obsahuje všechny soubory a složky na dané cestě, které nejsou nijak specificky seřazeny. To je záležitostí té třídy, která data požaduje. Z důvodu jednoduchého a rychlého řazení je použita třída `NSMutableArray`. Při řazení nad statickým polem, dochází k vytvoření nového pole se seřazenými ukazateli do pole původního a objektům je zvýšen počet ukazatelů o jedna. Bohužel se mi nepodařilo efektivně vyřešit, aby se původní pole zrušilo a neodstranilo všechny objekty. Řešením pro mě bylo použití dynamického pole, které při řazení pouze změní pořadí ukazatelů na objekty a nezvyšuje počet ukazatelů na ně. Položky v tomto poli jsou výhradně `GSFInode`.

Další metodou je `fileTreeOnPathByString:`, která poskytuje strom souborů a složek od zadané cesty. Strom je navázaný v objektech třídy `GSFInode`, které jsou typu složka. Tyto objekty mohou obsahovat pole dalších `GSFInode`. Tento podstrom lze přiřadit každému objektu této třídy, ale pouze u typu složka má smysl. Data se získávají pomocí metody pro načtení aktuálního adresáře, která je pokaždé volána s aktuální cestou načítané složky. Tato metoda se používá pří získávání stromu pro kopírování a přesouvání, proto není potřeba virtuální složka pro přechod o úroveň výše.

Velmi úzce spolu souvisí metody

```
- (NSInputStream *)loadFileStreamFromPath:(NSString *) aPath
                           WithController:(id) controller
                                InRunLoop:(NSRunLoop *)aLoop;

- (NSOutputStream *)storeFileStreamToPath:(NSString *) aPath
                           WithController:(id) controller
                                InRunLoop:(NSRunLoop *)aLoop;
```

poskytující přístup k souborů na zadané cestě pomocí proudu *stream*. Tento přístup je potřebný k úplnému odstínění vyšších vrstev od fyzického přístupu k souborům. Ty pak nepřistupují přímo k souboru, ale pouze čtou/zapisují z/do příslušného proudu. Tento proud je vytvořený na základě virtuální cesty, kterou musí objekt poskytující soubory správně reprezentovat. Prvním parametrem je cesta, na které by se měl nacházet požadovaný soubor. Druhý parametr je kontrolér, který přijímá zprávy generované datovým proudem. Poslední parametr je `NSRunLoop` ve kterém daný proud

pracuje. Tyto dva parametry jsou vyžadovány konstruktorem třídy `NSStream`, jejímž potomky jsou použité třídy. Třídy `NSInputStream` a `NSOutputStream` generují události, které je potřeba někde zpracovat a proto se předává objekt `controller`, který bude tyto události zachytávat a řešit jejich zpracování. Kopírování souborů je tedy zcela odstíněno od fyzického přístupu k nim. Tyto proudy lze využít nejen při kopírování a přesouvání, ale také při načítání interním editorem či prohlížečem, které by mohly být v budoucnu implementovány.

Další metody uvedu pouze ve stručném přehledu, protože jejich použití odpovídá jejich názvům a na jejich implementaci není nic zvláštního:

- `getFileFromPath:` tato metoda vrátí jeden virtuální soubor na zadané cestě. Tento soubor je objekt třídy `GSFInode`
- `createFolder:onPath:` vytvoří novou složku na zadané cestě a vrátí YES pokud se akce zdařila. V opačném případě vrátí NO.
- `renameOnPath:toPath:` přejmenuje složku nebo soubor na zadané cestě na cestu novou a vrátí YES pokud se akce zdařila. V opačném případě vrátí NO.
- `filePermission:` přečte práva souboru nebo složky na zadané cestě. Tato metoda není příliš potřebná, protože práva jsou obsaženy přímo ve virtuální reprezentaci.
- `setFilePermission:` nastaví práva souboru nebo složky na zadané cestě a vrátí YES pokud se akce zdařila. V opačném případě vrátí NO.
- `deleteFile:` zcela odstraní soubor nebo složku na zadané cestě a vrátí YES pokud se akce zdařila. V opačném případě vrátí NO.
- `moveFileToTrash:` přesune soubor nebo složku na zadané cestě do koše a vrátí YES pokud se akce zdařila. V opačném případě vrátí NO.
- `canMoveToTrash:` vrátí YES pokud daný modul umí přesouvat do koše. V opačném případě vrátí NO. Například modul pro práci s FTP nebude mít implementováno přesouvání do koše a proto vrátí NO.
- `availableSpaceAtPath` vrátí velikost volného prostoru na zadané cestě v bytech.
- `fileExistsAtPath:` vrátí YES pokud je na požadované cestě soubor. V ostatních případech vrací NO.
- `folderExistsAtPath:` vrátí YES pokud je na požadované cestě složka. V ostatních případech vrací NO.
- `fileExistsAtPath:canOverwrite:` vrátí YES pokud existuje na požadované cestě soubor. V ostatních případech vrací NO. V druhém parametru je odkazem předána hodnota YES pokud lze soubor změnit, jinak je vráceno NO.
- `preparePathForFile:` vytvoří cestu pro soubor, pokud neexistuje. Tato metoda vytvoří celou cestu až po požadovanou úroveň a vrátí YES pokud se akce zdařila. V opačném případě vrátí NO.

Nakonec přece jen uvedu dvě zajímavé metody. Jsou to

```
- (NSArray *)pathCompletition:(NSString *)path;
- (NSArray *)pathCompletition:(NSString *)path
                caseSensitive:(BOOL)flag
                  filterTypes:(NSArray *)filterTypes;
```

Tyto metody poskytují `NSArray`, které obsahuje názvy souborů odpovídající názvům souborů a složek, jenž začínají poslední částí cesty. Bude-li předána cesta „/Users/Ondra/Documents/prac“, budou vráceny všechny soubory a složky z `/Users/Ondra/Documents` a `prac`. Nejsou-li takové soubory k dispozici bude vrácen `nil`. Využití je při automatickém doplňování cesty při jejím zadávání. Část pro poskytování seznamu souborů v modulu pro práci se soubory na disku je hotová, ale její využití ve vyšších vrstvách je nedokončené, protože vyžaduje vytvoření nové komponenty, která bude nabízet tento seznam aktualizovaný při každém stisknutí klávesy. Rozšířená verze umožňuje definovat kromě cesty také citlivost na velikost písmen a filtr, podle kterého mlze být omezený seznam souborů na konkrétní typy souborů. Pokud bude například nadřazený objekt potřebovat pouze složky, uvede to v posledním parametru a budou mu předány pouze složky. To může být relevantní třeba při změně cesty, kdy jsou názvy souborů nepotřebné. Zkrácená verze metody volá druhou metodu v základním nastavení.

Návratové hodnoty, které poskytují metody definované protokolem `GSFModuleProtocol` jsou buď booleovské hodnoty či případně číslo reprezentující velikost, ale především to jsou objekty třídy `GSFInode` buď samostatně nebo obsažené v nějakém poli. Tato třída je součástí vrstvy *Model*, protože je jejím výstupem a reprezentuje data, která zpracovává vyšší vrstva *Control*, a bude popsána v kapitole 4.2.3 na straně 74.

Pokud budou nově vytvářené třídy pro různé moduly poskytující přístup k různým zařízením a vzdáleným serverům implementovat metody definované protokolem `GSFModuleProtocol` budou transparentně přístupné všem aktivitám v souborovém manažeru.

4.1.3 Vrstva View

Do této vrstvy lze zahrnout všechny třídy, které poskytují grafické uživatelské rozhraní. Sem patří kontroléry oken. To jsou třídy, které se starají o uživatelské akce a volají služby poskytované vrstvou *Control*. Tyto kontroléry jsou určeny vždy pro jedno okno, kterému poskytují akce, na které se napojí jednotlivé komponenty v Interface Builderu. Okna jsem si rozdělil na dvě základní skupiny: Windows a Dialogs.

První skupina Windows obsahuje ta okna, která jsou schopna „samostatného života“ a zobrazují nějakou informaci v samostatném nezávislém okně. Sem patří `GSFMainWindow`, které je hlavním oknem aplikace. Může se vyskytovat ve více instancích. Další okno, které sem patří, je `GSFFileInformationWindow` zobrazující informace o určité složce, souboru nebo jejich skupině.

Druhá skupina se nazývá Dialogs a obsahuje okna, která slouží k nějaké interakci s uživatelem. Jsou rozdělena na ty, která mohou být spuštěna jako takzvaný *sheet*. Tak se nazývají okna, která slouží jako dialogová a vysouvají se z horní lišty okna. Tato okna zamezí pokračování nějaké akce, dokud nejsou uzavřena. Pro tyto účely jsem si vytvořil speciální třídu `GSFCustomDialog`, která poskytuje vyžaduje tlačítka `OK` a `Cancel`, uchovává informaci o delegátovi a kontextový objekt a implementuje metody pro otevření a zavření okna. Okna této třídy lze zobrazit buď jako modální okna [1] nebo jako sheet. Tato třída je pak využita většinou dialogových oken, které tak lze zobrazit oběma způsoby. V současné implementaci se využívá pouze druhého způsobu a dialogy se zobrazují pouze jako výsuvné panely.

[1] Okna, která odeberou přístup oknu, ze kterého pochází, dokud nejsou uzavřena.

4.1.4 Vrstva Control

Mezi vrstvami *Model* a *View* se nachází zprostředkující vrstva *Control*. Sem patří všechny objekty, které zajišťují práci s virtuálními soubory, zpracovávají akce zadané uživatelem a jsou zodpovědné za chod aplikace. Mezi nejdůležitější patří třídy

- `GSFAppController`
- `GSFPanelsController`
- `GSFActionController`,
- `GSFCopyThread`
- `GSFPanelController`
- `GSFHistoryController`

Těmto třídám bude věnovaný podrobný popis v samostatných kapitolách. Ostatní třídy jsou pomocné.

4.2 Implementace částí

4.2.1 Více jazyčnost

Je standardem, že aplikace pro platformu macintosh jsou vícejazyčné. Pro vývoj jsem zvolil jako výchozí jazyk angličtinu, protože je přecejenom ve světě počítačů nejrozšířenější. Jako další v budoucnu přibude samozřejmě čeština. Framework Cocoa a samotný Xcode problémy lokalizace programu velmi usnadňují.

Tato problematika se dá rozdělit na nekolik částí:

- lokalizace grafického uživatelského rozhraní
- lokalizace řetězců obsažených v kódu aplikace
- lokalizace ostatních prvků - nápověda, obrázky atd.

Velmi stručně shrnu první dvě části lokalizace. Pro počeštění uživatelského rozhraní je potřeba v potřeba lokalizovat všechny `nib` soubory. Pravým tlačítkem se vyvolá info a lokalizacích se přidá potřebný jazyk. To vytvoří nový adresář `Czech.proj`, do kterého se budou ukládat všechny věci týkající se češtiny. Tento adresář je na počátku kopií původního `English.lproj`.

Ve skupině *R*esources v hlavním okně Xcode se objeví dvě verze lokalizovaného `nib` souboru. Otevřením české verze lze začít s lokalizací. To, že jsou vytvořeny dvě kopie je důležité, protože některá tlačítka mohou svým rozměrem být pro některá slova z nového jazyka nevhodná. Takto lze upravit vzhled GUI aby aplikace vypadala stále skvěle.

Pro lokalizaci řetězců se musí vytvořit soubor s koncovkou `.strings`, do kterého se umisťují překlady do daného jazyka.

```
"Klic1" = "Hodnota1";
"Klic2" = "Hodnota2";
```

Pro nalezení správné hodnoty pro zadaný klíč se používá `NSBundle`:

```
NSBundle *main;
NSString *aString;

main = [NSBundle mainBundle];
aString = [main localizedStringForKey:@"Klic1"
                                value:@"Vychozi hodnota"
                                table:@"Find"];
```

To nalezne hodnotu pro „Klic1“ ve Find.strings souboru. Pokud není nalezený první preferovaný jazyk bude pouzit druhý preferovaný atd. Pokud není nalezena žádná z hodnot, je vrácena „Vychozi hodnota“. Pokud není poskytován soubor se zadaným jménem, je použit soubor `Localizable`. Nejjednodušší aplikace budou obsahovat pro každý jazyk pouze soubor `Localizable.strings`.

Jazyk je v aplikaci automaticky volen podle výchozího nastavení systému v *S*ystem preferences v sekci *I*nternational . Volitelně lze preferovaný jazyk pro konkrétní jazyk nastavit přímo ve Finderu v informacích o souboru. Souborový manažer jsem prozatím lokalizovat nestihl, protože je pod intenzivním vývojem a jeho části se velmi rychle mění. Udržování více jazykových verzí by v této chvíli bylo zbytečné.

4.2.2 Abstraktní souborová vrstva

Třídy nejnižší datové vrstvy *Model* (viz strana 67) slouží pro fyzický přístup k souborům. Odstiňují vyšším vrstvám práci se soubory a umožňují implementaci nejrůznějších služeb (například FTP). Tuto vrstvu si lze představit jako abstraktní, neboť aplikační vrstva *Control* neví jak se soubory ve skutečnosti pracuje a pouze odesílá zprávy objektům z této vrstvy. Tento přístup usnadňuje implementaci, protože stačí vytvářet jen nové moduly v datové vrstvě a přidat jejich ovládání (například vytvořit nové FTP připojení), a není potřeba upravovat základní konstrukce programu.

Návrh takové struktury byl poměrně náročný, protože aby nemusela být aplikace při vytváření nového modulu měněna, musel být návrh vytvořen již od začátku univerzálně. Vytvoření takového protokolu, který by byl co nejjednodušší, kvůli snadnému použití, a přitom poskytoval vše potřebné vyžadovalo velmi dobré znalosti používaného frameworku, které jsem na začátku vývoje, kdy se provádí hlavně návrh aplikace, bohužel neměl. Získával jsem je v až v průběhu programování a studia dalších částí frameworku Cocoa a jazyka Objective-C. Z tohoto důvodu docházelo k mnoha změnám v návrhu, protože nově nabyté informace umožňovaly čistější postupy a efektivnější provádění operací. To vedlo v jednom případě až k úplnému přepsání obou vrstev *Model* a *Control*, tak aby mohly lépe spolupracovat a přesto být na sobě nezávislé. Negativní stránkou toho postupné pronikání je značné zpoždění vyvíjené aplikace, která tak není v současné době dotažena do podoby, která by mohla být prezentována veřejnosti, ale naplňuje zadání diplomové práce. Posloužila však především jako velmi vhodný studijní materiál, protože díky své složitosti a komplexnosti si vyžádala nastudování mnoha programovací technik a částí frameworku Cocoa.

Abstraktní souborová vrstva je tvořena virtuálními soubory, které poskytuje třída `GSFInode` popsaná v kapitole 4.2.3. Tyto soubory jsou poskytovány třídami, které přistupují řeší fyzický přístup k souborům. V současné verzi je implementována pouze jedna taková třída `GSFLocalFiles` jejíž popis naleznete v kapitole 4.2.4. V kódu lze také nalézt první pokusy o vytvoření modulu pro přístup na FTP servery. Vývoj je však pouze teoretický, protože ze mi prozatím nepodařilo vyřešit, jaké prostředky použít pro přístup na servery. V Javě jsou velmi dobře navrženy datové proudy, které lze velmi jednoduše použít i pro vytvoření FTP serveru či klienta, v Objective-C však existuje pouze třída `NSStream` a její dva potomci `NSInputStream` a `NSOutputStream`, které se mi pro tyto účely využít prozatím nepodařilo. Zkoumám proto možnosti využití nějakého frameworku napsaného v Javě případně jak správně použít proudy v Cocoa. FTP klient by byl nad rámec diplomové

práce, proto jsem jeho vývoj pozastavil.

4.2.3 Virtuální soubory GSFInode

Abstraktní souborová vrstva implementuje třídu `GSFInode`, která představuje objekt, se kterým aplikace pracuje. Hlavička této třídy je v ukázce zdrojového kódu 11 na straně 74. Při jejím návrhu jsem se inspiroval třídou `NSFile` a částečně převzal způsob práce s atributy.

Ukázka zdrojového kódu 11 Hlavička třídy GSFInode

```
@interface GSFInode : NSObject {
    NSString *name;
    NSString *path;
    NSMutableDictionary * attributes;
    NSArray *subdirectories; // only if it is Folder
    NSImage *fileIcon;
    NSString *theApplication;
    NSString *theType;
    id controller;
    BOOL isUpFolder;
}

+(NSString *)stringSizeValue:(unsigned long long)aSize;
+(NSString *)stringSizeValue:(unsigned long long)aSize
                        step:(unsigned int)step;
- (id)init;
- (id)initWithController:(id)aController;
- (void)dealloc;
- (NSString *)name;
- (void)setName:(NSString *)aName;
- (NSString *)path;
- (void)setPath:(NSString *)aPath;
- (NSImage *)fileIcon;
- (void)setFileIcon:(NSImage *)aFileIcon;
- (NSString *)theApplication;
- (void)setApplication:(NSString *)anApplication;
- (NSString *)type;
- (void)setTheType:(NSString *)aType;
- (void)setAttributes:(NSMutableDictionary  *) anAttributes;
- (NSDictionary *)attributes;
- (BOOL)isFolder;
- (BOOL)isFile;
- (BOOL)isSymLink;
- (BOOL)isSocket;
- (BOOL)isCharacterSpecialFile;
- (BOOL)isBlockSpecialFile;
- (NSString *)fileType;
- (BOOL)isUpFolder;
- (BOOL)isHidden;
- (void) setFileSize:(unsigned long long)aSize;
- (unsigned long long)fileSize;
- (void)setSubdirectories:(NSArray *)anArray;
- (NSArray *)subdirectories;
- (BOOL)isEqual:(id)anObject;
@end
```

Třída `GSFInode` představuje virtuální soubor nebo složku, se kterými se provádí veškeré operace. Tento soubor může být umístěn fyzicky na disku, může se jednat o soubor na vzdáleném serveru nebo složka může představovat skupinu podsložka album a soubory jednotlivé písničky

alba, které se načítají z knihovny iPodu [2].

Třída `GSFInode` je přímým potomkem `NSObject`. Obsahuje proměnné `name`, která reprezentuje název objektu, `path`, která ukládá cestu a `attributes`, ve které jsou uloženy atributy souboru. Tyto atributy jsou odvozeny od atributů, jenž jsou poskytovány objektem třídy `NSFileManager`. Atributy souboru jsou uloženy v `NSDictionary` a popisují POSIXové vlastnosti souboru. Tyto vlastnosti jsou uvedeny pod klíči:

```
extern NSString *NSFileType;
extern NSString *NSFileSize;
extern NSString *NSFileModificationDate;
extern NSString *NSFileReferenceCount;
extern NSString *NSFileDeviceIdentifier;
extern NSString *NSFileOwnerAccountName;
extern NSString *NSFileGroupOwnerAccountName;
extern NSString *NSFilePosixPermissions;
extern NSString *NSFileSystemFileNumber;
extern NSString *NSFileExtensionHidden;
extern NSString *NSFileHFSCreatorCode;
extern NSString *NSFileHFSTypeCode;
extern NSString *NSFileImmutable;
extern NSString *NSFileAppendOnly;
extern NSString *NSFileCreationDate;
extern NSString *NSFileOwnerAccountID;
extern NSString *NSFileGroupOwnerAccountID;
```

K těmto atributům lze přistupovat pomocí metody `attributes`, která vrátí `NSDictionary`, ve kterém jsou tyto atributy uloženy, nebo lze využít některých metod, které transparentně zjistí aktuální hodnotu. Tyto metody jsou výhodnější při velmi častých operacích, jako je zjištění o jaký typ souboru se jedná `fileType`, jaké má jméno `name`, případně zda se nejedná o virtuální složku sloužící k přesunu o úroveň výše `isUpFolder` a nebo zjištění velikosti souboru `fileSize`. Obdobným způsobem lze atributy souboru nastavit. První písmeno se zvětší na velké a před název metody se přidá slůvko `set`. To neplatí u metody `isUpFolder`, která nemá svůj ekvivalent pro nastavení, protože tato vlastnost se definuje pouze při vzniku a nelze ji v průběhu měnit.

Další proměnnou, kterou musím zmínit, je `NSArray *subdirectories`, která má smysl pouze pokud je objekt složka. Může obsahovat obsah složky (její podsložky a soubory), ale pouze v případě explicitního vyžádání. Za normálních okolností je toto pole prázdné a pouze v případě potřeby dojde k načtení. Používá se to při kopírování souborů, kdy je potřeba znát všechny soubory, které se budou kopírovat, protože jedině tak se dá zjistit velikost dat, které se budou kopírovat. Stejně tak je potřeba znát všechny cesty, protože k souboru se přistupuje vždy pomocí úplné cesty a jen tak jej mohou třídy z vrstvy `Model` bezpečně identifikovat. Aby nemusela být tato cesta zjišťována opakovaně, ukládají se do pole `subdirectories` virtuální reprezentace souborů, které mohou být opět složky a opět uchovávat svůj obsah atd.

Velmi zajímavé jsou dvě proměnné, které ve třídě přibyly jako poslední a jsou to `NSImage *fileIcon` respektive `NSString *theApplication`, které uchovávají ikonu souboru nebo složky respektive název aplikace, kterou je soubor otevírán. Tyto informace jsou nastaveny buď ihned při vzniku virtuálního souboru nebo efektivněji až v době potřeby. Původní implementace nastavovala ikonu i aplikaci při načítání souboru z disku, ale to vedlo k velmi pomalému načítání souborů. Z toho

[2] To je jeden z plánovaných modulů do budoucna, měl by poskytovat transparentní přístup pro nahrávání písniček do iPodu a hlavně zpět, protože to iTunes neumožňují.

důvodu jsou ikony zjišťovány při zobrazování souboru a aplikace až při otevírání souboru, jsou však uchovávány při pozdější opětovné použití. Další plánované zrychlení, je pomocí přesunutí načítání ikon do samostatného vlákna, které bude nezávislé na zobrazování souborů a ikony se zobrazí až, když se je podaří načíst. Uvedený postup bohužel s sebou nese komplikace v podobě správného ukončení vlákna, pokud ještě nebylo dokončeno načítání a uživatel již změnil zobrazovanou složku. Tyto optimalizace jsou v současnosti na předním místě, protože rychlost zobrazování obsahu složky je prioritou aplikace.

Nakonec musím zmínit tyto metody:

- `isFolder`
- `isFile`
- `isSymLink`
- `isSocket`
- `isCharacterSpecialFile`
- `isBlockSpecialFile`
- `isHidden`

Kromě poslední uvedené metody, slouží k rychlému zjištění zda virtuální soubor odpovídá požadovanému typu. V implementované verzi jsou využity pouze první dva. Ostatní typy neumí aplikace zpracovat. Poslední metoda `isHidden` slouží k pohotovému načtení, zda je soubor skrytý. Skryté soubory se v Unixových aplikacích uvozují tečkou. Zjištění, zda je na prvním místě řetězce názvu je pomalejší než vrácení booleovské hodnoty, proto se provede vyhodnocení jen jednou při vytváření objektu, přesněji řečeno při definici jména. Při všech požadavcích je vrácena hodnota uložená v proměnné `hidden`.

Prozatím se mi nepovedlo efektivně využít barevné štítky, které lze vytvářet ve Finderu a podle kterých lze velmi rychle hledat jak vizuálně, tak je využít jako parametr hledání. Pro jejich implementaci je potřeba využít funkce z frameworku Carbon a s tím jsem se ještě dostatečně neseznámil. Jeho použití je nad rámec zadání diplomové práce, přesto bych je velmi rád co nejdříve začlenil do programu, protože práce s nimi je velmi příjemná.

4.2.4 Fyzický přístup k souborům pomocí GSFLocalFiles

Třída `GSFLocalFiles` slouží pro práci se soubory na lokálním disku. V současné verzi nepřistupuje přímo k souborům, ale využívá objekty a služby, které nabízí framework Cocoa. Tento frameworku umožňuje velmi pohodlnou práci se soubory bohužel za cenu nižšího výkonu, protože práce s objekty má poměrně velkou režii, a proto není jejich využití zcela ideální. Poskytují však robustní základ, jak začít se soubory pracovat a získávat potřebná data. Časem lze tyto ryze objektivní prostředky nahradit nízkoúrovňovými funkcemi, které by mohly být rychlejší a efektivnější. Na druhou stranu řeší framework Cocoa veškeré záležitosti týkající se přístupu k disku za programátora.

Na straně 67 je popsán protokol `GSFModuleProtocol`, který musí implementovat každá třída, která chce vystupovat v systému jako modul pro přístup k souborům. Třída `GSFModuleProtocol` je prozatím jedinou třídou, která tento protokol používá. Momentálně poskytuje všechna potřebná data s využitím frameworku Cocoa, který poskytuje třídy `NSFileManager` a `NSWorkspace`. Třída `NSFileManager` umožňuje provádět generické souborové operace a odděluje aplikaci od souborového systému. Základní metodou je metoda třídy `+ defafultManager`, která vrátí instanci objektu `NSFileManager`, se kterým se dále pracuje. Tato třída poskytuje všechny základní souborové

operace jako je vytvoření adresáře, odstranění souboru, přesun souboru nebo získání informací o souboru či složce. Umí také soubory kopírovat, avšak při žádné z operací neposkytuje informace o průběhu akce, ale pouze vyvolá událost při zahájení akce (ať už je to kopírování nebo mazání), při jejím dokončení a nebo při chybě. Modul pro přístup k lokálním souborům umožňuje operace, které mají jediný cíl: vytvoření adresáře, smazání, nastavení práva atd., a jednosměrný přístup k souboru. Třída nemůže být schopná provést kopírování nebo přesun, protože zdroj a cíl se mohou nacházet v jiných modulech a je na vyšších vrstvách aby zajistily správný transfer souborů mezi nimi. Výjimkou je přesun souborů do koše, protože na tuto operaci aplikace pohlíží jako na operaci s jedním cílem, kterým jsou soubory přesouvající se do koše.

Většina operací je velmi jednoduchých, protože při obdržení zprávy na nějakou operaci se tato pouze deleguje na `NSFileManager`, který se o vše potřebné postará. Výjimkou je přesun souborů do koše, který si vyžádá několik akcí a využívá také funkce z Core Foundation. Ty jsou uvozeny prefixem CF. Ukázka metody, která přesouvá soubory do koše je v ukázce zdrojového kódu 12 na straně 78. Nejprve je potřeba zjistit cestu ke složce, která představuje koš na daném diskovém oddílu, protože každý oddíl má svůj koš, aby nedocházelo k přesunům mezi fyzickým oddíly, protože tato operace se rovná kopírování a je časově velmi náročná. Teprve po získání cesty se provede akce přesun souborů, kterou zajistí `NSFileManager`. Při chybě vrátí metoda NO a v případě úspěchu YES. Důležité je přetypování řetězce z Core Foundation `trashPath`, který je typu `CFStringRef`, na řetězec `NSString`, který nabízí Cocoa.

Přesun souborů do koše není možný v každém modulu, protože ne každý může poskytovat službu koše. Příkladem může být opět modul pro přístup na FTP server, jehož protokol už ve své podstatě použití koše vylučuje. Dokonce ale i v modulu na místním disku existují situace, kdy nelze soubory přesunout do koše. Pokud má uživatel otevřenu složku, která je přípojným bodem samba sdílení, také nelze provést akci přesunutí do koše. Pro tyto účely je definována metoda `canMoveFileToTrash`, která vrací YES nebo NO v závislosti na tom, zda je možno soubory do koše přesunout. V současné implementaci `GSFLocalFiles` je vždy vráceno YES a vyřešení zda je pro danou cestu možné využít služeb koše je teprve v pořadí.

Další metodou, která obsahuje víc než jen odeslání jedné zprávy instanci třídy `NSFileManager` je `preparePathForFile:`. Tato metoda provádí otestování od jaké úrovně se zadaná cesta nachází a chybějící část podstromu vytvoří. Je potřeba mít na paměti, že v jednotlivých úsecích cesty může již existovat adresář se stejným názvem nebo nic. Oba případy jsou korektní, protože první ušetří práci s vytvářením a druhý způsobí vytvoření nového adresáře. Složitější situace nastává v případě, že některá ze složek cesty je souborem libovolného typu (tím jsou myšleny symbolické linky, roury atd.), kdy se musí aplikace rozhodnout jak se zachová. V případě, že se jedná o soubor nebo rouru, dochází jednoznačně k chybě, protože nelze vytvořit adresář se stejným názvem jako má soubor. Pakliže stojí v cestě symbolický link, musí být rozhodnuto, zda ohlásit chybu, že nelze adresář s daným názvem vytvořit nebo využít symbolického linku jako adresáře a pokračovat v cestě dále. Protože tento souborový manažer prozatím neumí pracovat s odkazy, bude ohlášena chyba.

V popisu uvedeném o několik odstavců výše byla zmíněna velice zajímavá a uživatelsky příjemná metoda `pathCompletition: caseSensitive: filterTypes:`, která umožňuje automatické doplňování cesty při psaní. K vytvoření této metody jsem se inspiroval rozšířením třídy `NSString`, které jsem nalezl v dokumentaci. Modul `GSFLocalFiles` tuto metodu implementuje a využívá rozšíření třídy `NSString`, která pomocí kategorie získala schopnosti práce s cestami a jednou z nich je nalezení všech odpovídajících cest z předaného řetězce. Touto metodou je:

```
- (unsigned)completePathIntoString:(NSString **)outputName
                     caseSensitive:(BOOL)flag
                  matchesIntoArray:(NSArray **)outputArray
                       filterTypes:(NSArray *)filterTypes
```

Ukázka zdrojového kódu 12 Metoda pro přesun souborů do koše

```
- (BOOL)moveFileToTrash:(NSString *)filePath
{
    CFURLRef        trashURL;
    FSRef           trashFolderRef;
    CFStringRef     trashPath;
    OSErr           err;

    err = FSFindFolder(kUserDomain,
                       kTrashFolderType,
                       kDontCreateFolder,
                       &trashFolderRef);

    if (err == noErr) {
        trashURL = CFURLCreateFromFSRef(kCFAllocatorSystemDefault,
                                        &trashFolderRef);
        if (trashURL) {
            trashPath = CFURLCopyFileSystemPath (trashURL,
                                                 kCFURLPOSIXPathStyle);
            if (![fileManager movePath:filePath
                               toPath:[(NSString *)trashPath
                                         stringByAppendingPathComponent:
                                           [filePath lastPathComponent]
                                      ]
                              handler:nil])
                return NO;
            else{
               [[NSWorkspace sharedWorkspace]
                 noteFileSystemChanged:(NSString *)trashPath];
            }
        }
        else {
            return NO;
        }
        if (trashPath) {
            CFRelease(trashPath);
        }
        CFRelease(trashURL);
    }

    return YES;
}
```

Pokud je kandidát pouze jeden, je předaný v prvním parametru, je-li jich více, jsou uloženi do pole předaném v parametru posledním. Počet kandidátů na doplnění cesty je předán v návratové hodnotě metody.

Třída `NSString` poskytuje nepřeberné množství metod, které usnadňují práci s řetězci jako s cestou. Nejpoužívanějšími metodami jsou:

- `fileSystemRepresentation`
- `isAbsolutePath`
- `lastPathComponent`
- `pathExtension`
- `stringByAbbreviatingWithTildeInPath`
- `stringByAppendingPathComponent:`
- `stringByAppendingPathExtension:`
- `stringByDeletingLastPathComponent`
- `stringByDeletingPathExtension`
- `stringByExpandingTildeInPath`
- `stringByStandardizingPath`

Metody mají velmi výstižné názvy, které přesně popisují co daná metoda provádí, a aplikují se přímo na řetězec, který chce programátor upravit. Kromě metody `isAbsolutePath`, která vrací booleovskou hodnotu, je jejích výstupem řetězec s požadovanou úpravou. Nechť existuje řetězec `path`, který obsahuje nějakou cestu k souboru. Odesláním zprávy `lastPathComponent` bude vytvořen nový řetězec, který obsahuje název souboru včetně přípony. Příklad použití je v ukázce zdrojového kódu 13.

Ukázka zdrojového kódu 13 Získání nazvu souboru ze zadané cesty

```
NSString * path = @"/Users/Ondra/Document/diplomova prace.pdf";
NSString * filename = [path lastPathComponent];
NSLog(@"Nazev souboru je \"%@\"", filename);
```

Na výstupu bychom měli obdržet:

```
2006-11-10 13:06:48.736 test[20368] Nazev souboru je "diplomova prace.pdf"
```

Třída `NSString` poskytuje metod pro práci s řetězci jako cestami mnohem více, všechny je lze nalézt i s popisem v dokumentaci frameworku Cocoa. Výše uvedené metody hojně využívám pro veškerou práci s cestami. Velká výhoda pramení z toho, že se jedná o kategorii třídy `NSString`, jedná se tedy o rozšíření práce s řetězci a cesty se nemusí při vypisování do komponent, které očekávají řetězec, nijak přetypovávat nebo speciálně upravovat. S cestami se pracuje nejen v modulu pro práci se soubory, ale také ve vrstvě *Control*, kde se sice pracuje s virtuální cestou, ale ta se v principu od fyzické cesty nijak neliší.

4.2.5 Zpracování souborů

Spolupráce vrstev aplikace

Jak již bylo mnohokrát řečeno, aplikace je rozdělena do tří vrstev. Tyto vrstvy musí mezi sebou komunikovat a jsou velmi těsně provázány. Na obrázku 4.1 je znázorněno jak se navzájem jednotlivé třídy využívají. V centru dění se nachází `GSFPanelsController`, který obstarává ovládání obou panelů a zajišťuje pro ně soubory. Tento objekt obsahuje dva objekty `GSFPanelController`, které každý představují řídící jednotku pro své protějšky v podobě `GSFTableView`, které přímo vykreslují soubory na obrazovku. Propojení mezi řídící a zobrazovací jednotkou kontroluje právě `GSFPanelsController`, který se nachází v každém okně, a lze například prohodit oba panely pouhým přehozením ukazatelů. Toto oddělení umožňuje také nezávislost vrstvy *View* a *Control*, protože vše je prováděno přes jednoho prostředníka, a pokud bude potřeba v budoucnu zcela vyměnit jednu z částí nebude muset být přepisováno velké množství kódu.

Instance třídy `GSFPanelsController` zajišťuje zpracování předaných událostí z klávesnice a myši, které se týkají zobrazování souborů a složek. Jsou to kliknutí na komponenty zobrazující soubory a všechny vstupy z klávesnice, kromě těch, které jsou zpracovány jako zkratky příkazů v menu. Jejím hlavním posláním je sloužit jako zdroj dat pro tabulky, které zobrazují soubory. Ty vyžadují aby jim byl v době návrhu přiřazen `dataSource`, kterému delegují požadavky na zobrazované soubory. Popis tohoto principu a komponenty pro zobrazení souborů je v kapitole 4.2.7 na straně 88. Ve skutečnosti není přímým zdrojem, ale podle toho, která z tabulek požaduje data, sáhne do příslušného panelu a vyžádá si soubory pro zobrazení, které předá jako odpověď na dotaz tabulce.

Kromě této činnosti také ovládá aktuální pozici v jednotlivých panelech, předává informace o tom, že mají být označeny některé soubory atd. K tomu jsou vytvořeny metody začínající na „moveCursor“, které zajistí, aby kurzor byl posunován pouze v povolených mezích – 0 až počet souborů ve složce. Ostatní možnosti by způsobily vynucený pád aplikace přístupem do nedefinované části pole. Metodou, která také pracuje s kurzorem, je `actualCursorPositionInTable`. Tato metoda na základě předané tabulky zjistí, aktuální pozici na které se nachází kurzor. Pozice kurzoru je uložena přímo v kontroléru panelu `GSFPanelController` nikoliv v `GSFPanelsController`.

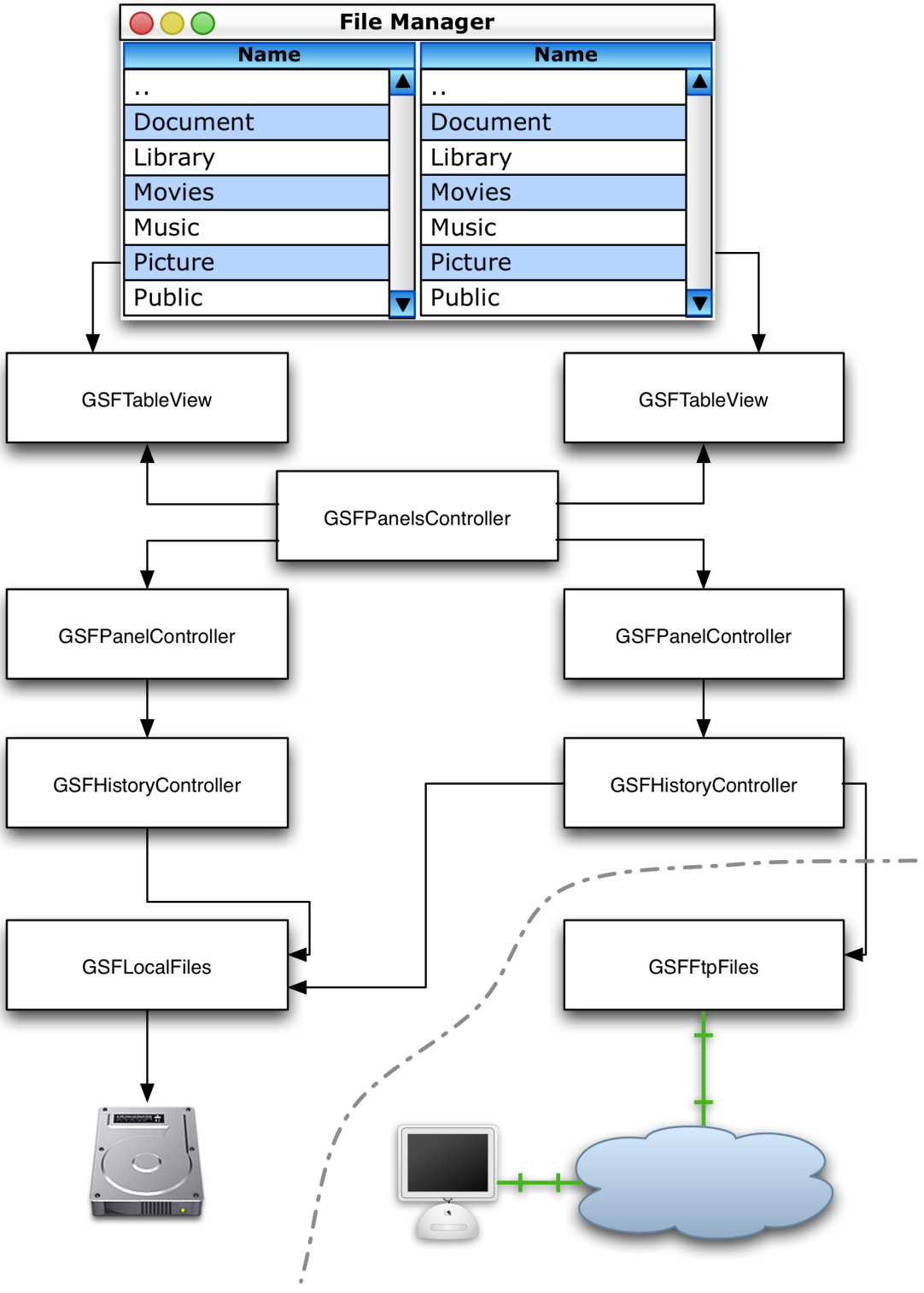
Velmi důležitým úkolem, který `GSFPanelsController` také vykonává, je zajištění provedení nějaké akce nad souborem či složkou, nad kterou se nachází kurzor [3]. Provedení operace zajišťuje metoda `makeAction`. V ukázce zdrojového kódu 14 lze vidět, že tato metoda neočekává žádný parametr a pracuje vždy s aktuálním panelem a aktuálně vybraným souborem. Implementovány jsou celkem tři typy, kterými může otevíraná položka být:

1. aplikace
2. složka
3. soubor

Tyto typy by měli zvládnout zpracovat všechny dostupné soubory. Pokud aplikace narazí na typ kterému nerozumí, neprovede se žádná akce a tato skutečnost je oznámena do logu. Explicitně nejsou prozatím zpracovávány symbolické odkazy a pokud se jedná o odkaz na složku, bude otevřena ve Finderu. Správné využití symbolických odkazů je plánováno velmi brzy.

V ukázce metody `makeAction` je využito služeb Workspace, které je poskytováno prostřednictvím instance třídy `NSWorkspace`. Workspace umožňuje mimo jiné právě otevíraní souborů jim

[3] V tom se aplikace liší od Finderu, který akci provádí nad všemi vybranými (označeným) soubory a kurzor vůbec nemá.



Obrázek 4.1: Spolupráce tříd při zobrazování souborů. Část oddělená šedou čerchovanou čarou není v aplikaci implementována a je uvedena pro představu

Ukázka zdrojového kódu 14 Metoda `makeAction` zajišťuje provedení akce nad aktuálně vybraným souborem

```
- (void)makeAction
{
    unsigned int row = [activePanel actualCursorPosition];
    GSFInode * inode = [activePanel returnFileOnRow:row];

    if ( [[inode type] isEqual:@"app"] == YES){
        [[NSWorkspace sharedWorkspace] launchApplication:
                                                    [inode application]];
    }
    else if ( [inode isFolder] ){
        [self changeDirectoryToPathOnRow:row inPanel:activePanel];
    }
    else if ( [[inode application] isNotEqualTo:@""] ){
        [[NSWorkspace sharedWorkspace] openFile:[inode path]];
    }
    else {
        NSLog(@"Neznamy typ.");
    }
}
```

v systému definovaných aplikacích, případně otevírat samotné aplikace. Dříve jsem používal systémového volání `system("open " + _nazev_souboru);` vykonávajícího stejnou činnost, ale použití `NSWorkspace` je v Cocoa aplikaci vhodnější a elegantnější.

O otevření složky se postará samotný objekt `GSFPanelsController`, který zpracuje nově zadanou cestu, ověří zda existuje, uloží ji jako aktuální pracovní cestu a předá ji aktivnímu panelu aby si ji aktualizoval a načetl nové soubory. V této metodě je také odeslána zpráva tabulce paritní k aktivnímu panelu, aby dostala informaci o tom, že se změnila aktuální data.

`GSFPanelsController` je tedy prostředníkem mezi objekty, které zajišťují vykreslování na obrazovku, a objekty, které se starají o dodání správných dat. Zpracovává tedy všechny události, které se týkají aktuálně zobrazovaných souborů.

Načítání souborů

O načítání souborů se starají objekty třídy `GSFPanelController`. Soubory se předávají vždy pro aktuální adresář a jsou uloženy v poli `NSArray`, které se předává vždy seřazené tak, jak se bude vypisovat na obrazovku. Pro přístup k souborům slouží několik metod:

```
- (NSArray *)returnFiles;
- (NSArray *)returnSelectedFiles;
- (NSArray *)returnSelectedFilesPaths;
- (GSFInode *)returnFileOnRow:(unsigned int)row;
```

Tyto metody vrací objekty tak jak je uvedeno v jejich názvu. Pro všechny operace by stačila první metoda, která vrátí ukazatel na pole se všemi soubory, ale pro jednodušší použití v aplikaci jsou vytvořeny metody, které poskytují přímý přístup k požadovaným objektům Například tabulka při vykreslování chce vždy jeden soubor na konkrétním řádku, proto je pro ni implementována tato funkce, která se využívá i jinde.

Výchozím bodem je tedy metoda `returnFiles`, která získává data již přímo z modulu pro přístup k souborům. Tento modul je některou z tříd z vrstvy *Model* například `GSFLocalFiles`. Modul i aktuální cesta je uložena v pomocném objektu `GSFHistoryController`, který má na

starosti ukládání aktuálních cest, aby je bylo možno využít při navigaci. Pro potřeby ovládání je to zcela transparentní a s objektem `GSFPanelController` se pracuje tím způsobem, že se mu předá nová cesta a očekává se, že bude vracet aktuální data.

Protože k souborům se přistupuje velmi často, například jen tabulka, která je vykresluje, si je vyžádá poprvé když potřebuje vědět kolik objektů bude vykreslovat a potom si řekne o každý ze souborů samostatně, jsou tyto soubory při prvním načtení uloženy do pole, se kterým se pak dále pracuje. Pole je obnoveno vždy, když dojde ke změně aktuální cesty nebo je o to panel explicitně požádán. Při výstupu je pole souborů seřazeno a filtrováno podle aktivního filtru. Tato operace se samozřejmě děje také jen jednou a opakuje se, pokud dojde ke změně filtru nebo způsobu řazení, případně pokud se změní data. Dojde-li ke změně filtrování, nejsou data znovu načítána ale filtr se aplikuje na již načtená data.

Prozatím se mi nepodařilo vyřešit, jak zajistit aby se data automaticky obnovila pokud se změní obsah nějaké složky. `NSWorkspace` sice poskytuje prostředky, kterými se oznámí systému, že došlo ke změně v souborovém systému, bohužel se mi však tuto funkci nepovedlo zprovoznit aby reagovala na přidaný soubor pomocí Finderu nebo uložený jinou aplikací. Systém notifikací funguje pouze v rámci aplikace.

Pro řazení souborů se využívá funkce, která je implementována přímo v Cocoa pro řazení pole. Každé pole `NSArray` poskytuje metody:

- `sortedArrayHint`
- `sortedArrayUsingFunction:context:`
- `sortedArrayUsingFunction:context:hint:`
- `sortedArrayUsingDescriptors:`
- `sortedArrayUsingSelector:`

Tyto metody vrací ukazatel na pole seřazen pole. Pro účely řazení se využívá prostředků, které lze odvodit z názvu. Prvním způsobem je řazení s využitím nějaké funkce. V prvním parametru zprávy je předán ukazatel na funkci, která bude rozhodovat o pořadí prvků. Parametr hint je ukazatel na objekt `NSData`, který se využívá jako úložiště výsledků při řazení a umožňuje rychlejší řazení pokud došlo v poli jen k malým změnám. Funkce musí mít hlavičku `(int (*)(id, id, void *))`. První dva parametry jsou objekty které se porovnávají a třetí je ukazatel na libovolnou strukturu, ve které mohou být například uloženy parametry nastavení třídění. Funkce provede porovnání prvního a druhého prvku a vrátí jednu z hodnot: `NSOrderedDescending`, `NSOrderedAscending` nebo `NSOrderedSame`. Ukázka 15 je funkcí, která se používá pro řazení souborů podle názvu. Je z ní patrně, že řazení souborů nemusí být triviální záležitostí, protože se neřadí jen podle jednoho parametru, kterým je jméno, ale musí se rozhodnout i zda se jedná o složku, které se podle vybrané volby řadí na začátek.

Duhou možností je řadit prvky pomocí selektoru. To je ukazatel na metodu nějakého objektu a v principu funguje jako funkce, pouze je součástí nějaké třídy. Zajímavou možností je využití řadicích deskriptorů `NSSortDescriptor`, které se používají v poslední variantě řazení. Tento deskriptor je tvořen polem, ve kterém jsou uloženy názvy klíčů, podle kterých se porovnává. Prvním deskriptorem je primární klíč, podle kterého se porovnávají všechny prvky. Další položky jsou využity, vždy když předchozí klíč vrátí stejné hodnoty. Pro podrobnější popis je nutno nahlédnout do dokumentace.

Ani jednu z uvedených metod jsem nevyužil, protože při použití `NSArray` nelze řadit prvky přímo v poli, ale vrátit pouze ukazatel na nové pole, které obsahuje seřazené ukazatele na původní

Ukázka zdrojového kódu 15 Ukázka řadicí funkce

```
int sortByFileName(id objectA, id objectB, GSFSortOption *sortOption)
{
    if( [objectA isUpFolder] )
        return NSOrderedAscending;
    else if ( [objectB isUpFolder] )
        return NSOrderedDescending;

    int temp;
    if ( (temp = sortByDirectory(objectA, objectB, sortOption)) !=
                                                        NSNotFound ){
        return temp;
    }

    if ( [sortOption isRevertedOrder] ){
        if( [[objectA name] caseInsensitiveCompare:[objectB name]] ==
                                                        NSOrderedAscending )
            return NSOrderedDescending ;
        else if( [[objectA name] caseInsensitiveCompare:[objectB name]] ==
                                                        NSOrderedDescending )
            return NSOrderedAscending;
        else
            return NSOrderedSame;
    }
    else {
        return [[objectA name] caseInsensitiveCompare:[objectB name]];
    }
}
```

objekty. Těmto objektům se při každém řazení zvýší `retainCount` o jedna a tím se zamezí jejich zrušení. To se mi nepodařilo odladit, protože při zrušení jednoho z polí došlo k pádu aplikace, přitom při testovaní došlo opravdu ke zvýšení počtu ukazatelů na pole.

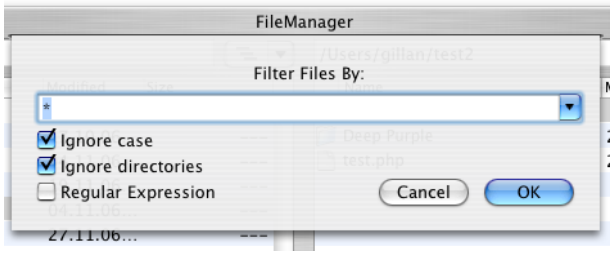
Problém jsem řešil poměrně dlouho, až jsem přišel na to, že mohu využít dynamické pole `NSMutableArray`, které implementuje metody:

- `sortUsingDescriptors:`
- `sortUsingFunction:context:`
- `sortUsingSelector:`

Tyto metody pracují stejně jako jejich protějšky u `NSArray` s tím rozdílem, že nevrací žádnou hodnotu, ale změní pořadí prvků přímo v poli. Tak lze dosáhnout seřazení pole aniž by bylo potřeba vytvářet pole nové. Samozřejmostí je, že nedochází k přesunům prvků v paměti, ale pouze ke změně pořadí ukazatelů na ně.

Pro řazení souborů jsem použil metodu `sortUsingFunction:context:`, která je sice, jak píše dokumentace, méně flexibilnější, protože pomocí deskriptorů lze definovat řazení na několik řádků, ale umožňují lepší kontrolu nad řazením. Řazení souborů je specifické tím, že se adresáře umísťují na začátek. Zachoval jsem také možnost řadit soubory stejně jako ve Finderu bez rozlišení zda se jedna o složku nebo soubor. Tato vlastnost je funkční, ale nemá vytvořené ovládání.

Kromě řazení se třída `GSFPanelController` stará také o filtrování vypisovaných souborů. Základním filtrem je odstranění skrytých souborů, které začínají v Unixových systémech tečkou. Tyto soubory jsou prozatím odstraňovány z výpisu vždy, ale až bude vytvořen panel pro nastavení bude přidána volba, zda skryté soubory vypisovat či nikoliv. Pokročilé filtrování umožňuje vypisovat jen ty soubory o které má uživatel zájem. Dialog pro nastavení filtrování je na obrázku 4.2.



Obrázek 4.2: Dialog pro nastavení filtrování souborů

Filtrování je možné buď pomocí zástupných značek * a ?, které představují libovolný počet znaků nebo právě jede znak, anebo pomocí regulárních výrazů, kterým vyhoví jen požadované soubory. První varianta je jednoduchá a použitelná i pro nezkušené uživatele. Například pokud je zadán řetězec `*.h` budou vypisovány všechny hlavičkové soubory. Při zadání `img_2006-10-*.jpg` budou vypsány všechny obrázky s jpg koncovkou a začínající na *img_2006-10-*. Pří správném pojmenování souborů tak lze získat všechny fotky z října roku 2006. Zástupný znak ? nahrazuje jedno písmeno v názvu. Pro získání plného výpisu souborů je potřeba zadat znak *, který zastupuje všechny soubory ve složce.

Druhá, pokročilejší varianta umožňuje filtrovat pomocí regulárních výrazů. Aplikace porozumí regulárním výrazům, tak jak je interpretuje Perl, protože jsem využil Cocoa verzi knihovny pro interpretaci Perl-kompatibilních regulárních výrazů `AGRegex`. Tato knihovna je volně dostupná a její použití je velmi jednoduché.

```
AGRegex * regex = [[AGRegex alloc] initWithPattern:@"regularni vyraz"];
AGRegexMatch = [regex findInString:@"prohledavany retezec"];
```

Výstupem je objekt, který obsahuje výsledky hledání. Jeho hlavička vypadá takto:

```
@interface AGRegexMatch : NSObject {
        AGRegex *regex;
        NSString *string;
        int *matchv;
        int count;
}
```

Pokud není nalezen vyhovující řetězec, je vracen `nil`. Třída `AGRegex` poskytuje více metod pro práci s regulárními výrazy, ale pro účely této diplomové práce stačí uvedená metoda. Tato metoda má volitelný parametr `option:int`, který nastavuje volby při vyhledávání. Mezi ně patří například používané `AGRegexCaseInsensitive`, které způsobí ignorování velikosti písma Otestování zda řetězec vyhovuje zadání je velmi jednoduché, protože se pouze vytvoří objekt s regulárním výrazem a aplikuje na řetězec, který chceme ověřit. Pokud je výsledkem cokoliv jiného než `nil`, řetězec vyhověl.

Ve skutečnosti i první varianta filtrování využívá knihovnu AGRegex. Řetězec se zástupnými znaky * a ? je převeden na regulární výraz a ten je pak zpracován stejným způsobem jako regulární výraz zadaný přímo.

Tyto funkce se ještě využívají při hromadném označování souborů. Označené soubory se ukládají v `GSFHistoryController`, který je popisován v následující kapitole, a jsou to přímé odkazy na konkrétní objekty `GSFInode`. Dříve bylo označení souborů uloženo v množině jako sada čísel

s pozicí která je označena, ale tento způsob byl naprosto nevhodný pro řazení, neboť se tak nedalo uchovávat správné pořadí. Při označení je objektu `GSFPanelController` odeslána zpráva s pozicí, ale ten si ji převede na konkrétní soubor a předá dále k uložení objektu `GSFHistoryController`, který je uchovává v množině `NSMutableSet`. Při hromadném označení je předán řetězec, podle kterého se mají soubory vybírat, stejně jako u filtru a stejným způsobem probíhá i zpracování. Všechny uzly, které vyhovují zadanému řetězci jsou buď označeny nebo odznačeny tím, že se přidají nebo odeberou z množiny, ve které se ukládají. Hromadné označení probíhá nad všemi soubory v aktuální složce.

U filtrování i označování jsou volby *Ignore directories* a *Ignore case*. První z nich při provádění akce ignoruje adresáře a buď je nepodrobí filtrování a rovnou předá na výstup nebo u nich neřeší označování. Druhou volbou lze nastavit citlivost na velikost písma. Značení souborů se provádí vždy na filtrovaném seznamu, aby se zamezilo zbytečným operacím při kontrole vyhovujících souborů.

Historie procházených složek

`GSFPanelController` je objektem, který v aplikaci vystupuje jako výhradní zprostředkovatel souborů a nositel aktuální cesty v levém nebo pravém panelu, ale on sám využívá ještě pomocný objekt, do kterého si tyto cesty ukládá aby je mohl využít při rolování v historii. Tímto objektem je `GSFHistoryController`, kterému jsou předány údaje o aktuální cestě, ukazatel na modul pro přístup k souborům, položka na které se nachází kurzor a množina vybraných souborů a složek. Tento objekt je uložen do zásobníku `GSFStack`, který je navržen speciálně pro účely ukládání historie. Umí automaticky zahazovat nejstarší otevřené složky, aby nerostla velikost používané paměti do nekonečna.

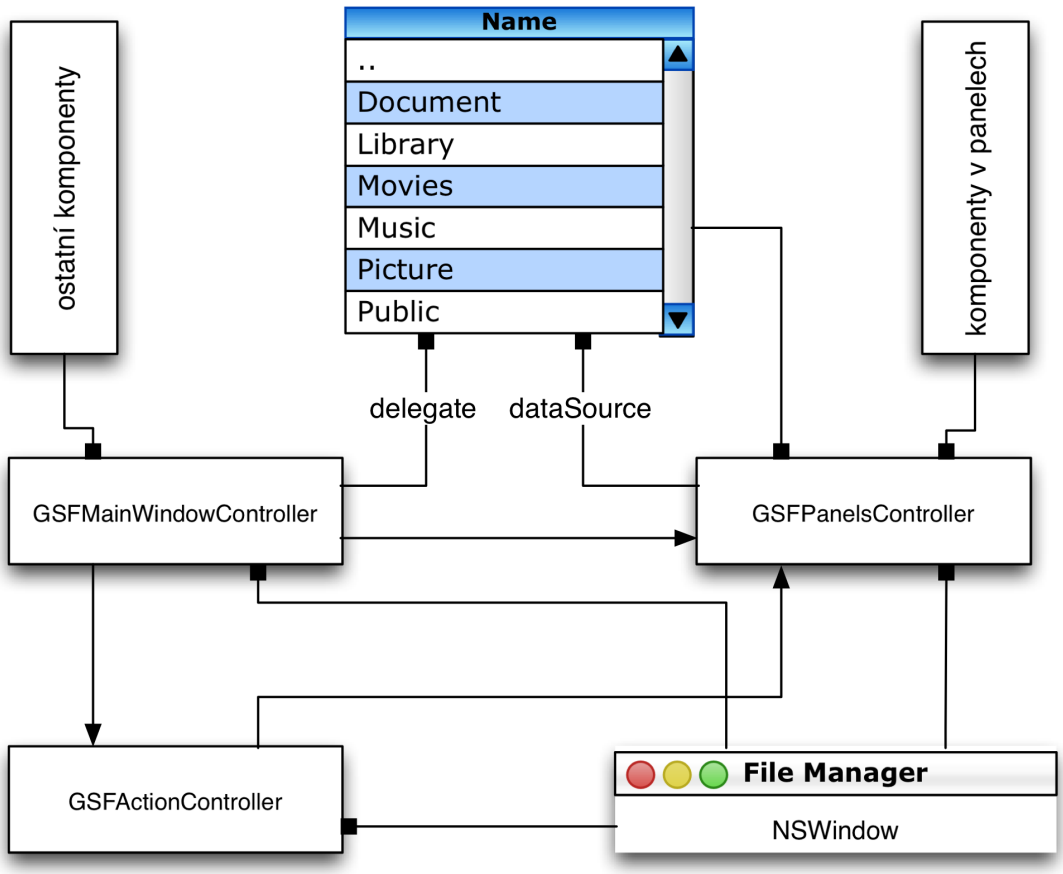
Při změně cesty, která se provede odesláním zprávy `setActualPath:` s aktuální cestou objektu `GSFPanelController`, dojde k vytvoření nového objektu `GSFHistoryController`, který se uloží do zásobníku. Pokud aplikace potřebuje vědět aktuální cestu v jednom z panelů, odešle mu zprávu `actualPath` a příjemce deleguje tento dotaz do nejvýše uloženého objektu v zásobníku a vrátí jeho odpověď. Stejným způsobem fungují i dotazy na ostatní údaje, které je schopen `GSFHistoryController` uchovávat.

`GSFHistoryController` je pomocným objektem, který uchovává informace pro aktuálně otevřenou složku. Jeho metody pouze nastavují a přistupují k jeho proměnným a neobsahují žádný výkonný kód.

4.2.6 Zpracování a delegování událostí v hlavním okně

Hlavní okno aplikace obsahuj dva nezávislé panely, které zobrazují dva na sobě nezávislé adresáře. Tyto panely jsou doplněny pomocnými komponentami, které zobrazují aktuální cestu, informace o zobrazených souborech apod. Na chodu okna se podílí tři kontroléry: `GSFPanelsController`, který byl popsán na straně 80, `GSFMainWindowController`, `GSFActionController`, které popíši nyní. Na obrázku 4.3 je znázorněno jakým způsobem se delegují zprávy a požadavky mezi těmito kontroléry a jak jsou napojeny komponenty na tyto kontroléry. Toto propojení se provádí v Interface Builderu a v objektech je vyžadováno klíčovým slovem `IBOutlet`. Pokud by při návrhu okna nebyly instance správně propojeny, aplikace by prostě nefungovala, přestože by se při překladu neobjeví žádná chyba.

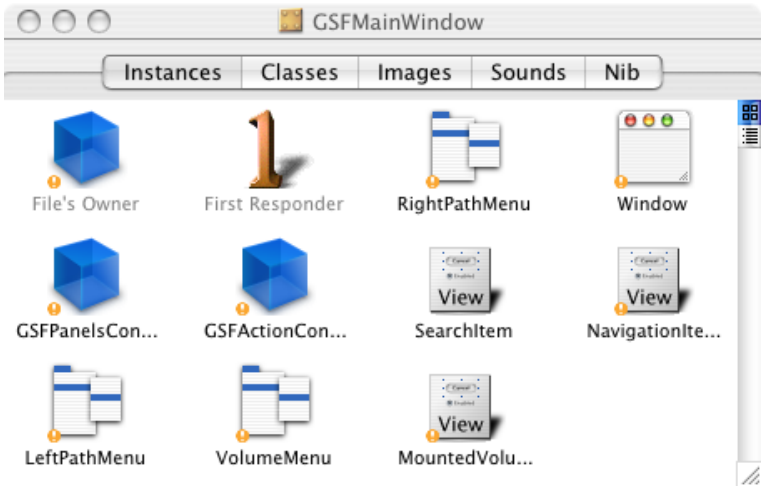
Hlavní kontrolér okna je `GSFMainWindowController`, který se stará o zpracování událostí předávaných oknem a je příjemcem zpráva jako `NSFirstResponder`, který dostává zprávy z menu apod. Pro každou položku v menu je vytvořena jedna akce (speciálně označená metoda), která zpracovává kliknutí na tuto položku. Jsou dvě možnosti jak vyřešit přijímaní zpráv z menu. Buď mít



Obrázek 4.3: Spolupráce objektů v hlavním okně `GSFMainWindow`. Komponenty v panelech jsou všechny ovládací prvky obsažené v levém i pravém panelu a jsou vždy dvakrát a velmi úzce souvisí se zobrazovanými soubory. Mezi ostatní komponenty patří například `NSSplitView`.

jednu metodu, která bude přijímat zprávy od všech položek menu a rozhodnout se na základě odesílatele, nebo mít pro každou položku jednu metodu, která se bude starat výhradně o příjem konkrétní zprávy. Já jsem se rozhodl pro druhý způsob, který má sice za následek o něco větší kód, ale odpadá režie způsobená dalším ověřením kdo je odesílatelem této zprávy. Kromě toho mi také tento způsob připadá přehlednější, protože při úpravách lze vyhledat konkrétní metodu pomocí nástrojů v Xcode. Mimo to `GSFMainWindowController` také zpracovává stisknuté klávesy, kliknutí myší a je příjemcem zpráv při drag & drop i copy & paste operacích. Tyto události však jen přijímá a deleguje je dále.

Události týkající se pohybu kurzoru, označování souborů, otevíraní složek nebo spouštění aplikací se předávají objektu `GSFPanelsController`, který tyto události umí zpracovat. Ostatní se předávají druhému kontrolérů `GSFActionController`, který řídí průběh všech operací, které se v aplikaci provádí. `GSFMainWindowController` má za úkol také zobrazovat potřebná dialogová okna. Těmto dialogovým oknům je jako delegát předáván většinou `GSFActionController`, který podle odpovědi uživatele dokončí požadovanou operaci. Na obrázku 4.4 je zobrazeno okno Interface Builderu s instancemi tříd použitých v `GSFMainWindow`.



Obrázek 4.4: Všechny instance řídících prvků hlavního okna `GSFMainWindow`. Obsahuje také speciální `NSView` položky, které slouží pro uchování tlačítek z panelu nástrojů (`NSToolbar`).

4.2.7 Komponenta pro zobrazování souborů

Zobrazování souborů je klíčovým prvkem souborového manažera, protože jeho rychlost a přehlednost určuje míru komfortu uživatele. Nepřehledné zobrazení pouze ztěžuje orientaci ve vypsaných souborech a složkách. Je-li zobrazení pomalé, stává práce velmi nepříjemnou například při rychlém procházení složek. Rychlost zobrazení není ovlivněna jen napsanou komponentou, ale také jak rychle je aplikace schopna dodat data, avšak její náročnost by měla být velmi minimalizována.

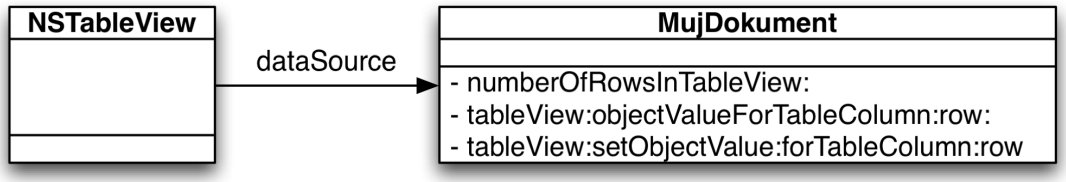
Cocoa nabízí pro zobrazování seznamů třídu `NSTableView`, která je schopná zobrazovat data ve sloupcích. Její použití takzvaných pomocných objektů (*Helper Object*). Ty poskytují data jiným objektům v případě, že jsou o ně požádány. Jedná se o opačný přístup, než když programátor sám určí a řekne: „Do buňky 3 na řádku 5 zapiš hodnotu *Hello world!*.“ V Cocoa se používá opačný přístup, kdy si objekt třídy `NSTableView` sám určuje, který sloupec potřebuje zobrazit a pošle zprávu svému pomocnému objektu `dataSource` (volně přeloženo: zdroj dat), aby mu poslal data, která mají být na požadovaném místě uvedena. Tento objektu musí implementovat tyto dvě metody:

```
- (int)numberOfRowsInTableView:(NSTableView *)aTableView;
```

Objekt `dataSource` vrátí počet řádků, které budou zobrazeny.

```
- (id)tableView:(NSTableView *)aTableView
      objectValueForTableColumn:(NSTableColumn *)aTableColumn
                            row:(int)rowIndex;
```

Objekt `dataSource` vrátí objekt, který bude zobrazen na řádku s číslem `rowIndex` a ve sloupci `aTableColumn`.



Obrázek 4.5: Vztah objektu dataSource a NSTableView

Pokud mají být buňky v tabulce editovatelné, pak musí delegát implementovat ještě jednu metodu:

```
- (void)tableView:(NSTableView *)aTableView
   setObjectValue:(id)anObject
   forTableColumn:(NSTableColumn *)aTableColumn
              row:(int)rowIndex;
```

Co se tyká zobrazování dat v tabulce, je pozice programátora velmi pasivní. Vytvořený objekt poskytující data, čeká dokud není požádán o konkrétní hodnotu. Až je tabulka nachystaná zobrazit hodnotu pátého řádku, sama si o něj řekne. Tento způsob je efektivnější, protože pokud bude mít tabulka třeba deset tisíc řádků, je nesmysl snažit se je zobrazit všechny najednou. Na obrazovce může být zobrazeno většinou jen několik desítek záznamů, a právě proto si o ně `NSTableView` samo požádá.

Jak tedy přinutit tabulku aby zobrazila nové data, pakliže se změnila? Odešleme ji zprávu `reloadData`. Tím je objektu `NSTableView` oznámeno, že data která zobrazuje jsou neplatná, a on sám požádá o překreslení těch buněk, které jsou zobrazeny.

Při vytváření komponenty pro zobrazování souborů jsem použil právě třídu `NSTableView`, od které jsem vytvořil potomka `GFSTableView`. Tato třída pouze přímo implementuje zachytávání událostí stisknutí kláves a kliknutí myší, které pak předává svému delegátovi. Předpokládám, že bych mohl využít pouze původní třídu, ale nepodařilo se mi ji donutit, aby nezpracovávala události svým vlastním způsobem (výběr řádku, editace buňky, ...), ale delegovala je dále.

Třída `GSFTableView` přetěžuje tyto metody svého předka:

```
- (void)keyDown:(NSEvent *)event;
- (void)mouseDown:(NSEvent *)event;
```

Původní třída tyto události sama zachytila a zpracovala a nebylo možné je dále použít. Já jsem potřeboval zpracovávat události v aplikační vrstvě, abych mohl vytvořit například výběr složek a adresářů podle jejich počátečních písmen, proto jsem zmíněné metody přetížil.

Na rozdíl od `NSTableView` neřeší třída `GSFTableView` posun po řádcích sama, ale předává stisknutí klávesy aplikační vrstvě, ve které se provede výpočet nového řádku a ten se pak oznámí zpět tabulce, aby jej vybrala. Je tak zajištěna kontrola nad aktuálně vybraným souborem.

4.2.8 Kopírování souborů

Pro kopírování souborů umožňuje Cocoa velmi jednoduše pomocí třídy `NSWorkspace`, která poskytuje metodu `performFileOperation:source:destination:files:tag:`. Tato metoda umožňuje provádět většinu používaných operací nad soubory. Tyto operace se definují pomocí konstant, které jsou přímo v Cocoa definované:

- `NSString *NSWorkspaceMoveOperation;`
- `NSString *NSWorkspaceCopyOperation;`
- `NSString *NSWorkspaceLinkOperation;`
- `NSString *NSWorkspaceCompressOperation;`
- `NSString *NSWorkspaceDecompressOperation;`
- `NSString *NSWorkspaceEncryptOperation;`
- `NSString *NSWorkspaceDecryptOperation;`
- `NSString *NSWorkspaceDestroyOperation;`
- `NSString *NSWorkspaceRecycleOperation;`
- `NSString *NSWorkspaceDuplicateOperation;`

Použití je velmi snadné. Jako první parametr se uvede požadovaná operace, druhý a třetí parametr jsou zdroj a cíl, parametr files je typu `NSArray` a obsahuje všechny soubory a složky, nad kterými se má operace provést, poslední parametr předává výsledek prováděné operace a metoda vrací YES nebo NO v případě úspěchu či neúspěchu. První nevýhodou této metody je, že není schopna poskytovat informace o průběhu a kontrolovat její provádění (pozastavit, zrušit nebo znovu spustit). Druhým a důležitějším nedostatkem je, že umí pracovat pouze se soubory, které se nachází v lokálním souborovém stromu. Úmyslně neuvádím s lokálními soubory, protože do stromu může být připojený vzdálený adresář nebo síťový disk pomocí některého z podporovaných protokolů.

Tyto dva nedostatky mi bránily tuto metodu využít. Pro potřeby souborového manažera je důležité mít naprostou kontrolu nad prováděnou operací a také zobrazit její průběh uživateli. Také jsem potřeboval zajistit přenositelnost na různé moduly, a proto jsem si vytvořil vlastní třídu nazvanou `GSFCopyThread`, která zajišťuje kopírování a přesouvaní souborů s využitím datových proudů. Jak název třídy napovídá, je uzpůsobena spouštění jako vlákno. Poskytuje jednu metodu, která provádí kopírování či přesouvání, nesoucí název `myThreadMethod:`, jež je pozůstatkem názvu z příkladu využití vláken a při využití v systému je patrné, že se jedná o vlákno. Po inicializaci parametrů se tato metoda předá vytvořenému vláknu k provádění. Vlákno, které provádí akci kopírování, má vlastní `NSRunLoop`, ve kterém se provádí. Proto musí nejnižší vrstva poskytující datové proudy jako přístup k obsahu souborů akceptovat předaný `NSRunLoop` viz strana 68.

Kopírování celých adresářových struktur se zobrazením průběhu není triviální záležitostí jak by se mohlo na první pohled zdát a skládá se z několika kroků. Nejprve se musí definovat objekty, které se budou přesouvat nebo kopírovat, nastavit zdrojový a cílový kontrolér, který zajistí správnou

interpretaci souborů, a předat výsledek zadání uživatele pomocí dialogu. Při inicializaci vlákna je potřeba také uvést delegáta, který přijme výsledek kopírování.

Zda se bude provádět kopírování nebo přesouvání je rozhodnuto při inicializaci objektu. Tento objekt je vytvořen buď pomocí konstruktoru `initAsCopy` nebo `initAsMove` v závislosti na vybrané akci. Tyto konstruktory nastaví přepínač, který rozhoduje o tom zda se provede po zkopírování souboru také jeho vymazání na původní pozici. Protože přesouvání souborů na lokálním disku by mělo být velmi rychlé, využil jsem výše popsané metody `performFileOperation` pro soubory, které využívají modulu pro práci s lokálními soubory, a tedy se nachází na jednom disku. Tím jsem sice mírně narušil úplnou nezávislost kopírování na fyzické reprezentaci virtuálních souborů, ale modul pro práci se soubory na lokálním disku bude v aplikaci přítomen vždy, proto může být takto přímočaře využit. V ostatních případech se přesouvání od kopírování liší pouze tím, že po dokončení jsou soubory a složky odstraněny.

Kopírovací vlákno si samo zajistí dialog, ve kterém zobrazí průběh kopírování a pomocí kterého lze kopírování kontrolovat. Tento dialog je vytvořen ihned po spuštění vlákna. Kopírování předchází analýza kopírovaných souborů. Vláknu jsou předány pouze cesty označených souborů a adresářů na nejvyšší úrovni. Pro kopírování je potřeba znát cesty všech kopírovaných položek, které se nachází v podadresářích. K tomu slouží metoda `enumerateFilesFromPath`, která projde předané adresáře, jsou-li nějaké, a vytvoří pole dvojic cest zdroj - cíl. Druhá část dvojice je potřebná k tomu, aby bylo možno vytvářet další úrovně stromu v cílové destinaci. Postupně se bere část cesty za výchozí cestou daného adresáře a připojuje se na konec cesty adresáře cílového. Při této analýze se také vypočítá velikost a počet kopírovaných souborů, aby bylo možno zobrazit průběh prováděné akce. Průběh této analýzy se již zobrazuje v informačním dialogu kopírování.

Jakmile je analýza složek a souborů dokončena, začíná samotná akce kopírování. Postupně se prochází pole, do kterého byly uloženy dvojice zdroj - cíl, a ty se předávají příslušným kontrolérům aby vrátily vstupní respektive výstupní datový proud. Po otevření proudů předává řízení přímo `NSRunLoop`, které běží v cyklu a odchytává události, které oba datové proudy vyvolávají. Mezi tyto události patří oznámení, že vstupní proud má nachystána data ke čtení a výstupní proud je připraven tato data přijmout. K oběma těmto stavům může dojít nezávisle na sobě a vlákno čeká až nastanou oba dva. Pak se zavolá metoda `copyData`, která načte připravená data a uloží do vyrovnávacího bufferu a z něj je pak přepíše do výstupního datového proudu. Tato část kódu je jediným místem v aplikaci, kde se pracuje přímo s ukazateli a nízkoúrovňovým polem jazyka C.

Po přečtení části dat velikosti bufferu se přepočítá velikost zbývajících dat a obnoví se informační ukazatele průběhu kopírování. K obnovení dochází jen jednou za čas, když je aktuálně zkopírovaná část větší než nastavená procentuální část.

Pokud už vstupní proud nemá žádná další data ke čtení, vyvolá událost uzavření datového proudu, který je zpracován stejnou metodou jako ostatní události a oba proudy jsou uzavřeny. Tím končí kopírování jednoho souboru. Pokud se jedná o přesun souboru, je po dokončení překopírování dat tento soubor z původního místa odstraněn.

Jak jsem uvedl na začátku kapitoly, není kopírování stromové struktury úplně jednoduché. Při kopírování je potřeba zajistit, aby se na cílové cestě vytvořil patřičný podstrom. Při jeho vytváření je však potřeba zjišťovat, zda již tato cesta existuje či nikoliv a zda je tato cesta souborem nebo složkou. Daný soubor nebo složka nesmí existovat nebo musí být shodného typu. Soubor lze přepsat pouze souborem a složku jen složkou. Ta se ve skutečnosti nepřepisuje, pouze se použije. Pokud by mělo dojít k záměně složky za soubor nebo opačně, musí aplikace oznámit chybu. Zobrazí dialog, ve kterém lze akci opakovat, zrušit nebo přeskočit její provádění.

Pokud má dojít při kopírováni k přepsání existujícího souboru je zobrazen dialog, kterým aplikace zjistí jak se má zachovat. Původní soubor lze buď přeskočit nebo přepsat souborem novým. Tyto dvě operace lze nastavit pro všechny následující případy existujících souborů, bez opětovného

zobrazení tohoto dialogu. Dialog také umožňuje zadat nový název, se kterým bude původní soubor kopírován.

Zobrazený dialog přeruší provádění vlákna na počátku kopírovacího cyklu za místem detekce duplicitního souboru, nastaví status existence souboru na YES a zobrazí modální okno, které si vyžádá všechny vstupy od uživatele. Po uzavření toto modální okno předává řízení zpět vláknu, ovšem to přeskakuje do metody `copyContinue:newName`, která nastaví proměnnou `overWriteStatus`. Po ukončení této metody pokračuje na původním místě přerušení. Protože byl nastaven status existence souboru na YES, dojde k vyhodnocení proměnné, která udává jak se má naložit s přepisovaným souborem (`overWriteStatus`). Má-li být soubor přeskočen, je jednoduše vynechána operace kopírování. Má-li být přepsán, provede se kopírování jako kdyby neexistoval. Žádné další speciální operace se neprovádí.

Po dokončení kopírování jednoho souboru, se vynuluje proměnná `overWriteStatus`, pokud nebyla nastavena na trvalé provádění přeskočení či přepisování. Dále je při přesunu provedeno vymazání původního souboru. Po dokončení celé větve je při přesouvání odstraněn také prázdný adresář.

Tyto úkoly se provádí v cyklu dokud není vyčerpáno celé pole obsahující soubory ke kopírování. Jakmile je kopírování dokončeno, je aplikace upozorněna přes `NSNotificationCenter`, že došlo k aktualizaci obsahu disku. Tuto notifikaci zpracovávají kontroléry zajišťující zobrazování souborů (viz strana 82), které zobrazují soubory aktuální složky a jsou tak uvědomeny, že musí tyto soubory opět načíst.

Uživatel může kopírování kdykoliv přerušit kliknutím na patřičné tlačítko. Tím se nastaví proměnná `keepCopy`, která udává zda se má v kopírování pokračovat. Tato proměnná se testuje v každém kroku a pokud je zjištěno, že uživatel chce kopírování ukončit, dojde k okamžitému přerušení. Nedokončené kopírování souboru nechá tento soubor v takové velikosti, v jaké byl přerušen. Sofistikované neúplných souborů plánuji do budoucna.

Po potvrzení dialogu kopírování se stává celá operace zcela autonomní a je řízena samostatným vláknem s vlastním `NSRunLoop`. Jakmile je operace dokončena je využito společného notifikačního centra, aby byla aplikace uvědomena o ukončení operace.

4.3 Operace se schránkou

4.3.1 Jak funguje schránka

Tato kapitola čerpá z [Hil05].

Velmi užitečnou funkcí v souborovém manažeru jsou operace copy & paste. Lze tak vložit soubor nebo celé složky z jiného programu (například z Finderu) a naopak. Implementace je s využitím Cocoa frameworku velmi jednoduchá.

V každém Mac OS X běží proces `/System/Library/CoreServices/pbs` - „pasteboard server“, který se česky běžně nazývá schránka. Aplikace využívají třídu `NSPasteboard` k zápisu dat do tohoto procesu a ke čtení z něj. Schránka umožňuje operace jako kopírování (copy), vystřižení (cut) a vložení (paste) mezi aplikacemi.

Aplikace mohou vkládat data do schránky v několika formátech. Například obrázek může být vložen do schránky jako PDF dokument a také jako bitmapa. Aplikace, která data čte si vybere data, která ji nejvíce vyhovují.

Při vkládání dat do schránky typicky aplikace deklaruje typ, který vkládá, a v zápětí tato data do schránky nakopíruje. Přijímající aplikace se nejprve zeptá schránky zda je k dispozici požadovaný typ dat a potom si přečte data v preferovaném formátu.

Druhou možností je kopírovat data do schránky „líným způsobem“. To znamená, že se jednoduše dekklarují všechny typy dat, které mohou být vloženy do schránky a potom jsou data dodána až jsou potřeba. Pro podrobnější popis doporučuji literaturu [Hil05]

K dispozici jsou také vícenásobné schránky. Existuje schránka pro kopírování a vkládání (copy & paste) a jiná pro táhni a pusť (drag & drop) úlohy. Existuje schránka, která ukládá poslední hledaný řetězec nebo schránka pro kopírování pravítek (rules) a jiná pro kopírování fontů.

Třída `NSPasteboard` funguje jako rozhraní k serveru schránky. Zde uvedu některé běžně používané metody `NSPasteboard`.

```
+ (NSPasteboard *)generalPasteboard
```

Vrátí ukazatel na obecnou schránku. Tato schránka se používá pro běžné operace cut, copy a paste.

```
+ (NSPasteboard *)pasteboardWithName:(NSString *)name
```

Vrátí ukazatel na schránku definovanou jménem. Existuje seznam globálních proměnných, které obsahují jména standardních schránek.

```
NSGeneralPboard
NSFontPboard
NSRulerPboard
NSFindPboard
NSDragPboard
```

```
- (int)declareTypes:(NSArray *)types owner:(id)theOwner
```

Tato metoda vynuluje všechno co se nachází ve schránce a deklaruje typy dat, které `theOwner` vloží do schránky. Zde je seznam globálních proměnných pro standardní typy:

```
NSStringPboardType
NSFilenamesPboardType
NSPostScriptPboardType
NSTIFFPboardType
NSRTFPboardType
NSTabularTextPboardType
NSFontPboardType
NSRulerPboardType
NSFileContentsPboardType
NSColorPboardType
NSRTFDPboardType
NSHTMLPboardType
NSPICTPboardType
NSURLPboardType
NSPDFPboardType
NSVCardPboardType
NSFilesPromisePboardType}
```

Vytvářet lze také vlastní typy schránek například pro vnitřní učely aplikace. Schránka nemusí být využita pouze pro akce copy-paste, ale například pro nejrůznější uložení dat pro sdílení, v rámci aplikace nebo mezi nimi. Nepředstavuje ovšem ideální řešení, protože vyžaduje poměrně velkou režii, která odpadá při použití například `NSPort`.

```
- (BOOL)setData:(NSData *)aData forType:(NSString *)dataType
- (BOOL)setString:(NSString *)s forType:(NSString *)dataType
```

Zápis dat do schránky.

```
- (NSArray *)types
```

Vrátí pole typů dat, která jsou k dispozici ke čtení ze schránky.

```
- (NSString *)avaliableTypeFromArray:(NSArray *)types
```

Vrátí prvni typ nalezený v poli `types`, který je k dispozici ke čtení ze schránky. `types` by měl být seznam všech typů, které je aplikace schopna číst.

```
- (BOOL)dataForType:(NSString *)dataType
- (BOOL)stringForType:(NSString *)dataType
```

Čtení dat ze schránky.

4.3.2 Využití v souborovém manažeru

Zápis do schránky

Souborový manažer by měl uživatelům umožnit kopírovat soubory a složky pomocí schránky. Označením souborů a vyráním akce „copy“ se uloží do schránky cesty k souborům. Cílová aplikace, která může být stejné nebo jiná, si tyto cesty zase přečte a s vybranými daty provede nějakou akci. Souborový manažer tyto složky a soubory s největší pravděpodobností zkopíruje na nové místo.

Programátor nemá s použitím schránky prakticky žádnou práci. Pro zapsání dat do `psb` je potřeba dvou metod (viz ukázka 16). Po první metodě `declareTypes:owner`, kterou aplikace odešle informace o tom, jaké typy dat bude do schránky zapisovat, by měla následovat metoda druhá, kterou data zapíše. Exsitují tyto metody pro zápis do schránky:

```
- (BOOL)setData:(NSData *)data forType:(NSString *)dataType
- (BOOL)setString:(NSString *)string forType:(NSString *)dataType
- (BOOL)setPropertyList:(id)propertyList forType:(NSString *)dataType
```

V mém případě jsem použil metodu `setPropertyList:forType`, která je schopna odeslat datovou strukturu `NSArray`, ve které mám uloženy cesty, které se budou kopírovvat. Typ ukládaných dat je `NSFilenamesPboardType`, který oznámí všem aplikacím, které budou ze schránky číst, že se jedná o cesty k souborům nebo složkám. Zkopíruji-li soubor ve svém manažeru a vložím jej ve Finderu, bude automaticky rozpoznán a Finder si tento soubor zkopíruje.

Definovat lze několik různých typů dat, například bych mohl přidat ještě obsah textového souboru, pak by aplikace preferující textová data raději sáhla po obsahu tohoto souboru. Nutno podotknout, že počet deklarovaných typů se musí shodovat s počtem dat, která se do schránky zapíší.

Metodu, která vykonává samotné vložení do schránky, jsem zapouzdřil do jiné, aby mohla být zajištěna univerzálnost získání souborů, které do schánky vkládají. Při obbyčejném kopírování se pouze načtou označené soubory a zavolá se metoda `writeFilenamesToPasteboard:filenames:`, jejímž prvním parametrem je schránka, do které se budou soubory zapisovat, a druhý parametr je pole s cestami. Tato metoda se využije také při operaci táhni a pusť popsané na straně 95.

Ukázka zdrojového kódu 16 Zapsání souborů do schránky

```
- (BOOL)writeFilenamesToPasteboard:(NSPasteboard *)pb filenames:(NSArray *)filenames
{
    [pb declareTypes:
        [NSArray arrayWithObject:NSFilenamesPboardType]
              owner:self];
    [pb setPropertyList:filenames forType:NSFilenamesPboardType];

    return YES;
}

- (void)copySelectedFilesInPanel:(GSFPanelController *)aPanel
{
    [self writeFilenamesToPasteboard:[NSPasteboard generalPasteboard]
                           filenames:[aPanel returnSelectedFilesPaths]];
}
```

Čtení ze schránky

Čtení ze schránky je sice stejně jednoduché jako zápis, ale přibývá implementace akce, která se musí s vloženými daty provést. V ukázkazce 17 je uvedena metoda, která si ověří, zda schránka obsahuje správný datový typ - `NSFilenamesPboardType`. Pokud jej obsahuje, tak jej načte pole obsahující cesty pomocí metody `propertyListForType:` a vytvoří vlákno pro kopírování a tyto cesty mu předá. Jako zdrojové cesty je vložen obsah schránek, jako cílová cesta je uvedena aktuální cesta vybraného panelu.

Stejně jako pro zapisování do schránky existuje více metod pro čtení:

```
- (NSString *)stringForType:(NSString *)dataType
- (id)propertyListForType:(NSString *)dataType
- (NSData *)dataForType:(NSString *)dataType
```

Vývojáři frameworku Cocoa usnadnili práci se schránkou na maximální možnou míru. Jednodušší už to snad být nemůže.

4.4 Operace Táhni a pusť

4.4.1 Jak funguje táhni a pusť

Tato kapitola čerpá z [Hil05].

Táni a pusť si vyžaduje o něco více než prosté kopírování do chránky. Když tažení začne, jsou nějaká data zkopírována do schránky pro operace drag-and-drop. Po upuštění taženého objektu, jsou data ze chránky přečtena. Jediná věc, která dělá tuto techniku zajímavější než copy-and-paste, je že uživatel potřebuje nějakou odezvu: obrázek, který se objeví při tažení, zvýrazněná komponenta při tažení do ní, a možná nějaký zvuk, když je obrázek upuštěn.

Při tažení dat z jedné aplikace do druhé může stát několik věcí: nic se nestane, může být vytvořena kopie dat nebo může být vytvořen odkaz na data existující. Konstanty reprezentující tyto operace:

```
NSDragOperationNone
NSDragOperationLink
NSDragOperationCopy
```

Konstanty pro méně časté operace:

Ukázka zdrojového kódu 17 Přečtení souborů ze schránky

```
- (void)pasteFilesInPanel:(GSFPanelController *)aPanel
{
    NSPasteboard *pb = [NSPasteboard generalPasteboard];
    NSArray *value;
    NSString *type;

    type = [pb availableTypeFromArray:[NSArray arrayWithObject:NSFilenamesPboardType]];

    if( type ){
        value = [pb propertyListForType:type];
        GSFCopyThread *pom = [[GSFCopyThread alloc] initAsCopy];

        /** Gets files to copy */
        [pom setFromPaths:value];

        /** sets destination path from dialog (as parametr of this method */
        [pom setToPath:[[panels activePanel] actualPath]];
        /** original path */
        [pom setNewToPath:[[panels activePanel] actualPath]];
        /** sets controllers */
        [pom setFromController:[[panels activePanel] dataController]];
        [pom setToController:[[panels inActivePanel] dataController]];
        [pom setDialogSender:self];

        [NSThread detachNewThreadSelector:@selector(myThreadMethod:)
                                 toTarget:pom
                               withObject:nil];
        [pom release];
    }
}
```

`NSDragOperationGeneric`
`NSDragOperationPrivate`
`NSDragOperationMove`
`NSDragOperationDelete`
`NSDragOperationEvery`

Jak zdroj tak cíl musí souhlasit s operací, která má nastat když uživatel upustí tažený obrázek.

Při přidávání služeb táhni a pusť do komponenty pohledu (view) existují dvě odlišné části, které je potřeba naprogramovat:

- Udělat ji zdrojem tažení
- Udělat ji cílem tažení

Vysvětlím tyto kroky odděleně. Nejprve je vhodné udělat komponentu zdrojem tažení. Jakmile tato operace funguje, přejdu k vytvoření cíle.

4.4.2 GSFTableView jako zdroj tažení

Ještě užitečnější než kopírování a vkládání souborů pomocí schránky je přesouvání respektive kopírování souborů pomocí tažení myší, a to jak v rámci aplikace, tak mezi různými aplikacemi. Nejprve vysvětlím jednodušší část, kterou je zpřístupnění aplikace jako zdroje tažení. Aby mohla být instance třídy `GSFTableView` zdrojem pro tažení, musí implementovat metodu, která se jmenuje `draggingSourceOperationMaskForLocal:` (viz ukázka 18). Tato metoda deklaruje, na kterých

operacích se chce třída podílet. Je volána vždy dvakrát: jednou je `isLocal` rovno YES, značící že operace, na které se chce trída podílet jako zdroj, kdy cíl je v rámci aplikace, a podruhé je `isLocal` rovno NO, což značí, že operace, na které se třída chce podílet jako zdroj pro jinou aplikaci.

Pro zahájení operace tažení, použiji metodu implementovanou v `NSView` a `NSWindow`, která se nazývá `dragImage:at:offset:event:pasteboard:source:slideBack:` (viz ukázka 18). Vytvořím tak obrázek, který bude zobrazen při tažení v bodu, ve kterém chci aby tažení začlo. Dokumentace říká, že by měla být volána při události `mouseDown`, ale stejně dobře funguje také urálost `mouseDragged`, kterou používám já. Pasteboard je obvykle standardní schránka používaná pro operaci drag-and-drop. Pokud se tažení neuskuteční, mohu si vybrat, zda se ikona vrátí zpět odkud přišla či nikoliv.

Také musím vytvořit obrázek, který se zobrazí při tažení. `NSImage` se používa velmi dobře, protože lze předa buď přímo hotový obrázek nejrůznějšího typu nebo lze dynamicky vytvořit pomocí standardních kreslicích operací.

Ukázka zdrojového kódu 18 Vytvoření zdroje operace pro tažení

```
- (unsigned int)draggingSourceOperationMaskForLocal:(BOOL)isLocal
{
    if ([self selectedRow] > 0){
        return NSDragOperationMove | NSDragOperationCopy | NSDragOperationLink |
               NSDragOperationDelete;
    }
    return NSDragOperationNone;
}

- (void)mouseDragged:(NSEvent *)event
{
    if ( [[self delegate] respondsToSelector:@selector(selectedFilesIndexSet:)]){
        selectedRows = [[self delegate] selectedFilesIndexSet:self];
        if ( [selectedRows count] > 0 &&
             [[self delegate] respondsToSelector:@selector(mouseDragged:from:)] ){
            [[self delegate] mouseDragged:event from:self];
        }
    }
}
```

V ukázce 18 je také uvedena metoda, která způsobí samotní spuštění akce drag and drop. Protože třída patří do části„view“, deleguje tuto akci do vrstvy „control“, s pozměněnými údaji. Tomuto delegátovi je předána ke zpracování celá událost. Tímto delegátem je samotný kontrolér okna, který předá zprávu ještě dále kontroléru pro zpracování akci (`GSFActionController`) zavoláním metody `dragFiles: withEvent:`. Z události `NSEvent` je vytažen bod, ve kterém se začne přesouvání souborů a jsou cesty k označeným souborům.

Tato část se neliší od práce se schránkou, protože se provádí zcela stejné metody. Pouze se využije schránka, která je určená pro operace drag-and-drop. Tuto schránku lze získat, když se odešle zpráva `NSPasteboard pasteboardWithName:NSDragPboard`. Vložení dat to schránky je jinak zcela stejné jako v předchozí kapitole. Definuje se typ dat jako cesty souborů a ty se posléze do schránky vloží. Narozdíl od prosté práce se schránkou je potřeba vytvořit ikonu pomocí metody

`dragImage:at:offset:event:pasteboard:source:slideBack:`

4.5 Panely nástrojů

4.5.1 Jak fungují panely nástrojů

Vytvoření panelu nástrojů, jak už to ve frameworku Cocoa bývá, je opět poměrně jednoduché, avšak vyžaduje již určitou dávku zkušeností, neboť pro začátečníka není vůbec intuitivní. K vytvoření panelu nástrojů slouží `NSToolbar` a `NSToolbarItem`, které poskytují standardní prostředky pro zobrazení panelu pod titulkem okna. Tyto třídy také poskytují uživatelům standardní způsoby pro vlastní nastavení panelu včetně uložení tohoto nastavení.

K vytvoření panelu nástrojů, je potřeba vytvořit delegáta, který poskytuje tyto důležité informace:

- Seznam základních položek panelu. Tento seznam je použit vždy při návratu k původním rozložení panelu a při jeho prvotní inicializaci.
- Seznam povolených položek panelu. Tento seznam se používá pro sestavení palety pro uživatelské úpravy panelu, pokud jsou povoleny.
- Položku panelu pro daný identifikátor.

Při vytváření `NSToolbar` se zadává jeho identifikátor. `NSToolbar` předpokládá, že všechny panely se stejným identifikátorem jsou stejné, a automaticky synchronizuje změny v nich. Pokud uživatel změní nastavení v jednom okně souborového manžera, projeví se tato změna také v ostatních oknech. Trochu lepší příklad skýtá aplikace Mail, která má dva typy oken: okno prohlížení mailů a okno pro jejich psaní. Tyto dva typy oken mají dva různé identifikátory. Pokud uživatel změní pořadí ikon v okně pro psaní, projeví se tato změna ve všech oknech pro psaní, ale nikoliv v okně pro prohlížení. Pokud je při změně v jednom okne panel nástrojů v jiném okne skrytý, zůstane skrytý i nadále, ale po zobrazení bude vypadat stejně jako panely ostatní.

Identifikátor panelu je pevně daný od vzniku objektu, ale programátor může měnit další atributy:

```
setAllowsUserCustomization:
setAutosavesConfiguration:
setDisplayMode:
```

Většina panelů nástrojů obsahuje jednoduché, klikací položky, které se chovají jako tlačítka. Nejjednodušší prvky jsou definovány ikonou, popisem, popisem pro paletu nastavení, kontextovou nápovědou, cílem a akcí. Většina panelů nástrojů může být vytvořena temito jednoduchými položkami. Pokud programátor potřebuje zobrazit složitější komponentu, zavolá `setView:` na položce panelu, která poskytne vlastní „view“ (například rozbalovací lištu nebo textové pole).

Existuje několik předdefinovaných prvků panelu:

- `NSToolbarSeparatorItemIdentifier` - standardní vertikální oddělovač
- `NSToolbarSpaceItemIdentifier` - pevná mezera
- `NSFlexibleSpaceItemIdentifier` - proměnlivá mezera
- `NSToolbarShowColorsItemIdentifier` - zobrazení panelu s barvami
- `NSToolbarShowFontsItemIdentifier` - zobrazení panelu s fonty
- `NSToolbarPrintItemIdentifier`

- and `NSToolbarCustomizeToolbarItemIdentifier` - zobrazeni palety pro nastavení panelu

Tyto prvky jsou přístupné pouze přes svůj identifikátor.

Každá komponenta panelu nástrojů `NSToolbar` je instancí třídy `NSToolbarItem`. Viditelná část se skládá z obsahu komponenty, textového popisu a menu reprezentace. Obsah může být buď `NSImage` nebo `NSView`. Tlačítka bývají zpravidla jen obrázek, ostatní komponenty (vyhledávací pole, roletky atd) jsou instancí `NSView`. Menu reprezentace je využita ve dvou případech: pokud jsou zobrazeny pouze textové popisy nebo pokud je okno příliš malé, aby mohlo zobrazit všechny komponenty najednou.

4.5.2 Vytváření panelů nástrojů

Vytvoření panelu nastrojů se provádí v několika krocích. Nejprve je potřeba vytvořit všechny komponenty, které se budou zobrazovat. Pokud se jedná o tlačítka, stačí přibalit k aplikaci obrázky, které se pak programově připojí. Jedná-li se o složitější komponenty, musí se nejprve vytvořit v Interface Builderu samostatné „view“, které se nastaví pomocí metody `setView:`. K těmto komponentám se musí napsat validátory a akce, které se budou provádět. Pokud jsou využity standardní komponenty pro barvy, fonty a tisk, musí být napsány také pro ně. Panel nástrojů vyžaduje delegáta, který musí být buď napsán v samostatné třídě nebo může být využit kontrolér okna. Já využívám kontrolér okna, ke kterému pridám kategorii, ve které jsou všechny náležitosi týkající se panelu nástrojů. Dále by měly být přidány dvě standardní položky „Hide Toolbar“ a „Customize Toolbar“ a případně jejich jazykové ekvivalenty do menu „View“. Nakonec je potřeba vytvořit objekt `NSToolbar` s identifikátorem a nastavit jeho delegáta na instanci třídy, ve které byly impementovány metody delegáta. Posledním krokem je připojení panelu nástrojů v metodě `awakeFromNib` k samotnému oknu. Já v této metodě volám `setupToolbar` (viz ukázka 19), ve které nastavím vše potřebné.

Ukázka zdrojového kódu 19 Nastavení identifikátoru panelu nástrojů, jeho konfigurace a připojení k oknu

```
- (void)setupToolbar
{
    NSToolbar *toolbar = [[NSToolbar alloc] initWithIdentifier:@"gsfFileManager"];
    [toolbar setDelegate:self];
    [toolbar setAllowsUserCustomization:YES];
    [toolbar setAutosavesConfiguration:YES];
    [[self window] setToolbar:[toolbar autorelease]];
}
```

Další ukázka 20, zachycuje dříve zmíněné metody, které pokystují senznam všech povolených identifikátorů komponent a seznam komponent v základním rozložení.

Nejsložitější na celém procesu vytváření panelu nástrojů je implementace jednotlivých komponent. Delegát musí poskytovat metodu

`toolbar:itemForItemIdentifier:willBeInsertedIntoToolbar:`

která na základě předaného identifikátoru vrátí výsledný `NSToolbarItem` připravený pro vložení do panelu. V ukázce 21 jsou dva příklady jakým způsobem se vytváří komponenta.

Jedná-li se o obyčejné tlačítko, stačí definovat pouze obrázek, který je přibalen v nib souboru. To se provede přetažením obrázku do Interface Builderu a jeho hlavního okna. Tento obrázek by se měl automaticky objevit na záložce „Images“. Metodou `setImage:` se přiřadí `NSImage` vytvořený pomocí konstruktoru `imageNamed:`, jemuž se předá název obrázku bez koncovky.

Ukázka zdrojového kódu 20 Metody vracející seznam povolených položek a vzhled základní konfigurace panelu nástrojů

```
- (NSArray *) toolbarAllowedItemIdentifiers: (NSToolbar *) toolbar {

    return [NSArray arrayWithObjects: NSToolbarSeparatorItemIdentifier,
        NSToolbarSpaceItemIdentifier, NSToolbarFlexibleSpaceItemIdentifier,
        NSToolbarCustomizeToolbarItemIdentifier,   @"NavigationItem",
        @"ReloadItem", @"BookmarkItem", @"MountedVolumesItem",
        @"CompressItem", @"InfoItem", nil];
}

- (NSArray *) toolbarDefaultItemIdentifiers: (NSToolbar *) toolbar {
    return [NSArray arrayWithObjects: @"NavigationItem",  @"ReloadItem",
        NSToolbarSeparatorItemIdentifier, @"InfoItem", @"BookmarkItem",
        @"MountedVolumesItem", NSToolbarSeparatorItemIdentifier,
        @"CompressItem", NSToolbarFlexibleSpaceItemIdentifier, nil];
}
```

Ukázka zdrojového kódu 21

```
- (NSToolbarItem *) toolbar:(NSToolbar *)toolbar
      itemForItemIdentifier:(NSString *)itemIdentifier
  willBeInsertedIntoToolbar:(BOOL)flag
{
    NSToolbarItem *toolbarItem = [[NSToolbarItem alloc] initWithItemIdentifier: itemIdentifier];
 if ( [itemIdentifier isEqualToString:@"BookmarkItem"] ) {
        [toolbarItem setLabel:@"Bookmarks"];
        [toolbarItem setPaletteLabel:[toolbarItem label]];
        [toolbarItem setImage:[NSImage imageNamed:@"bookmarks"]];
        [toolbarItem setTarget:self];
        [toolbarItem setAction:@selector(showBookmarks:)];
    }

    else if ( [itemIdentifier isEqualToString:@"MountedVolumesItem"] ) {
        NSRect fRect = [mountedVolumesItemView frame];

        [toolbarItem setLabel:@"Mounted Volumes"];
        [toolbarItem setPaletteLabel:[toolbarItem label]];
        [toolbarItem setView:mountedVolumesItemView];

        [toolbarItem setMinSize:fRect.size];
        [toolbarItem setMaxSize:fRect.size];

    }
return [toolbarItem autorelease];
```

Pokud je potřeba složitější komponenta, jako například rozbalovací menu, je potřeba ji v Interface Builderu nejprve vytvořit. Do nib souboru okna, ve kterém se bude nástrojů používat se vloží `NSView` přímo do instancí objektů v hlavním okně. Měl by se zobrazit prázdný `NSView` (okno bez okrajů a titulku), do kterého lze vložit libovolnou komponentu. Velikost tohoto „view“ podle toho jaká je velikost komponenty uvnitř. Doporučuji velikost nastavit ručně vyplněním hodnot v inspektoru, protože natahováním myší se nemusí vždy podařit udělat „view“ dostatečně velké aby neořezávalo stín vloženého objektu a zároveň nezabírala přiliš mnoho místa v panelu nástrojů.

Tento „view“ musí být napojený na outlet v delegátu pro panel nástroju. Ja mám tedy tři outlety v `GSFMainWindowController`, které jsou napojeny na tři `view`. Komponenta se přiřadí zasláním zprávy `setView:`, jehož parametrem je outlet směřující na požadovaný „view“.

Tím byla vytvořena vizuální podoba komponenty a nyní je potřeba určit jaká metoda se provede při jejím stisknutí. K tomu slouží metody `setTarget:` a `setAction:`, které nastaví objekt, kterému bude odeslána zpráva při stisknutí a selector na metodu, která se má provést.

Před předaním komponenty panelu nástrojů ji musí být odeslána zpráva `autorelease`, aby se zajistilo správné uvolnění, až nebude potřebná.

K dočasnému zakázání nebo povolení komponenty slouží takzvaná validace. Delegát panelu nástrojů musí implementovat metodu `validateToolbarItem:`, která vrací YES nebo NO, podle toho zda je daná komponenta momentálně k dispozici či nikoliv.

Práce s panelem nástrojů je jiná, než se kterou jsem se dopostud setkal, nicméně je velmi jednoduchá. Díky tomu, že programátor vytváří pouze ty komponenty, které nejsou tlačítkem, a ostatním přiřadí jen ikonu, lze velmi jednoduše přidávat nové ovládací prvky. Stačí ji přidat do metody vracející seznam všech komponent, přiřadit cíl a metodu kam bude komponenta napojena.

4.6 Napojení na síťové služby

4.6.1 Využívání skriptů v aplikacích

Tato kapitola čerpá z [T+05].

Jedním bodem zadaní diplomové práce je prověření možností napojení souborového manažera na síťové funkce operačního systému MAC OS X. Aplikace vytvořená v rámci této diplomové práce umožňuje využít prostředky pro připojení sdílení MS Windows a FTP serverů s využití programů dostupných v MAC OS X. Abych mohl použít tyto programy, musel jsem umožnit spouštění externích programů v rámci mého projektu. Existuje několik způsobů jak toho dosáhnout. Lze využít C funkcí `system` a `popen` nebo třídy NSTask poskytovanou frameworkem Cocoa.

Prvně uvedená funkce `system` je patrně nejjednodušším způsobem jak spustit subproces z Cocoa aplikace. Jako parametr se předává jediný C řetězec ukončený znakem `null` a vrací celočíselný chybový kód - *e*rror code . Tento řetězec je jednoduše shellový skript, který bude spuštěn pomocí `bash` shellu. Chybový kód je návratová hodnota tohoto shellového skriptu obsahující 0, pokud skript proběhl bez chyby. V ukázce 22 je uveden příklad použití této funkce. Provede se výpis adresáře `/usr/bin` do souboru na ploše.

Ukázka zdrojového kódu 22 Ukázka použití funkce system

```
int exitCode = system("/bin/ls /usr/bin > ~/Desktop/obsah_slozky.txt");
if ( !exitCode )
  NSLog(@"Probehlo vporadku");
else
  NSLog(@"Pri provadeni doslo k chybe %d", exitCode);
```

Přestože standardně `system` používá Bash shell k provádění skriptů, neznamená to, že nelze využít jiný skriptovací jazyk, protože lze spustit nejprve tento jazyk a jemu předat spuštěný skript. Stejně tak lze spustit i jiný kompilovaný program.

Nevýhodou funkce `system` je, že pozastaví provádění Cocoa aplikace dokud není ukončena a dále výstupní data lze převzít pouze pomocí došasného souboru, do kterého je lze zapsat. Pro asynchronní spouštění je vhodná funkce `popen`, která otevře standardní rouru a vrátí ukazatel na C `FILE`, do kterého lze zapisovat i zněj číst. Tento `FILE` je připojen na standardní vstup a standardní výtup spuštěného subprocesu. Použití je velmi podobné jako u funkce `system`, jen je funkce ukončena ihned po spuštění subprocesu. Provádění tak probíha asynchronně a umožnuje tak předávat data spuštěnému procesu.

Framework Cocoa také poskytuje řadu tříd pro vytváření subprocesů a komunikaci s nimi. Nejvýznamnějším je `NSTask`, který ve své nejprostší podobě může být použit stejně jednoduše jako `system`. Umožňuje také využití tříd `NSPipe` pro komunikaci s vytvořeným procesem a vytvářet tak složitější konstrukce s využitím Cocoa frameworku. Popis těchto tříd je nad rámec diplomové práce a zájemcům doporučuju prostudovat knihu [T+05] kapitolu 13 Using Scripts Within Applications, připadně dokumentaci k těmto třídám včetně návodu jejich použití, jež je v dokumentaci také obsažen.

Pro spuštění příkazů pro připojení síťových disků jsem použil funkci `system`, protože pro ukázku spouštění aplikací postačuje.;

4.6.2 Připojení oddílů pomocí Samby

Tento souborový manažer umožňuje připojit sdílený disk a zpřístupnit jej tak v celém systému. Využívá dostupných prostředků operačního systému Mac OS X. Je využita aplikace *mount_smbfs*, která připojí sdílený zdroj ze SMB souborového serveru na zvolenou cestu. Úspěch a neúspěch připojení je tedy zcela v režii operačního systému.

Připojování probíhá v metodě nazvané `connectServerWithURL:`, která očekává jako parametr `NSURL`. Metoda je uvedena v ukázce 23. Rozhodnutí jaké prostředky se využijí pri připojení k serveru se dělá na základě schématu, které je součástí objektu `NSURL`. Pro připojení k sambě musí být uvedeno jméno heslo a server, jinak dojde k chybě. Nepodařilo se mi implementovat dialog, který by se dotázal na jméno a heslo až po připojení, tak jak to dělá *mount_ftp*, a proto musí být zadány jako parametr při spuštění příkazu pro mountování.

Bohužel připojení do windows sdílení není na Mac OS X vždy bez problémů, protože může ztroskotat na nastavení způsobu ověřování. K některým serverům se mi nepodařilo připojit ani s využitím Finderu, který pro mountování používá stejné prostředky. Na konci metody je odeslání notifikace o tom, že se změnily namountované svazky a je potřeba aktualizovat jejich výpis.

4.6.3 Připojení na FTP servery

Připojení na FTP servery je také prozatím implementováno s využitím prostředků operačního systému. Ten poskytuje příkaz *mount_ftp*, kterým připojí obsah FTP serveru do zvolené složky. Obsah je tak transparentní v celém systému, ale složka je pouze pro čtení a neumožňuje přímou editaci souborů ani jejich uploadování na server. Toto omezení je způsobeno charakteristikou FTP protokolu, který neumožňuje sekvenční přístup k souborům, ale pouze jejich transfer.

Utilita *mount_ftp* sama zobrazí dialog, ve kterém se zeptá na zadání jména a hesla, pokud nebylo připojení se zadanými parametry úspěšné. Další nevýhoudou je, že neumožňuje anonymní připojení na server, a proto musí být vždy zadáno nějaké jméno a heslo.

Ukázka zdrojového kódu 23 Implementace metody connectServerWithURL:(NSURL *) pro připojení k serveru

```
- (void)connectServerWithURL:(NSURL *)url
{
    NSString *mount_command;
     if (url != nil){
         NSString * directoryName;
         NSString * mountingPath = @"/Volumes";
         NSString * tmpDirectoryName;

         if ( [[url path] length] > 2 ){
             tmpDirectoryName = [[[url path] lastPathComponent] uppercaseString];
         }
         else {
             tmpDirectoryName = [[url host]  uppercaseString];
         }
         directoryName = [NSString stringWithString:tmpDirectoryName];


         int i = 1;
         while( [[NSFileManager defaultManager] fileExistsAtPath:
                    [mountingPath stringByAppendingPathComponent:directoryName]] ){
             directoryName = [NSString stringWithFormat:@"%@_%d", tmpDirectoryName, i];
             i++;
         }

         [[NSFileManager defaultManager] createDirectoryAtPath:
                    [mountingPath stringByAppendingPathComponent:directoryName] attributes:nil];

         if ( [[url scheme] isEqual:@"smb"] || [[url scheme] isEqual:@"cifs"] ){
             mount_command = [NSString stringWithFormat:@"/sbin/mount_smbfs '//%@:%@@%@%@' %@/%@",
                                 [url user],
                                 [url password],
                                 [url host],
                                 [url path],
                                 mountingPath, directoryName];
         }
         else if ( [[url scheme] isEqual:@"ftp"] ){
             long port = [[url port] longValue]>0?[[url port] longValue]:21;
             mount_command = [NSString stringWithFormat:@"/sbin/mount_ftp %@:%d/%@ %@/%@",
                                 [url host],
                                 port,
                                 [url path],
                                 mountingPath, directoryName];
         }
         if( mount_command != nil ) {
             system([mount_command cString]);
             [[NSNotificationCenter defaultCenter] postNotificationName:@"GSFMountedVolumesDidChange"
                                                                 object:self];
         }
     }
}
```

Díky modulárnímu návrhu, lze do souborového manažera doplnit i kvalitní FTP klient, který nebude závislý na aplikacích, které jsou obsaženy v operačním systému. Implementace FTP klienta, který bude schopný pracovat aktivně i pasivně, s využitím připojení přes proxy a anonymního připojení, tedy základních požadavku kladených na dobrý program, by vystačilo na samostatntou diplomovou práci. Ve vývoji však budu pokračovat i po obhájení práce a FTP je na předních pozicích seznamu věcí k implementaci.

Kapitola 5

Závěr

Ačkoliv v České republice nejsou počítače Apple příliš rozšířeny, velmi jsem si je oblíbil a rozhodl jsem se seznámit s vývojem aplikací pro tuto platformu. Učení programovacího jazyka a používaných frameworků se neobejde bez praktického vyzkoušení prostudované problematiky Pro tyto účely jsem si zvolil vytvoření souborového manažera. Produkt této diplomové práce není primárně určen dlouholetým uživatelům macintoshů, kteří jsou zvyklí na poměrně uspokojivou práci s Finderem případně jeho rozšířeními, ale měl by usnadnit přechod z jiné platformy a nahradit tak novým uživatelům jejich oblíbený program. Dále by měl ulehčit práci uživatelům notebooků, kteří nebudou muset tak intenzivně používat myš jako je tomu zapotřebí u Finderu.

Vytvořit kvalitní souborový manažer je během na dlouhou trať. Všechny běžně používané manažery jsou vyvíjeny svými autory již po dobu několika let a mají za sebou nespočet verzí. Program vznikal inkrementálním způsobem, kdy jsem postupně programoval jednotlivé části, jak jsem získával nové znalosti o Objective-C a Cocoa. To si bohužel velmi často vyžádalo několikeré přepsání původních návrhů, protože s nově nabytými informacemi jsem je mohl naprogramovat lépe a efektivněji. Výsledný produkt není tedy zdaleka finální verzí pro použití běžnými uživateli, ale představuje základ, na kterém budu dále stavět a vytvářet souborový manažer se všemi funkcemi, které uživatelé očekávají.

Svou diplomovou práci jsem pojal převážně jako studium dostupných nástrojů a programovacích technik. Díky složitosti programu jsem musel nastudovt mnoho programovacích technik, které jsem teď nyní schopen uplatnit v praxi. Pokud někdo programoval pro jinou platformu, není přechod na Cocoa jednoduchý, protože ke známým věcem se přistupuje výrazně jinak, ovšem mnohem jednodušeji, což činí programování pro Mac OS X velmi zajímavým K tomuto účelu jsem prostudoval tři knihy, které popisují do hloubky jazyk Objective-C [Dav02], framework Cocoa [Hil05] a široce pojatou pulikaci popisující obecně programování v Mac OS X [T+05]. Každému zájemci o vytváření programů pro Mac OS X mohu jen doporučit vytrvat v počátečním úsilí, které je potřeba překonat, než je člověk schopný vytvořit první provozuschopnou aplikaci.

Přínos této diplomové práce spočívá v odkrytí základů programování pro Mac OS X naprostým začátečníkům a zahájení vývoje souborového manažera, který je sice běžným na ostatních platformách (MS Windows i Linux), ale odpovídající ekvivalent pro počítače macintosh prozatím chybí.

Literatura

[Dav02] James Duncan Davidson. *Learning Cocoa with Objective-C*. O'Reilly, 2002. ISBN 0-596-00301-3.

[Hil05] Aaron Hillegass. *Cocoa programming for Mac OS X*. Addison-Wesley, 2005. ISBN 0-321-21314-9.

[stra] WWW stránky. Mac os 8 - wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/Mac_os_8`.

[strb] WWW stránky. Mac os 9 - wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/Mac_os_9`.

[strc] WWW stránky. Mac os x - wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/Mac_OS_X`.

[strd] WWW stránky. Mach (kernel) - wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/Mach_(kernel)`.

[stre] WWW stránky. Macintosh finder - wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/Macintosh_Finder`.

[strf] WWW stránky. System 6 - wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/System_6`.

[strg] WWW stránky. System 7 - wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/System_7_(Macintosh)`.

[strh] WWW stránky. Xnu - wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/XNU`.

[T+05] Michael Trent et al. *Begginig Mac OS X Programming*. Wrox, 2005. ISBN 0-7645-7399-3.