České vysoké učení technické v Praze

Fakulta elektrotechnická

Diplomová práce

Komplexní víceuživatelský plánovací kalendář s vysokou úrovní zabezpečení

Bc. Vojtěch Outulný

Vedoucí práce: Ing. Pavel Kubalík, Ph.D.

Studijní program: Elektrotechnika a informatika strukturovaný magisterský

Obor: Informatika a výpočetní technika

Červen 2008

Poděkování

Rád bych poděkoval vedoucímu diplomové práce, panu Ing. Kubalíkovi, Ph.D., za jeho vynaložený čas a spolupráci při zdokonalování samotné aplikace. V neposlední řadě chci také poděkovat svým rodičům, kteří mě při vzniku této diplomové práce podpořili.

Prohlášení

Prohlašuji, že jsem svou diplomovou práci vypracoval samostatně a použil jsem pouze podklady uvedené v přiloženém seznamu.

Nemám závažný důvod proti užití tohoto díla ve smyslu §60 Zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon).

V Praze dne 23. 5. 2008 …….

Abstract

The thesis deals with design and implementation of application for time scheduling and management of contacts. The resulting system is fully functional three layer system which enables creating and controlling of time-schedule, contacts notes and fast obtaining of information about current happening. The thesis was developed for VLSI research group at the Department of Computer Science and Engineering on the FEE of the CTU in Prague for purpose of time management of its members.

This work adds possibility of creating of schedule nature way regarding on usability. Application enables access to data from KOS (Study Information System) and so users can work with their school schedule in application. System supports simply sharing of time-schedules between users within user defined groups. Application guarantees high security. On base of open and layered architecture the system can be develop in future and individual layers can be changed for new. Application is platform independent so it can run on any platform.

Abstrakt

Diplomová práce se zabývá návrhem a implementací aplikace pro správu času a kontaktů. Výsledkem práce je plně funkční aplikace založená na třívrstvé architektuře, která umožňuje tvorbu a správu časových plánů, kontaktů, poznámek a rychlé získání informací o aktuálním dění. Projekt vznikl na základě požadavků výzkumné skupiny VLSI na Katedře počítačů FEL, ČVUT v Praze za účelem koordinace času jejích členů.

Tato práce přináší možnost tvorby kalendáře zcela přirozenou a snadnou formou s ohledem na usabilitu. Výsledná aplikace poskytuje přístup k datům z KOSu (Komponenta Studia), a dovoluje tak pracovat se školním rozvrhem v rámci aplikace. Celý systém navíc podporuje snadné sdílení časových plánů mezi uživateli v rámci uživatelsky definovaných skupin. Zároveň je zaručena vysoká míra zabezpečení. Na základě otevřené a vrstevnaté architektury je možné systém dále rozvíjet a jednotlivé vrstvy nahrazovat za nové. Platformní nezávislost prezentační vrstvy dovoluje nasazení systému na celé řadě míst bez ohledu na danou platformu.

Obsah

Seznam obrázků

1 Úvod

Umět si správně zorganizovat čas a mít kontakty na lidi vždy po ruce je dnes jeden ze základních pilířů úspěchu. Aplikace Víceuživatelský plánovací kalendář (MUSC) slouží k snadné a rychlé tvorbě časových plánů, kontaktů a poznámek. Současně aplikace poskytuje rychlý přístup k informacím o aktuálním dění prostřednictvím RSS čtečky či vyhledávání na internetu.

S ohledem na globalizaci světa získává stále více na významu možnost sdílet a následně koordinovat kalendáře mezi jednotlivými uživateli. Aplikace umožňuje snadné sdílení časových plánů mezi uživateli v rámci uživatelsky definovaných skupin. Je umožněna tvorba tzv. veřejných skupin, do kterých se mohou uživatelé přihlašovat samy, a tzv. neveřejných skupin, které jsou plně pod kontrolou administrátora, jenž rozhoduje o členství.

Kalendář umožňuje přístup k datům z KOSu, a tak uživatelé mohou v rámci programu pracovat s vlastním školním rozvrhem nebo si prohlédnout rozvrh všech členů dané skupiny a zkoordinovat tak své aktivity. Design klientské aplikace byl navržen s ohledem na usabilitu, tak aby ovládání bylo nejen snadné a rychlé, ale hlavně přirozené a intuitivní.

Cílem práce je klasická třívrstvá aplikace – klient, server, databáze. Klientská část bude tvořena desktopovou aplikací, serverová část aplikačním serverem a databáze bude realizována s využitím databázového serveru. Na základě otevřené a vrstevnaté architektury bude možné systém v budoucnu dále rozvíjet a jednotlivé vrstvy měnit bez ohledu na použitou technologii. S využitím jazyka Java, jak na klientské části tak i na serverové, bude dosažena platformní nezávislost celého systému. Jazyk Java nám navíc umožní definování jednotného vzhledu na všech platformách.

Aplikace poskytuje vysokou míru zabezpečení. Základním kamenem je zabezpečení účtu uživatelským jménem a heslem. Tento základ je dále rozvinut dále např. prostřednictvím zabezpečení komunikace, kdy veškerá komunikace mezi klientskou částí a serverovou je šifrována.

S ohledem na roztříštěnost trhu s programy s podobným zaměřením, bude aplikace podporovat export a import kalendáře do formátu iCalendar (RFC 2445) a kontaktů do formátu vCard (RFC 2426).

1.1 Členění textu

Druhá kapitola tohoto textu přináší přehled stávajících řešení v oblasti softwaru pro plánování času a správu kontaktů jak v desktopové, tak i ve webové podobě.

Třetí kapitola se pak zabývá důvody pro vznik této práce a stanovuje cíle kladené na tento projekt.

Čtvrtá kapitola se pak zabývá analýzou dané domény. Součástí této kapitoly je analýza významných součástí systému a návrh jejich možných řešení.

Následující kapitola se pak zabývá popisem implementace specifických částí, které jsou podstatné z pohledu celého systému.

Šestá kapitola se pak zaměřuje na otázku bezpečnosti jak uvnitř samotné aplikace, tak i na bezpečnost s ohledem na okolí systému.

Předposlední sedmá kapitola se pak zabývá testováním multiplatformnosti a uživatelského rozhraní.

V poslední osmé kapitole jsou shrnuty dosažené výsledky a jsou nastíněny možné směry dalšího rozvoje aplikace.

2 Popis stávajících řešení

V současnosti existuje velké množství aplikací zabývající se stejnou nebo podobnou problematikou. Z tohoto důvodu by bylo velmi nerozumné nepokusit se poznat konkurenci a vymýšlet všechny koncepty od začátku. Pokusil jsem se tedy zmapovat několik nejznámějších produktů. Tento průzkum trhu mi nejen pomůže zjistit služby poskytované ostatními, ale především mi umožní odhalit slabá místa těchto aplikací, což mi dovolí nabídnout nové služby či zlepšit již stávající. V současnosti se trh rozpadá na dvě velké skupiny – desktopové a webové aplikace. Pokusím se tedy zmapovat obě dvě tyto skupiny.

2.1 Rešerše desktopových kalendářů

Desktopové kalendáře jsou tu s námi od samotného počátku GUI. Tyto aplikace mohou plně využít potenciálu současných technologií a poskytnout uživateli příjemný způsob tvorby kalendáře s ohledem na jeho přání. Desktopové kalendáře vždy sloužili k plánování jednotlivců, dnes je však celý svět propojen internetem, a tak je i na ně kladen požadavek na sdílení kalendářů. Synonymem pro desktopové kalendáře je dnes určitě MS Office Outlook, proti němu jsem postavil Open Sourcový Mozilla Sunbird.

2.1.1 Microsoft Office Outlook 2007

MS Outlook 2007 je program, který neslouží pouze jako plánovací kalendář, ale také jako správce kontaktů, úkolů, poznámek a v neposlední řadě také jako e-mailový klient. Jedná se čistě o komerční projekt, který je svázán s platformou MS Windows. Jak vypadá GUI MS Outlook 2007, je vidět na obrázku 1.

Po levé straně programu je svislý panel, který poskytuje jednoduché navigační menu. Zde může uživatel nalézt nejčastěji používané služby *„Outlook dnes“*, kalendář, kontakty, úkoly, poznámky a poštu.

Služba *„Outlook dnes“* poskytuje rychlý přehled akcí zaznamenaných do kalendáře a úkolů pro aktuální den. Je zde vždy seznam všech akcí a úkolů, na které je možné najet kurzorem myši a po kliknutí na ně se otevře okno pro editaci příslušné akce či úkolu.

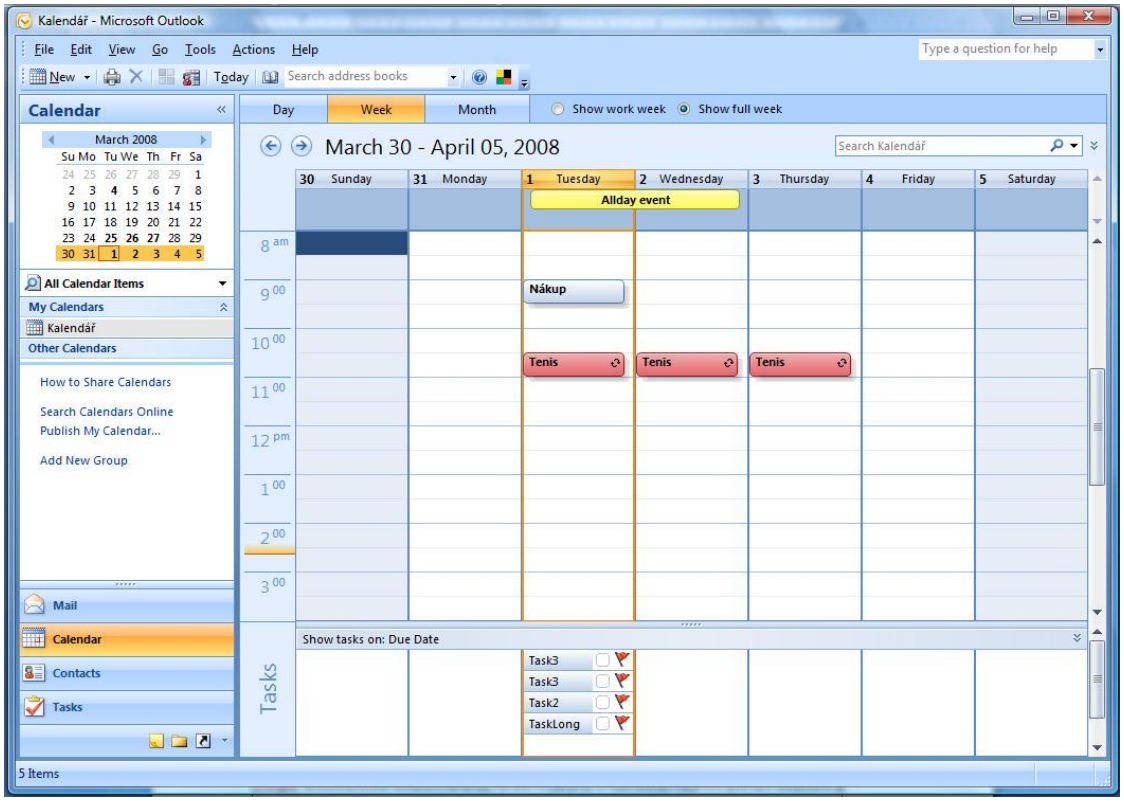
Na záložce kalendář lze nalézt tři části. Po levé straně panel s mini kalendářem a seznamem kalendářů, na pravé straně pak plocha pro zobrazení časových plánů a ve spodní části aplikace seznam s úkoly pro aktuální den. Kalendář na plánování akcí zobrazuje akce minimálně s 30 minutovými intervaly trvání. V jednotlivých pohledech lze s využitím DnD (Drag and Drop) události přesouvat mezi dny nebo měnit jejich počátek a konec. Po kliknutí na akci se zobrazí okno pro editaci akce. Do kalendáře lze umístit více souběžně probíhajících akcí. Tyto akce se pak zobrazují vedle sebe a jejich velikost se přizpůsobuje vzhledem k pevně určené šířce tabulky. V dialogu pro editaci akce lze určit několik položek pro každou událost. U akce lze určit titulek (text následně zobrazený v plánovacím kalendáři), místo konání, počáteční datum a čas, konečné datum a čas akce a nakonec samotný text akce. Datum u akcí lze určit dvěma způsoby, buď vepsat datum ručně, nebo s využitím kalendáře. Čas u akcí lze volit pouze po předem daných půlhodinových blocích. U každé akce lze určit, zda se má akce

opakovat či ne. Lze si vybrat, jestli se má akce opakovat denně, týdně, měsíčně nebo ročně. Pro každé opakování lze určit rozsah opakování. Tento rozsah lze určit jako neomezený, konečným datem anebo počtem opakování. Na každou akci je možné se nechat upozornit. Outlook umožňuje čtyři základní pohledy na plán – denní, pracovní týden (pondělí až pátek), týdenní (pondělí až neděle) a měsíční. Aplikace umožňuje akce třídit do skupin, kdy každé skupině lze určit barvu. Vyhledávání akcí je možné podle názvu nebo textu v nich obsaženém. Vybráním dne v měsíčním pohledu se změní pohled na příslušný den. Poslední samostatnou částí je panel úkolů, který zobrazuje úkoly pro aktuální den. Tyto úkoly lze editovat po kliknutí na ně.

Na nadcházející událost je možné se nechat upozornit alarmem. Outlook podporuje export a import kalendáře do iCalendar nebo CSV (Comma-separated values) a kontaktů do vCard. Kalendář je možné sdílet třemi způsoby – zveřejnění na Microsoft Office Online, prostřednictvím e-mailu nebo s využitím Microsoft Exchange server.

Na záložce s kontakty lze snadno vytvářet kontakty na osoby. Po kliknutí se zobrazí dialog pro vytvoření nového kontaktu. U každého kontaktu lze vyplnit všechny potřebné informace pro danou osobu. Kontakty lze třídit podle abecedy do jednotlivých sekcí.

Další službou je tvorba úkolů. Záložka je tvořena jednou tabulkou obsahující seznam úkolů. Úkoly lze snadno vytvořit pomocí kliknutí do tabulky. Samotné úkoly jsou v podstatě velmi podobné akcím v kalendáři. Úkoly jsou vlastně akce s určení trvání s přesností na dny. U každého úkolu je třeba zadat datum zahájení a datum splnění. Ostatní položky jsou obdobné jako u akce v plánovacím kalendáři.



Obrázek 1: MS Office Outlook 2007

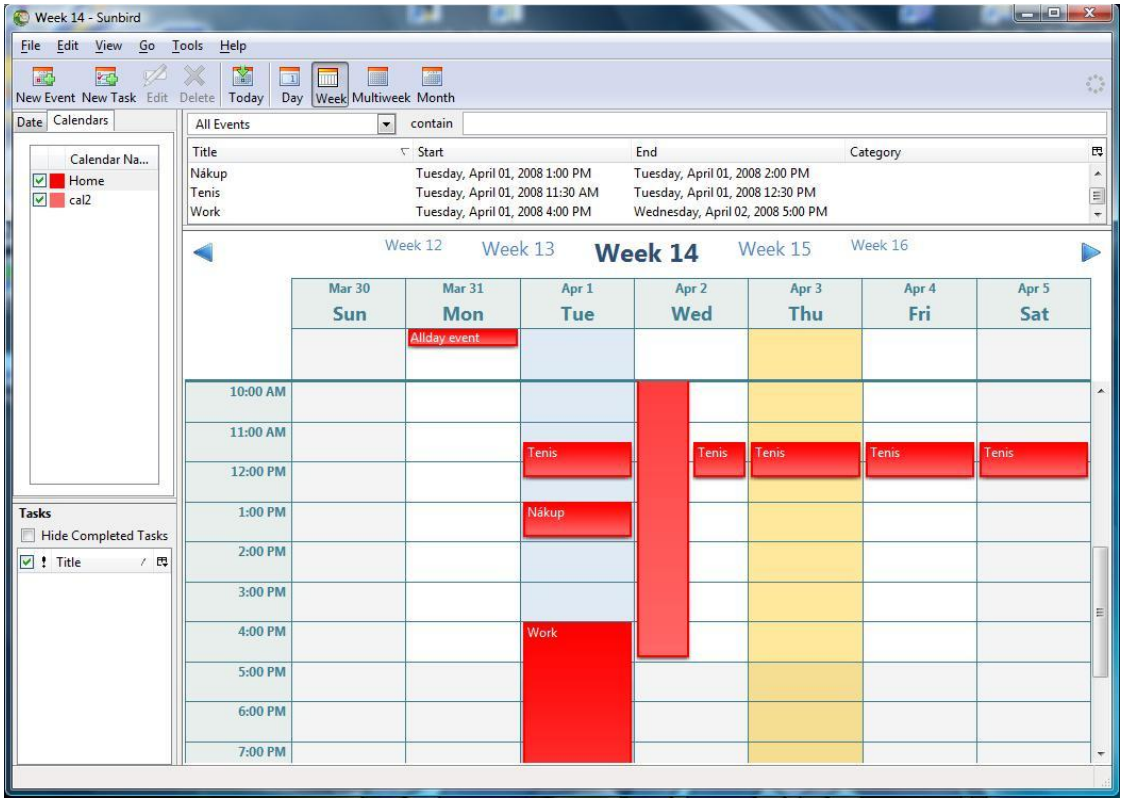
Poslední sekcí je možnost tvorby poznámek. Jedná se o možnost zaznamenání textu (poznámky) bez jakéhokoliv bližšího časového určení. Poznámky jsou reprezentovány ikonami, kdy pro jejich přečtení je nutné je otevřít a není zde tak možnost rychlého náhledu na jejich obsah.

2.1.2 Mozilla Sunbird 0.8

Aplikace Mozilla Sunbird je Open Sourcový projekt, který je navíc platformě nezávislý. Na první pohled působí program přehledně a intuitivně. Je zde nabízena možnost tvorby událostí a úkolů. Plocha je rozdělena do tří částí. Vlevo je panel s mini kalendářem nebo se seznamem kalendářů. Pod ním je seznam úkolů. V pravé části je pak samotný kalendář a nad ním je seznam událostí, které lze třídit dle kriterií. Jak vypadá GUI Mozilla Sunbird 0.8, můžete vidět na obrázku 2.

Kalendář lze zobrazit v denním, týdenní, více týdenním a měsíčním pohledu. Události je možné vytvářet s přesností na pět minut, ale v pohledech lze docílit maximální přesnosti 15 minut. Pohledy podporují DnD mezi dny a je možné měnit počátek a konec události s 15 minutovým skokem.

Opakující události nejsou nijak speciálně označeny v pohledech. Barva události je určena barvou kalendáře. V programu je také možné kromě událostí vytvořit úkoly. Pokud je úkolu přiřazen konkrétní čas, tak je zobrazen v pohledu společně s událostmi, od kterých však není žádným způsobem odlišen. Na nadcházející se událost může být uživatel upozorněn alarmem nebo zasláním e-mailu.



Obrázek 2: Mozilla Sunbird 0.8

Sunbird umožňuje zveřejnění kalendáře na webový server nebo jeho sdílení prostřednictvím iCalendar či CalDAV. Aplikace podporuje export a import do formátu iCalendar a CSV.

2.2 Rešerše webových kalendářů

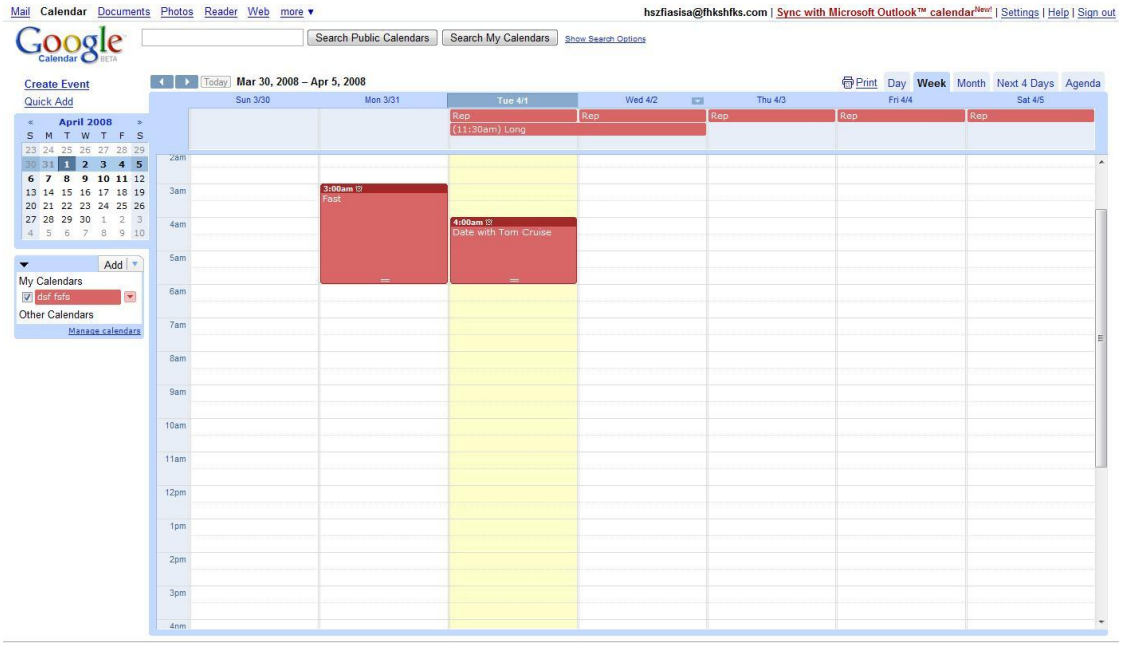
Webové aplikace jsou dnes velmi oblíbeným tématem a již celá řada aplikací se z desktopu přesunula do prostředí internetového prohlížeče. Tento přesun je dnes teprve na počátku a hlavní boom nás teprve čeká, přesto však již dnes lze nalézt povedené příklady. Cílem této rešerše není hodnotit omezení daná technologií, ale zhodnotit nabízené funkce, intuitivnost ovládání a možnosti správy časových plánů. Zaměřil jsem se na pravděpodobně největší hráče na tomto poli a to Google Calendar, Windows Live Calendar a Yahoo Calendar.

2.2.1 Google Calendar

Bez pochyby nejznámějším webovým kalendářem je Google Calendar. Služba je nabízena zatím v anglickém jazyce, což se ale patrně v blízké budoucnosti změní a služba bude lokalizována do českého jazyka. Prostředí je na první pohled intuitivní a přehledné. Jak vypadá GUI Google Calendar, můžete vidět na obrázku3. Kalendář umožňuje DnD mezi dny, dále lze událostem pomocí DnD měnit konec, nelze však již měnit začátek. Každý uživatel si může vytvořit více kalendářů, které lze barevně odlišit, projeví se na barvě události, a snadno tak oddělit pracovní události od soukromých.

Google Calendar umožňuje sdílení kalendáře s kýmkoliv nebo s určitou skupinou uživatelů. Pokud jste již dříve používali nějaký jiný program, můžete své události importovat ve formátu iCalendar nebo CSV. Naopak exportovat kalendář lze také do formátu iCalendar či RSS (Really Simple Syndication). Aplikace zjevně pracuje na principu technologie Ajax (Asynchonous JavaScript and XML).

Kalendář nabízí pět režimů zobrazení – denní, týdenní, měsíční, následující čtyři dny a agenda. Pohled agenda je seznam událostí dle data. Implicitní pohled po přihlášení lze nastavit. Po levé straně aplikace je zobrazen mini kalendář, který umožňuje rychlé listování. Nové události lze rychle vytvářet přes tlačítko *„Quick Add“*. Plnohodnotné události pak přes tlačítko *„Create Event“*. Obě tlačítka jsou poněkud skryta a nejsou hned na očích viditelná. Čas události lze určit s krokem 30 minut. U události lze nastavit opakování, a to neomezeně či do konkrétního data, chybí možnost pomocí určení počtu opakování. V kalendáři bohužel není nijak zvýrazněno, že se jedná o opakující se událost. Pokud uživatel vytvoří celodenní událost nebo událost trvající více dní, je umístěna nad samotný kalendář, což přispívá k zvýšení přehlednosti. U sdílené události je možné určit, zda bude v rámci skupiny viditelná či nikoliv. V kalendáři je možné vyhledávat. Na blížící se událost se může uživatel nechat upozornit pop-up oknem, SMS či e-mailem.



Obrázek 3: Google Calendar

2.2.2 Windows Live Calendar

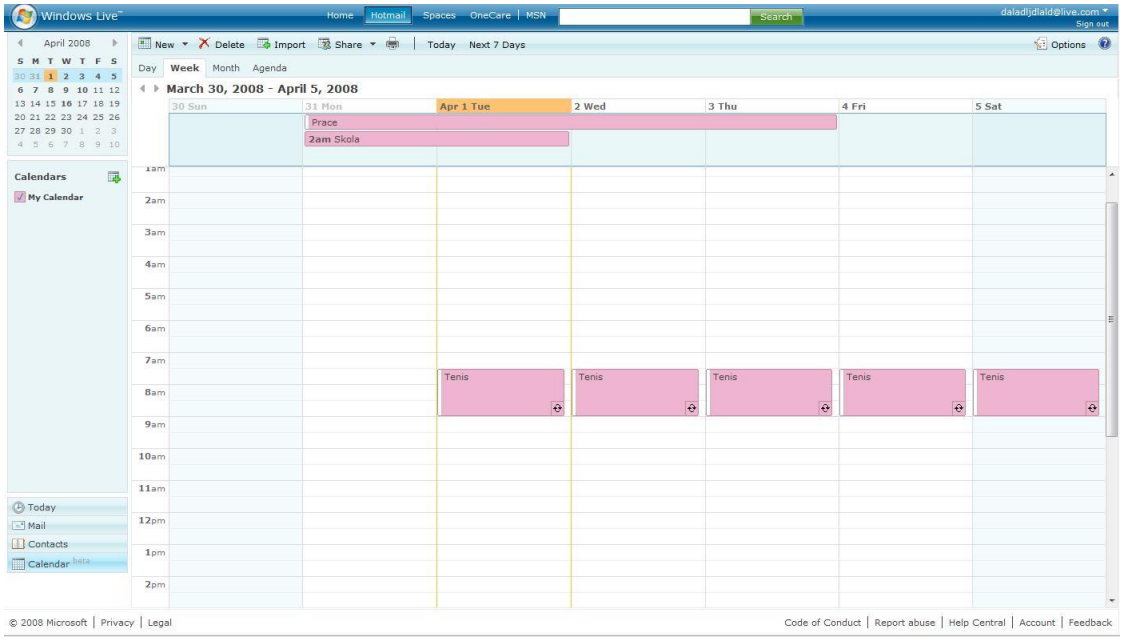
Aplikace po prvním spuštění působí velmi moderním dojmem a po grafické stránce pěkně zapadá do portfolia služeb portálu Live. Jak vypadá GUI Windows Live Calendar, můžete vidět na obrázku 4. Bohužel pod líbivým kabátkem se skrývá zastaralá aplikace, kde se její neohrabanost projevuje příliš častými opětovnými nahráváními téměř při každém kliknutí. Samotná práce s kalendářem působí velmi pomalím dojmem, zvláště při přepnutí mezi pohledy i při zcela prázdném kalendáři nebo DnDu. Kalendář také není lokalizován do češtiny.

V rámci účtu lze vytvářet libovolné množství kalendářů. Je zde podpora pro DnD mezi dny, ale i změna počátku a konce události. Po kliknutí v menu na *„New“* → *„Event“*, je uživateli nabídnuto vytvoření jednoduché akce, Ta je však stále složitější než *„Quick Add“* u Google Calendar, která pouze obsahuje počátek a popisek události. Po kliknutí na *„Add more details“* může uživatel specifikovat další informace. Kalendář nabízí denní, týdenní, měsíční pohled a agendu.

Události lze vytvářet opět s půl hodinovým krokem. U opakovaní lze naproti Google Calendar určit i počet opakování. Opakující se události jsou v kalendáři odlišeny pomocí ikony. Celodenní a více denní události jsou opět zobrazeny v samostatném pruhu nad kalendářem. Bohužel barva události je určena barvou kalendáře.

Možnosti sdílení jsou opravdu velmi velké. Kalendář lze publikovat, tak aby k němu měl přístup každý, nebo ho může zpřístupnit pouze určité skupině lidí. Dále si lze u každé události určit, zda bude viditelná či nikoliv ve sdíleném kalendáři.

Na blížící se událost se může uživatel nechat upozornit e-mailem nebo MSN Alerts. Kalendář umožňuje import ve formátu iCalendar.



Obrázek 4: Windows Live Calendar

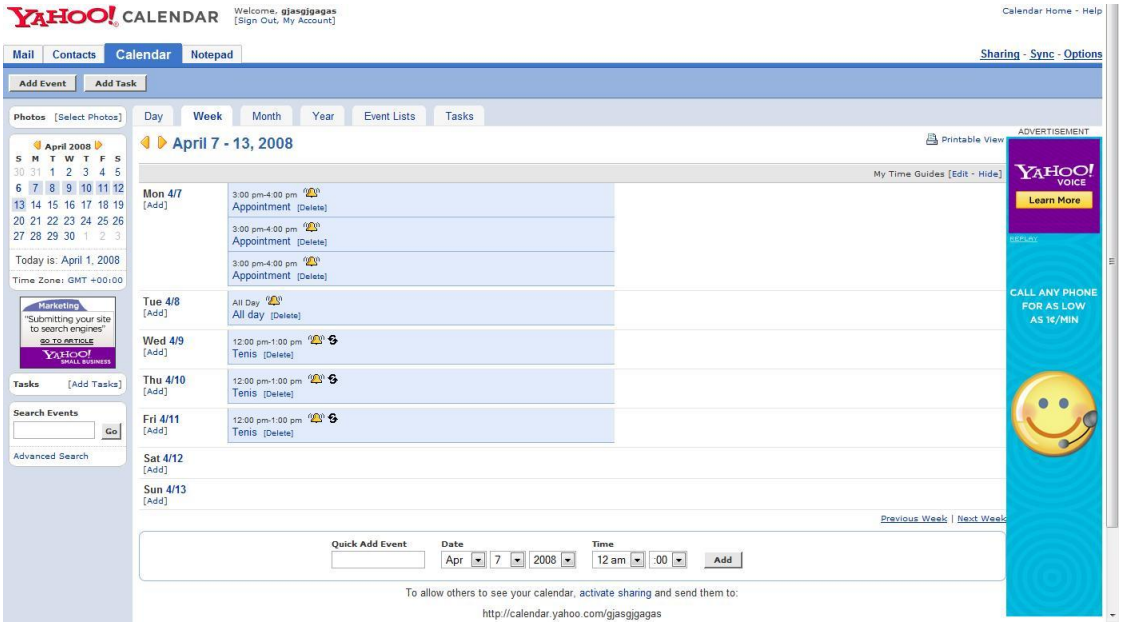
2.2.3 Yahoo Calendar

Kalendář na první pohled působí příjemným dojmem. Bohužel na pravé straně je umístěna reklama. Jak vypadá GUI Yahoo Calendar, je vidět na obrázku 5. Pod příjemným designem se skrývá stará aplikace, která působí velmi neobratně a ovlivňuje významně způsob ovládání. To se projevuje, jak při každé interakci s kalendářem, kdy je nutné znovu načtení celé stránky, tak i při ovládání kdy událost nelze jednoduše vybrat kliknutím na jí zabíranou plochu.

Rozvržení aplikace je obdobné jako u ostatních aplikací. Na levé straně malý kalendář a pravá strana je vyplněna velkým kalendářem. Kalendář nabízí denní, týdenní, měsíční, roční pohled a obdobu agendy. Na rozdíl od konkurence není týdenní pohled rozšířený denní pohled, ale podobá se spíše agendě. Aplikace nenabízí žádné možnosti DnD. V událostech lze vyhledávat.

Pro vytvoření nové události nelze jednoduše kliknout do kalendáře, ale musíte kliknout na čas či na odkaz *„add“* po levé straně anebo využit tlačítko *„Add Event“*. Události lze vytvářet s přesností na 15 minut. Na rozdíl od konkurence zde neurčujete počátek a konec události, ale počátek a dobu trvání v hodinách a minutách. Oproti ostatním má Yahoo Calendar samostatný formulář pro rychlé vytvoření události pod každým pohledem.

Na blížící se událost se lze nechat upozornit e-emailem, zprávou v rámci Yahoo Messengeru či si nechat poslat SMS na mobilní telefon. Kalendář si můžete nechat zasílat každý den na e-mail. Kalendář můžete opět sdílet s přáteli či ho publikovat na web. Vybraní uživatelé mohou kalendář editovat. Ke sdílenému kalendáři je snadný přístup na adrese http://calendar.yahoo.com/jmeno. Kalendář navíc podporuje import a export do formátu DBA nebo CSV.



Obrázek 5: Yahoo Calendar

2.3 Shrnutí rešerše

V podstatě všechny aplikace nabízejí obdobnou nabídku pohledů na kalendář a to prostřednictvím denního, týdenního a měsíčního pohledu. Téměř všechny aplikace se snaží podpořit intuitivní práci s událostmi pomocí DnD. Lze vytvářet jak normální, tak i opakující se události. Celodenní a vícedenní události bývají zpravidla zobrazeny nad aktuálním pohledem. Barevné odlišení události je zpravidla pevně určeno kalendářem, do něhož událost patří.

Před samotnou událostí je možné se upozornit alarmem či e-mailem. Aplikace umožňují kalendář buď zveřejnit, nebo ho sdílet mezi určitou skupinou uživatelů, avšak bez možnosti vzájemné editace. Události je možné obvykle importovat a exportovat ve formátu iCalendar.

Přehled nevýhod existujících aplikací:

a) Dostupnost kalendáře pouze na vlastním PC u desktopových kalendářů.
b) Platformní závislost některých desktopových kalendářů.
c) Technologická omezení u webových kalendářů neumožňující např. dokonalý DnD.
d) Nemožnost třídit události do skupin.
e) Nemožnost barevně odlišit události v pohledu jinak než novým kalendářem.
f) Nemožnost editace sdílených kalendářů.
g) Nepodpora kontaktů a poznámek u některých kalendářů.
h) Neumožňují zobrazit rozvrh z KOSu.

3 Popis problému a specifikace cílů

Tento projekt vznikl na základě požadavku výzkumné skupiny VLSI na Katedře počítačů FEL, ČVUT v Praze, za účelem koordinace schůzek členů této skupiny. Z tohoto důvod se nejdříve pokusím zanalyzovat aktuální situaci na Katedře počítačů a získat tak seznam relevantních požadavků. Následně s využitím těchto poznatků a informací získaných z rešerše stávajících produktů stanovým požadavky kladené na výslednou aplikaci.

3.1 Analýza aktuální situace na Katedře počítačů FEL, ČVUT v Praze

Jako na každé škole, tak i na ČVUT je život spjat se školním rozvrhem. Ten může být vnímán ze dvou pohledů. Z pohledu studenta se jedná o školní rozvrh určující jeho výuku. Naopak z pohledu učitelů určuje seznam jím vyučovaných předmětů a místa konání. Jak je zřejmé, rozvrh je důležitý pro obě skupiny. Na ČVUT se o správu rozvrhů stará KOS. Je tedy velmi vhodné pokusit se zpřístupnit rozvrh z KOSu do kalendáře a umožnit, tak plné zapojení školního rozvrhu do tvorby časových plánů.

Vzhledem k požadavku na zprostředkování rozvrhu z KOSu, tu vzniká i bezpečnostní požadavek. Rozvrh obsahuje citlivá data, jako je uživatelské jméno nebo časový harmonogram s místem výskytu jednotlivých osob. Z toho důvodu je třeba zajistit, aby veškerá komunikace mezi distribuovanými částmi aplikace byla zabezpečená.

Také na Katedře počítačů existuje řada výzkumných skupin. Členové těchto skupin mají ať již pravidelné, tak i nahodilá setkání. Je zřejmé, že zkoordinovat volný čas všech členů je velmi obtížná činnost. Zároveň je třeba, aby se všichni členové dozvěděli zejména o nepravidelném setkání. Bylo by tedy jistě vhodné, aby aplikace umožňovala vytvoření a správu sdílených kalendářů v rámci jednotlivých skupin s možností editace těchto sdílených kalendářů jednotlivými členy a napomohla tak koordinaci času jednotlivých členů.

Většina pedagogů vypisuje konzultační hodiny. Je zde proto nasnadě možnost vytváření tzv. veřejných skupin, do kterých by se mohli studenti sami přihlašovat a získat, tak informace o aktuálních konzultacích jednotlivých pedagogů.

Zejména na Katedře počítačů je možné narazit na velkou řadu operačních systémů, jak v řadách učitelů či studentů. Namátkou jmenujme Windows XP a Vista, různé distribuce Linuxu či Solaris. Je patrné, že je nutné pro implementaci zvolit platformě nezávislý jazyk, který umožní běh na všech těchto platformách.

3.2 Požadavky na elektronické plánovací kalendáře a stanovení cílů

Aplikace by měla sloužit k tvorbě a údržbě časových plánů, kontaktů a poznámek. Zároveň by měla poskytnout snadný a rychlý způsob získání informací např. s pomocí RSS či vyhledávání na internetu.

Jedním z hlavních požadavků je intuitivní a snadná tvorba časových plánů. Z tohoto důvodů je vyžadována podpora DnD při tvorbě kalendáře. Standardně je nabízen denní, týdenní (pracovní týden a celý týden) a měsíční pohled. Aplikace by měla sloužit jak začátečníkům, tak i pokročilým uživatelům. Vhodná by tedy byla podpora jak klasického vytvoření události se všemi dostupnými volbami, tak i tzv. *„Quick add“* pro rychlé vytvoření události s minimem úsilí. Události bude možné barevně odlišit dle kategorií.

Kalendář by měl umožňovat sdílení rozvrhu v rámci uživatelsky definovaných skupin. Členové těchto skupin by mohli následně takto sdílený kalendář editovat s ohledem na práva jednotlivých členů. Aplikace by měla umožňovat tvorbu veřejných skupin, do kterých se můžou uživatelé přidávat sami, a privátních skupin, které jsou plně pod kontrolou administrátora. Součástí programu by měl být importní a exportní modul, který bude umožňovat import a export událostí do formátu iCalendar (RFC 2445) a kontaktů do formátu vCard (RFC 2426). V rámci kalendáře bude možné pracovat se školním rozvrhem z KOSu.

Veškerá komunikace mezi distribuovanými částmi aplikace bude šifrovaná. Výsledná aplikace bude platformě nezávislá. Architektura aplikace by měla být otevřená, aby bylo umožněno další rozšiřování aplikace v budoucnu. Vrstevnatá architektura ve spojení s návrhovými GoF vzory by měla zajistit zaměnitelnost jednotlivých vrstev za nové. Architektura by měla umožnit dostupnost kalendáře kdykoliv a kdekoliv a zároveň by měla podpořit možnost tvorby nových klientských částí (např. klient pro mobilní telefony). Aplikace by měla být vystavěna jako klasická třívrstvá aplikace – klient, server a databáze.

Správce kontaktů bude podporovat víceúrovňové zobrazení kontaktů s rozdílným množstvím poskytovaných informací. Poznámky bude možné vytvářet snadným způsobem a bude umožněna jejich interaktivní editace.

4 Analýza a návrh řešení

V této kapitole se budu zabývat zejména analýzou a návrhem řešení. Nejprve se pokusím zanalyzovat danou problémovou doménu. Poté se budu zabývat uživatelskými právy, což je v úzkém spojení s následující kapitolou zabývající se návrhem sdílení kalendářů mezi uživateli. Poté se zabývám návrhem datového úložiště. Dále zde bude probráno téma logické architektury a rozdělení systému na komponenty. Následně je ukázána komunikace v typickém případu užití a návrh uživatelského rozhraní. Na závěr pak rozeberu problematiku výběru vhodné implementační platformy.

4.1 Doménová analýza a doménový model

V této kapitole se pokusím popsat problémovou doménu s využitím poznatků z kapitoly *Popis stávajících řešení* a z analýzy v kapitole *Popis problému a specifikace cílů*. Doménový model můžete vidět na obrázku 6. Centrem veškerého dění je uživatel, který je v rámci systému jednoznačně identifikován svým uživatelským jménem. Každý uživatel si může definovat nastavení, která jsou platná pouze pro něho samotného. V rámci systémů existují tzv. skupiny. Uživatelé se mohou sdružovat do těchto skupin. Uživatel může být členem v libovolném počtu skupin, ale nejméně v jedné. Naopak skupina nemusí mít žádné členy.

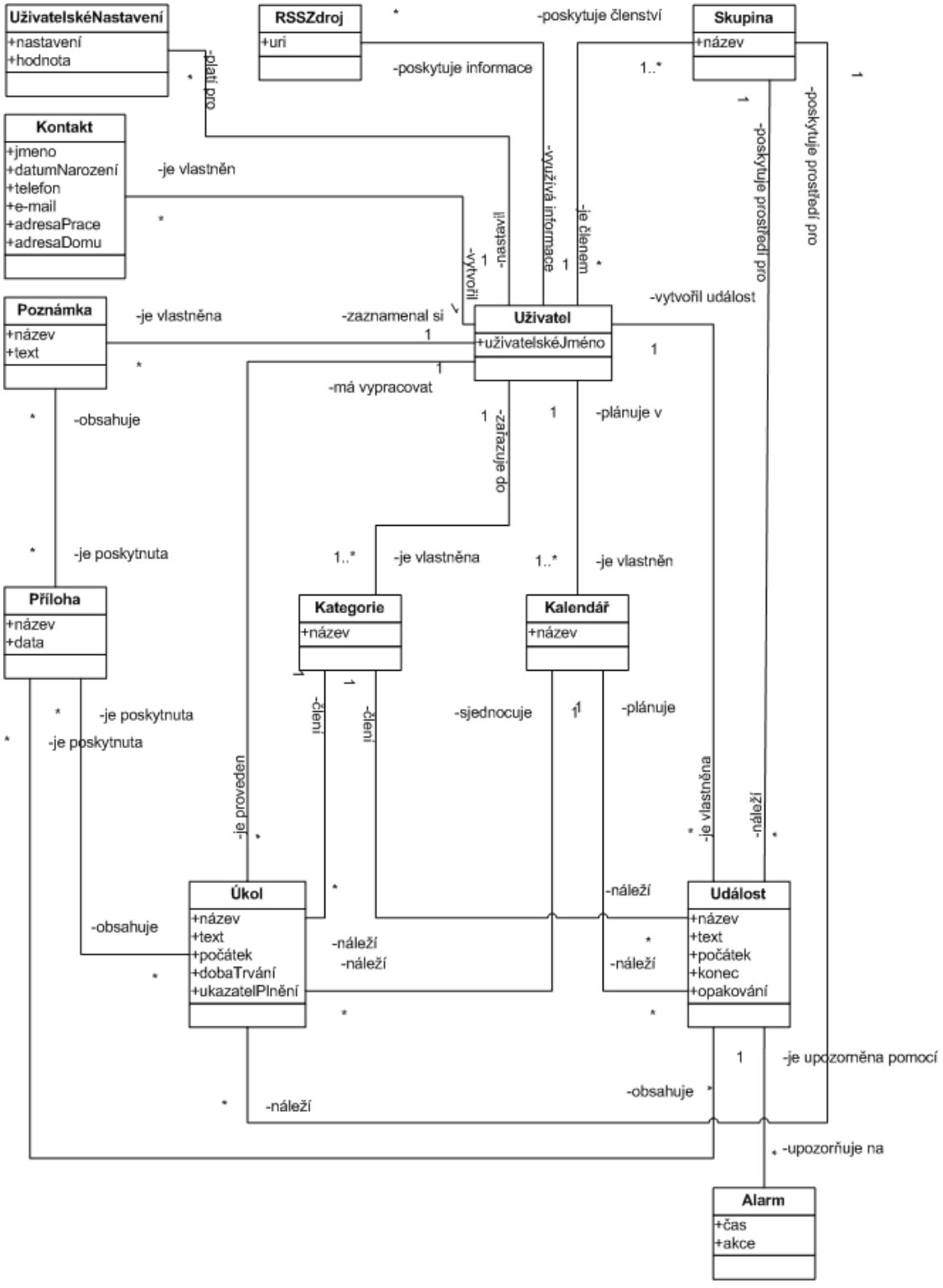
Každý uživatel si vytváří vlastní kalendáře. Tyto kalendáře poskytují dvě základní funkcionality. Z pohledu uživatele slouží k logickému rozdělení jednotlivých časových plánů, např. jeden kalendář slouží k tvorbě časových plánů v rámci zaměstnání a druhý ve volném čase. Naopak události a úkoly jsou plánovány v rámci kalendáře. Každý uživatel musí mít alespoň jeden kalendář.

Kategorie slouží k logickému třídění jednotlivých událostí a úkolů. Příkladem budiž narozeniny či pracovní schůzka. Uživatel si může definovat vlastní kategorie. Pro události a úkoly je možné definovat vlastní kategorie. Každý uživatel musí mít alespoň jednu kategorii pro úkoly a jednu pro události.

Událost slouží k zaznamenání časového úseků s přesností na minuty. Příkladem události může být např. pracovní schůzka nebo nákup. Každá událost má název, počátek a konec. K události lze přiřadit i delší popisek, který vysvětluje obsah události. Událost může být jednorázová nebo se může opakovat. Událost je plánována v rámci jednoho kalendáře a náleží právě jednomu uživateli. Každá událost je zatříděna právě do jedné kategorie a náleží vždy pouze do jedné skupiny. Uživatel se může nechat na nadcházející událost upozornit alarmem. Jednotlivé alarmy jsou svázány s konkrétními událostmi. Na každou událost se lze nechat upozornit více alarmy s rozličnou akcí.

Úkol slouží zaznamenání nějaké činnosti, která má být provedena. Lze ho definovat s přesností na dny. U úkolu lze specifikovat název, vysvětlující popisek a počátek. Každý úkol má předpokládanou dobu trvání, v rámci které lze sledovat postupné plnění úkolu. Jednotlivé úkoly patří vždy konkrétnímu uživateli. Každý úkol je jednoznačně zatříděn do některé kategorie a náleží právě do jedné skupiny. Úkoly jsou logicky seskupeny v rámci kalendáře.

Uživatelé si mohou vytvářet poznámky. Poznámka má název a text. K poznámce lze připojit libovolný počet příloh. Také událost nebo úkol smí mít přílohu. Přílohy existují v systému pouze jednou a mohou být sdíleny mezi více poznámkami, událostmi či úkoly.



Obrázek 6: Doménový model

Každý uživatel si může vytvářet kontakty na další osoby. Kontakt slouží k zaznamenání informací o lidech jako je jméno, telefon, e-mail či poštovní adresa. Kontakt patří pouze uživateli, který ho vytvořil.

RSS zdroj slouží k zaznamenání RSS kanálu, ze kterého lze získat informace o aktuálním dění. Každý uživatel si udržuje seznam vlastních RSS zdrojů. Jednotlivé zdroje patří vždy danému uživateli.

4.2 Sdílení kalendářů mezi uživateli

V kapitole *Popis problému a specifikace cílů* byly zmíněny hlavní důvody pro požadavek sdílení kalendářů mezi uživateli. Uživatelé se často seskupují do skupin se společným zájmem. Je tedy zcela logické, aby mohli uživatelé sdílet časové plány v rámci skupiny. Jelikož nelze předem určit seznam skupin, musí se jednat o uživatelsky definované skupiny. Jednotliví uživatelé se mohou stát členy konkrétních skupin. Každý člen má přiřazena vlastní práva v rámci každé skupiny, více o právech v kapitole *Uživatelská práva*. Skupina nemusí mít žádné členy nebo jich může mít neomezeně.

Každá událost nebo úkol je jednoznačně přiřazena právě do jedné skupiny, přičemž skupina může poskytnout prostředí pro neomezené množství úkolů či událostí. Každá událost nebo úkol patří právě jednomu uživateli. Událost nebo úkol nemůžou náležet do skupiny, v které není jejich vlastník členem. Na základě těchto informací lze jednoznačně určit, které události patří do dané skupiny a kdo je jejich vlastník v rámci skupiny. Uživatel si tedy může nechat zobrazit všechny události, které jsou v dané skupině a jsou jeho.

Skupiny lze vytvářet jako veřejné nebo jako privátní. Pokud je skupina označena jako privátní, je plně pod kontrolou administrátora. Pouze administrátor rozhoduje o tom, kdo bude členem dané privátní skupiny a jaká bude mít práva.

Veřejná skupina je specifická tím, že je pro všechny uživatele volně dostupná. Každý uživatel má právo si stáhnout kompletní seznam všech veřejných skupin. Následně se může sám stát členem libovolné veřejné skupiny.

Uživatelé se mohou svobodně odhlašovat z členství ve skupinách, jak z veřejných tak i z privátních. Více o právech uživatelů v rámci skupin v následující kapitole.

4.3 Návrh datového úložiště aplikace

Návrh datového úložiště samotné aplikace vychází z poznatků v kapitole *Doménová analýza a doménový model*. Jako reprezentaci jsem zvolil UML diagram tříd, který můžete vidět na obrázku 7. V jednotlivých třídách jsou zachyceny pouze nejpodstatnější atributy, jinak třídy obvykle obsahují ještě celou řadu méně podstatných atributů. Každá třída obsahuje atribut *id*, který slouží k jednoznačné identifikaci objektů a provázání mezi objekty jak v aplikaci, tak i v databázi.

Uživatel je reprezentován třídou *VUser*. Každý uživatel je jednoznačně identifikovaný kromě *user_id* také *username* (uživatelské jméno). K identifikaci uživatele při přihlášení slouží uživatelské jméno a heslo. Pro možnost komunikace v případě nějakého problému jsou

o uživateli uchovány kontaktní informace. Každý uživatel má přidělena určitá bezpečnostní práva, která jsou uložena v atributu *policy*, více v kapitole *Uživatelská práva*.

Každý uživatel si může definovat vlastní nastavení v třídě *VProperty*. Ta je tvořena dvojicí nastavení a její hodnotou. Uživatel může mít definovaných 0 až *n* nastavení.

V systému mohou vznikat skupiny, které jsou reprezentovány třídou *VGroup*. U každé třídy známe její jméno, a zda je veřejná či nikoliv. Uživatelé se mohou stát členy jednotlivých skupin. Tento vztah je zachycen třídou *VMembership*, v rámci kterého má uživatel vždy přidělena určitá práva *policy*. Skupiny mohou existovat, aniž by měli nějakého člena. Každý uživatel musí být členem alespoň v jedné skupině.

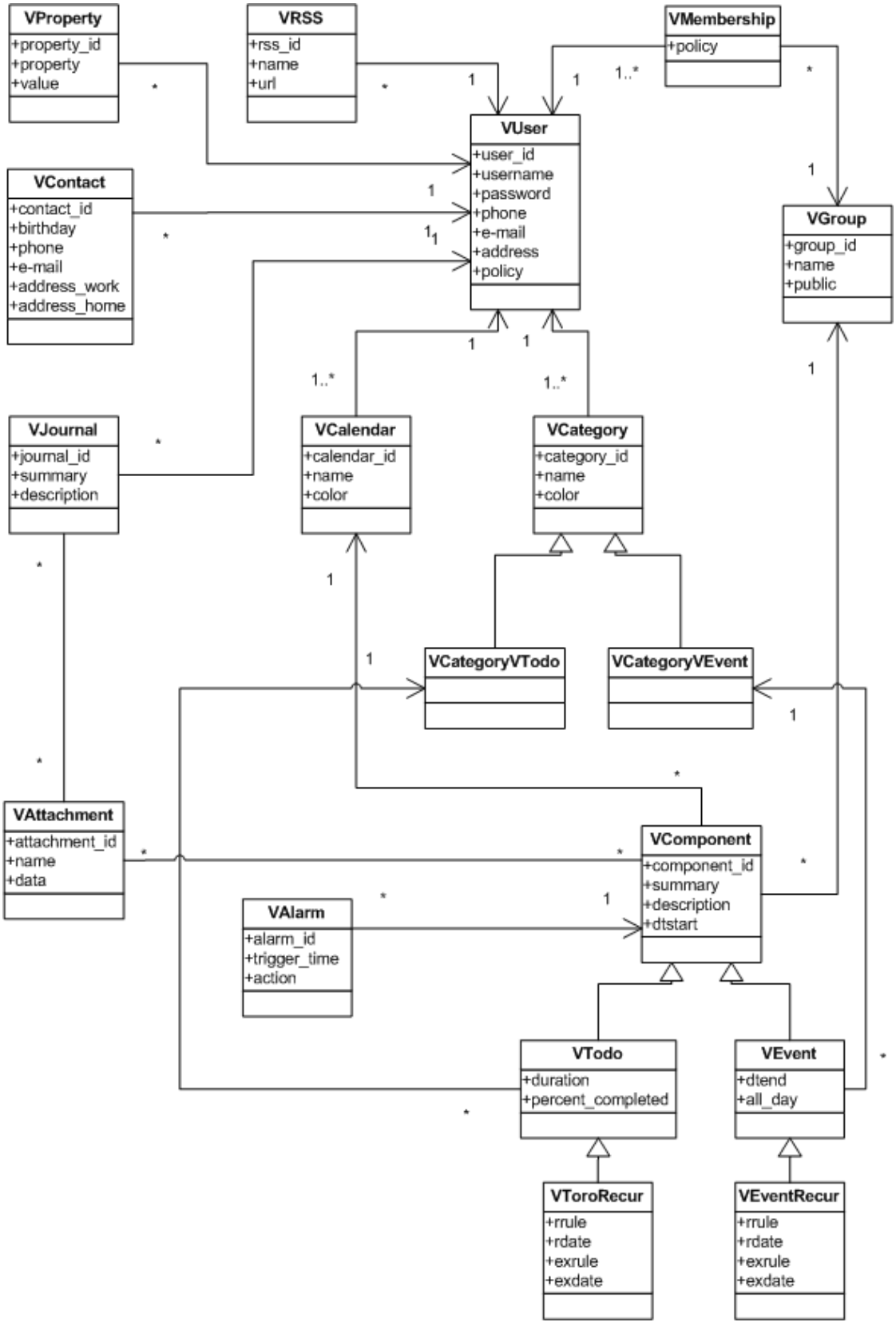
Jelikož úkol a událost jsou si velmi podobní a mají celou řadu atributů shodnou, mají společného předka *VComponent*. Každá komponenta má název, popisek a čas začátku. Třída *VComponent* má dva potomky – *VEvent* (reprezentuje událost) a *VTodo* (reprezentuje úkol). U úkolu máme stanovenou dobu trvání *duration* a sledujeme postupné plnění úkolu *percent_completed*. U události nás naopak zajímá její konec *dtend*, a zda se jedná o celodenní událost *all_day*. Celodenní událost trvá celý den od 0:00 do 23:59. Každá komponenta náleží právě do jedné skupiny, naopak do jedné skupiny může náležet 0 až *n* komponent. Takto definovaný úkol nebo událost vyjadřují jednorázový děj. Často však potřebujeme vyjádřit opakující se událost. Už z jednoduché podstaty, že lze také definovat neomezené opakování, je zcela nemožné vyjádřit opakování rozmnožením dané události či úkolu. Z tohoto důvodu jsem zvolil způsob definovaný standardem iCalendar (RFC 2445), kde daný úkol či událost je zaznamenána pouze jednou a jsou definována pravidla pro opakování. Opakování lze definovat pomocí atributu *RRule* nebo *RDate*. *RRule* slouží k definici opakování dle určitého pravidla, viz RFC 2445. *RDate* umožňuje definování opakování skrze výčet konkrétních dat výskytu. Výjimky z těchto pravidel lze definovat pomocí *ExRule* a *ExDate*. *ExRule* slouží k definování pravidla, kdy se událost či úkol nevyskytuje. *ExDate* umožňuje definování výjimky prostřednictvím konkrétních dat. Třída *VTodo* má pro definování pravidel opakování potomka *VTodoRecur*. Třída *VEvent* má pak definovaná pravidla opakování v potomkovi *VEventRecur*.

Uživatel si třídí úkoly a události do kategorií. Kategorie je reprezentovaná třídou *VCategory*, u které nás zajímá název a barva pro grafické rozlišení. Kategorie lze definovat zvlášť pro události *VCategoryVEvent* a zvlášť pro úkoly *VCategoryVTodo*. Každý uživatel má definovanou alespoň jednu kategorii. Událost nebo úkol náleží právě do jedné kategorie.

Události nebo úkoly lze plánovat v rámci kalendářů. Kalendář *VCalendar* má název a přiřazenou barvu. Uživatel má alespoň jeden kalendář. Každá událost nebo úkol náleží právě do jednoho kalendáře.

Na událost nebo úkol se může uživatel nechat upozornit. Upozornění je reprezentováno instancí třídy *VAlarm*. U alarmu nás zajímá s jakou je spojen událostí, kdy nás má upozornit *trigger_time* a jaká akce se má provést. Každý alarm náleží právě jedné události či úkolu. Naopak na událost či úkol se lze nechat upozornit více alarmy, např. s rozdílnou akcí.

Uživatel si může vytvářet poznámky *VJournal*. Každá poznámka má název a vysvětlující popisek. Každá poznámka patří právě jednomu uživateli, ale uživatel si smí vytvořit 0 až *n* poznámek.



Obrázek 7: Diagram tříd datového úložiště aplikace

K poznámce lze přiložit přílohu. U poznámky může být 0 až *n* příloh. Příloha je reprezentována třídou *VAttachment*. U přílohy si pamatujeme jméno a data, která ji tvoří. Příloha existuje v systému pouze jednou, proto danou přílohu může vlastnit více poznámek. Ke komponentě je možné také přiřadit poznámku. Komponenta může mít přiřazeno 0 až *n* příloh.

Kontakty jsou reprezentovány třídou *VContact*. U kontaktu sledujeme jméno, telefon, e-mail, adresu domů či do práce atd. Každý kontakt byl vytvořen právě jedním uživatelem, naopak uživatel smí vytvořit a následně vlastnit 0 až *n* kontaktů.

RSS zdroj je reprezentován třídou *VRSS*, u které nás zajímá název zdroje a URL, ze které lze data získat. Každý uživatel smí vlastnit 0 až *n* RSS zdrojů, naopak RSS zdroj vždy patří právě jednomu uživateli.

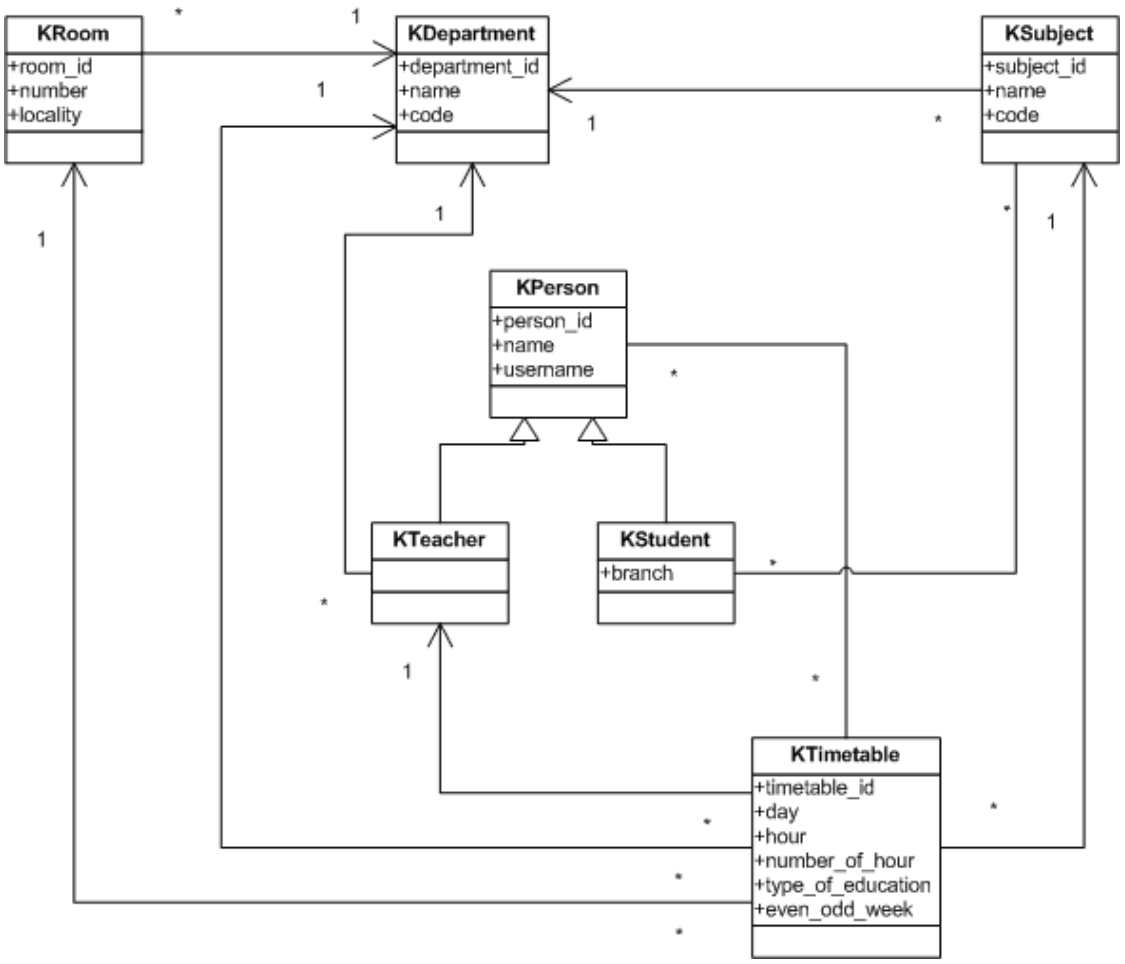
4.4 Návrh datového úložiště rozvrhu z KOSu

Návrh datového úložiště rozvrhu z KOSu vychází především z datového formátu, ve kterém jsou data z KOSu exportována. Jako reprezentaci jsem zvolil UML diagram tříd, který je vidět na obrázku 8. V jednotlivých třídách jsou zachyceny pouze nejpodstatnější atributy, jinak třídy obvykle obsahují ještě celou řadu méně podstatných atributů. Každá třída obsahuje atribut *id*, který slouží k jednoznačné identifikaci.

Katedra je reprezentována třídou *KDepartment*, u které si pamatujeme název a kód. Místnosti jsou reprezentovány třídou *KRoom*. U místnosti nás zajímá především její číslo a lokalita. Každá místnost patří právě jedné katedře a katedra může vlastnit 0 až *n* místností.

Každá katedra vyučuje 0 až *n* předmětů. Předmět *KSubject* má název a kód. Předmět je vyučován právě jednou katedrou. Ve škole se nachází řada osob *KPerson*. U osob sledujeme jméno a uživatelské jméno. Osoby jsou dvojího druhu – učitelé *KTeacher* a studenti *KStudent*. U studentů sledujeme obor studia *branch*. Studenti se zapisují na jednotlivé předměty. Studen může mít zapsáno 0 až předmětů a předmět si mohlo zapsat 0 až *n* studentů. Učitelé jsou zaměstnanci katedry. Každý učitel je zaměstnán právě u jedné katedry, ale katedra může zaměstnat 0 až *n* učitelů.

Každá osoba má svůj rozvrh pro jednotlivé hodiny *KTimetable*. Osoba může mít 0 až *n* vyučovacích hodin a na danou hodinu může být zapsáno 0 až *n* osob. U vyučovací hodiny sledujeme den, ve kterém se koná, čas kdy začíná, jak dlouho trvá, typ výuky (přednáška, cvičení, laboratoř či ostatní) a týden výuky (lichý, sudý nebo oba). Hodina je vyučována právě jedním učitelem a jeden učitel může vyučovat více hodin. V rámci každé hodiny se vyučuje právě jeden předmět, ale předmět může být vyučován ve více hodinách. Vyučování probíhá v rámci předmětu na dané katedře. Na katedře se může vyučovat 0 až *n* hodin. Výuka se odehrává právě v jedné místnosti a v místnosti se může vyučovat 0 až *n* hodin (ne však současně).



Obrázek 8: Diagram tříd datového úložiště rozvrhu z KOSu

4.5 Logická architektura

Z pohledu fyzické architektury se jedná o klasickou třívrstvou aplikaci – klient, server a databáze. Naopak z pohledu logické architektury je aplikace rozdělena do pěti vrstev – *UI* (prezentační), *Application* (aplikační), *Domain* (doménová či aplikační logika), *Technical Services* (technické služby) a *Foundation* (nízko úrovňové technické služby). UML *package* diagram zobrazující logickou architektur je na obrázku 9.

Prezentační vrstva se stará o zobrazení dat z doménové vrstvy poskytovaných prostřednictvím aplikační vrstvy. V této vrstvě lze nalézt balíček *View* a *UI Utilities*. Balíček *View* obsahuje dva typy komponent. Prvním typem jsou komponenty, které se vkládají do jiných komponent např. *VEventCreateEditPanel*. Druhým typem jsou komponenty, do kterých jsou naopak komponenty vkládány, např. *WizardDialog*. Každá komponenta, do které lze vkládat komponenty musí implementovat interface *ScreenInterface* z aplikační vrstvy. Balíček *UI Utilities* obsahuje pomocné třídy např. na změnu formy zobrazovaných dat atd.

Aplikační vrstva zpracovává požadavky prezentační vrstvy, stará se o přechod oken a konsolidaci dat pro prezentaci. V této vrstvě lze nalézt tři balíčky – *Data Manager*, *User Manager* a *Screen*. Balíček *Screen* obsahuje interface *ScreenInterface*, který definuje metody,

které musí implementovat všechny komponenty, do kterých lze vkládat další komponenty. Další třídou je pak *ScreenManager*. Tato třída slouží k tvorbě a sdílení komponent. Na základě dodaného typu *ScreenManager* dodá požadovanou komponentu. *ScreenManager* umožňuje dodat pokaždé novou komponentu nebo sdílenou komponentu. Sdílená komponenta je vytvořena během běhu programu pouze jednou a při každém požadavku na ni je vrácena stejná instance této komponenty. Sdílené komponenty jsou v *ScreenManageru* udržovány v bufferu typu *mapa*. V kombinaci se *ScreenInterfacem* nám tento návrh umožňuje vytvoření pouze jednoho dialogu a jeho sdílení, tak že se bude během běhu aplikace měnit pouze jeho obsah. Tento sdílený dialog se pak vždy přizpůsobí vkládané komponentě.

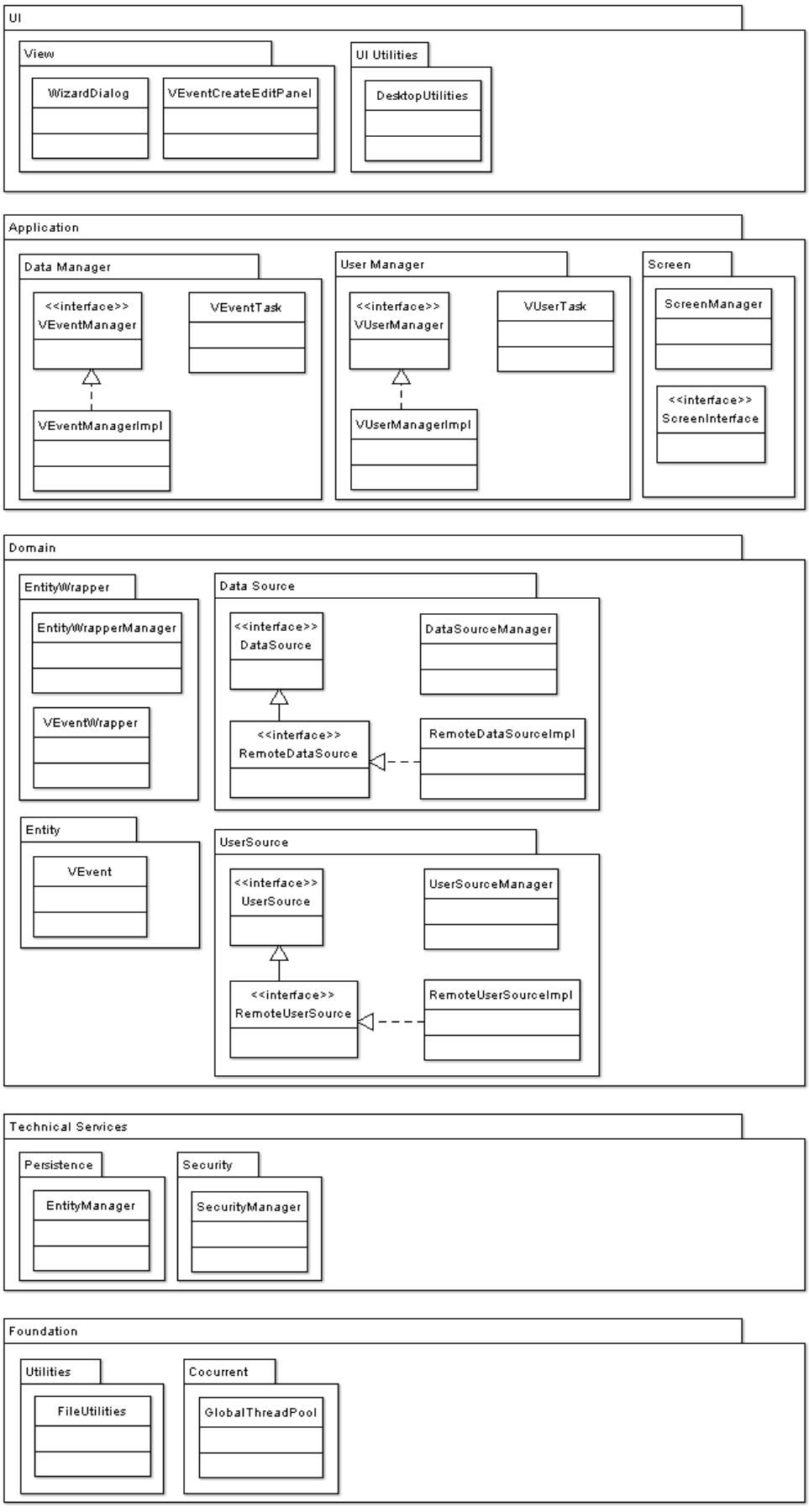
Druhým balíčkem je pak *UserManager*. Zde se nachází interface *UserManager*, který definuje sadu funkcí, které slouží pro správu uživatelů a jsou poskytovány prezentační vrstvě. *UserManagerImpl* je pak konkrétní implementace tohoto interfacu. Tato třída následně deleguje činnost dále na datové zdroje. Další třídou je *VUserTask*, která slouží k asynchronnímu zpracování požadavků na *UserManagera* z prezentační vrstvy.

Třetím balíčkem je pak *Data Manager*, který se stará o zprostředkování dat pro prezentační vrstvu. Zde lze nalézt např. *VEventManager*, který definuje služby pro práci s *VEvent* daty poskytované prezentační vrstvě. *VEventManagerImpl* je pak implementací daného interfacu a deleguje zpracování požadavků na datové zdroje. *VEventTask* pak slouží k asynchronnímu zpracování požadavků na *VEventManagera* od prezentační vrstvy.

Další vrstvou je pak doménová vrstva, která obsahuje balíčky – *Entity*, *EntityWrapper*, *UserSourse* a *DataSource*. Balíček *Entity* definuje doménové objekty, jako je např. *VEvent, VContact* atd. *EntityWrapper* obsahuje obalující třídy pro některé třídy z balíčku *Entity*. Tyto obalující třídy slouží k poskytnutí další funkcionality. Například u *VCalendar* je definována barva po RGB+A složkách a obalující třída poskytne přímo objekt typu *Color* s danými RGB+A složkami.

Balíček *User Source* obsahuje interface *UserSource*, který definuje služby poskytované pro správu uživatelů. Od těchto interfaců existují potomci např. *RemoteUserSource* nebo *DatabaseUserSource*, které charakterizují typ zdroje. Např. *RemoteUserSource* říká, že se jedná o vzdálený zdroj, který bude pravděpodobně vytvářet další fyzickou vrstvu, naopak *DatabaseUserSource* říká, že zdrojem je přímo databáze. *RemoteUserSourceImpl* je pak konkrétní implementace spjatá se specifickou technologií. Pro správu zdrojů pro správu uživatelů je zde třída *UserSourceManager*. Tato třída umožňuje spravovat více zdrojů současně a podle potřeby je měnit. Obdobný mechanismus existuje i pro správu dat a je definován v balíčku *Data Source*.

Předposlední vrstvou jsou pak technické služby. V této vrstvě se nachází balíček *Persistence* a *Security*. Balíček *Persistence* se stará o správu persistentního úložiště a persistenci objektů. Balíček *Security* se stará o zajištění bezpečnosti. V tomto balíčku lze nalézt třídy pro správu a inicializaci šifer.



Obrázek 9: Logická architektura

Poslední vrstvou jsou tzv. nízko úrovňové technické služby. Zde nalézt balíčky pro práci se soubory a s vlákny. Balíček *Utilities* obsahuje podpůrné třídy např. *FileUtilities* pro práci se soubory. Druhý balíček *Concurrent* se stará o práci s vlákny. Obsahuje třídu *GlobalThreadPool*, která poskytuje pool pro znovu využití vláken. Dále tato třída poskytuje možnost asynchronního spouštění metod v aplikaci.

4.6 Diagram komponent

Komponenty jsou rozděleny do tří vrstev – klient, server a databáze. Diagram komponent je vidět na obrázku 10. V tomto návrhu máme v databázové vrstvě pouze jedinou komponentu, která je společným úložištěm jak informací o uživatelích tak i uživatelských dat. Toto společné úložiště by mohlo být v budoucnu nahrazeno samostatných správcem uživatelů a samostatným úložištěm pro uživatelská data. Komponenta *Database* poskytuje SQL rozhraní.

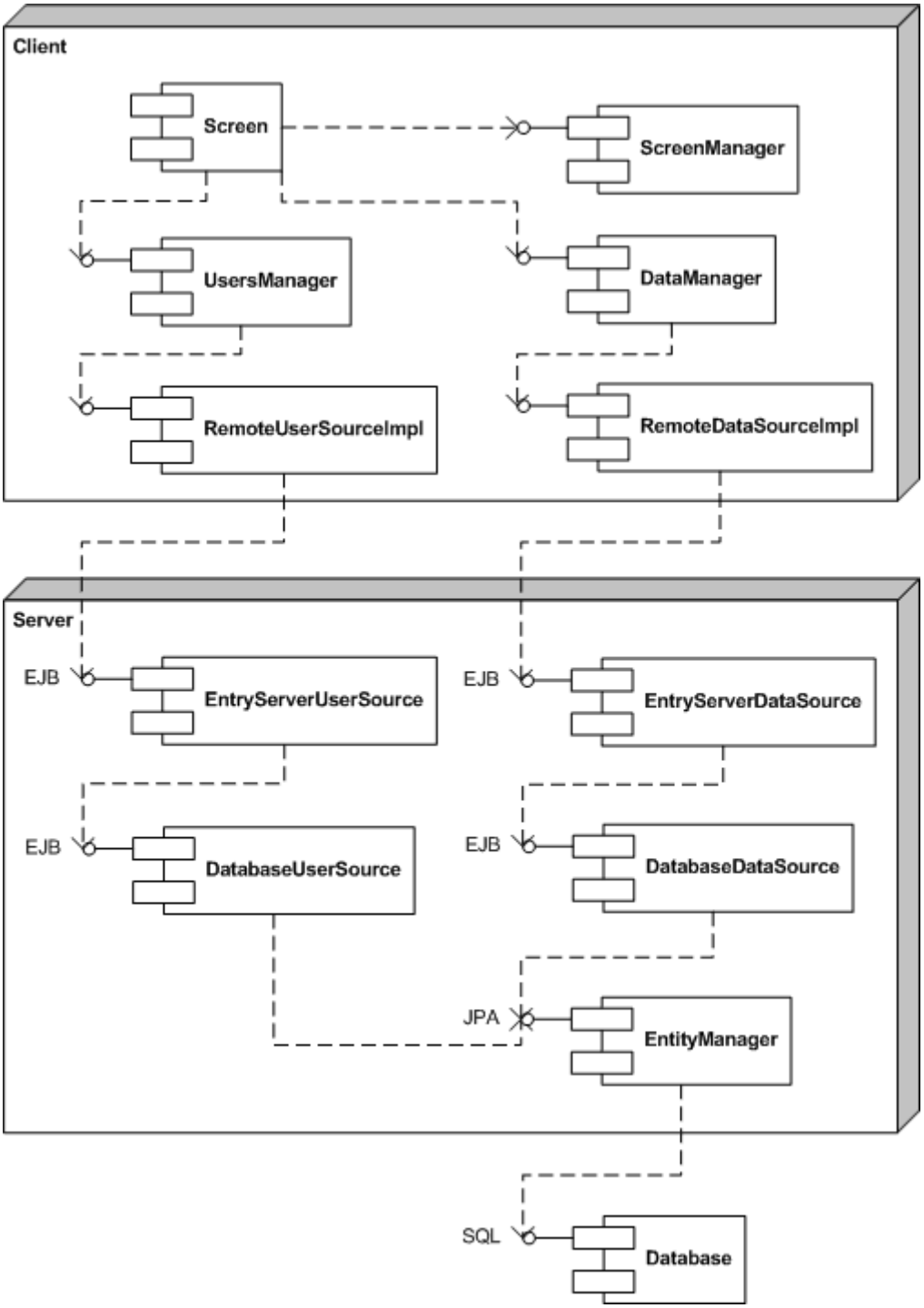
V serverové vrstvě se nachází komponenta *EntityManager*. Tato komponenta je komunikačně závislá na komponentě *Database*. *EntityManager* slouží k zajištění perzistence objektů a poskytuje ORM (Object-Relational mapping) mapování. Je zde poskytnuto rozhraní směrem k vyšším vrstvám prostřednictvím JPA (Java Persistence API). Na této komponentě jsou pak závislé komponenty *DatabaseUserSource* a *DatabaseDataSource*, které jsou realizované jako bezstavové EJB (Enterprise Java Beans).

DatabaseUserSource slouží k zpracování uživatelských požadavků na správu uživatelů a delegování požadavků na perzistenci objektů. Třída *DatabaseDataSource* zpracovává uživatelské požadavky na práci s daty a deleguje požadavky na perzistenci dat. Obě komponenty poskytují rozhraní prostřednictvím EJB. Zároveň *DatabaseUserSource* poskytuje *UserSource* rozhranní a *DatabaseDataSource* poskytuje *DataSource* rozhranní.

Na komponentě *DatabaseUserSource* je pak závislá komponenta *EntryServerUserSource*. Tato komponenta slouží jako vstupní brána na serverové straně aplikace pro správu uživatelů. Zde je možné definovat *UserSource* pro správu uživatelů, v našem případě je jím již zmíněný *DatabaseUserSource*, který však může být vyměněn za jiný, který také poskytuje rozhraní definované *UserSource*. U *EntryServerDataSource* je situace obdobná, zde se však jedná o vstupní bránu pro správu uživatelských dat. *EntryServerDataSource* je pak závislá na *DatabaseDataSource*. Obě komponenty poskytují EJB rozhranní.

Na straně klienta slouží pro transformaci požadavků na služby pro správu uživatelů na síťová volání komponenta *RemoteUserSourImpl*. Tato komponenta je závislá na *EntryServerUserSource* a poskytuje transformaci požadavků na síťová volání. *RemoteUserSourceImpl* poskytuje rozhraní definované *UserSource*. Naopak pro transformaci požadavků na uživatelská data na síťová volání slouží *RemoteDataSource*. Tato komponenta je závislá na *EntryServerDataSource* a poskytuje rozhranní *DataSource*.

UsersManager je komponenta, která slouží k zpracování požadavků pro správu uživatelů z komponenty *Screen* a delegování požadavků na *UserSource*. Tato komponenta je závislá na komponentě s rozhraním *UserSource*, v našem případě *RemoteUserSource*. *UsersManager* poskytuje *UserManager* rozhraní.



Obrázek 10: Diagram komponent

DataManager zpracovává požadavky na správu uživatelských dat z komponenty *Screen* a delegování požadavků na komponentu s rozhraním *DataSource*. Tato komponenta je závislá na komponentě s rozhraním *DataSource*, např. *RemoteDataSource*. *DataManager* poskytuje rozhraní *DataManager*.

Komponenta *ScreenManager* slouží k tvorbě a správě prvků uživatelské rozhraní. Více v kapitole *Logická architektura*. *Screen* je komponenta, která se stará o zobrazení uživatelských dat a delegaci uživatelských požadavků. Tato komponenta je závislá

na komponentě *ScreenManager*, na komponentě s rozhraním *UserManager*, zde *UsersManager*, a na komponentě s rozhraním *DataManager*, zde *DataManager*.

4.7 Sekvenční diagram

V celé aplikaci existuje nepřeberné množství případů užití. Pro každý tento případ užití pak lze vytvořit sekvenční diagram. Z tohoto důvodu jsem se rozhodl vytvořit sekvenční diagram pro typický případ užití, který v sobě zahrnuje všechny významné součásti systému. Takovýmto typickým případem užití je vytvoření nové události v kalendáři. Tento scénář je vidět na obrázku 11.

Na počátku jsme detekovali požadavek na vytvoření nové události. Nejprve požádáme *ScreenManager* o grafickou komponentu (dialogové okno), do které vložíme grafickou komponentu pro vytvoření nové události. Poté požádáme *ScreenManager* o grafickou komponentu pro vytvoření nové události. Následně požádáme *ScreenManager* o vložení komponenty pro vytváření nové události do dialogu. *ScreenManager* deleguje požadavek na vložení na samotný dialog, který implementuje rozhraní *ScreenInterface*. Dialog vloží komponentu jako svůj obsah. Nakonec zavoláme nad dialogem metodu *setVisible(true)*, abychom zobrazili dialog. Dialog zavolá také *setVisible(true)* nad jeho obsahem, aby i on byl viditelný.

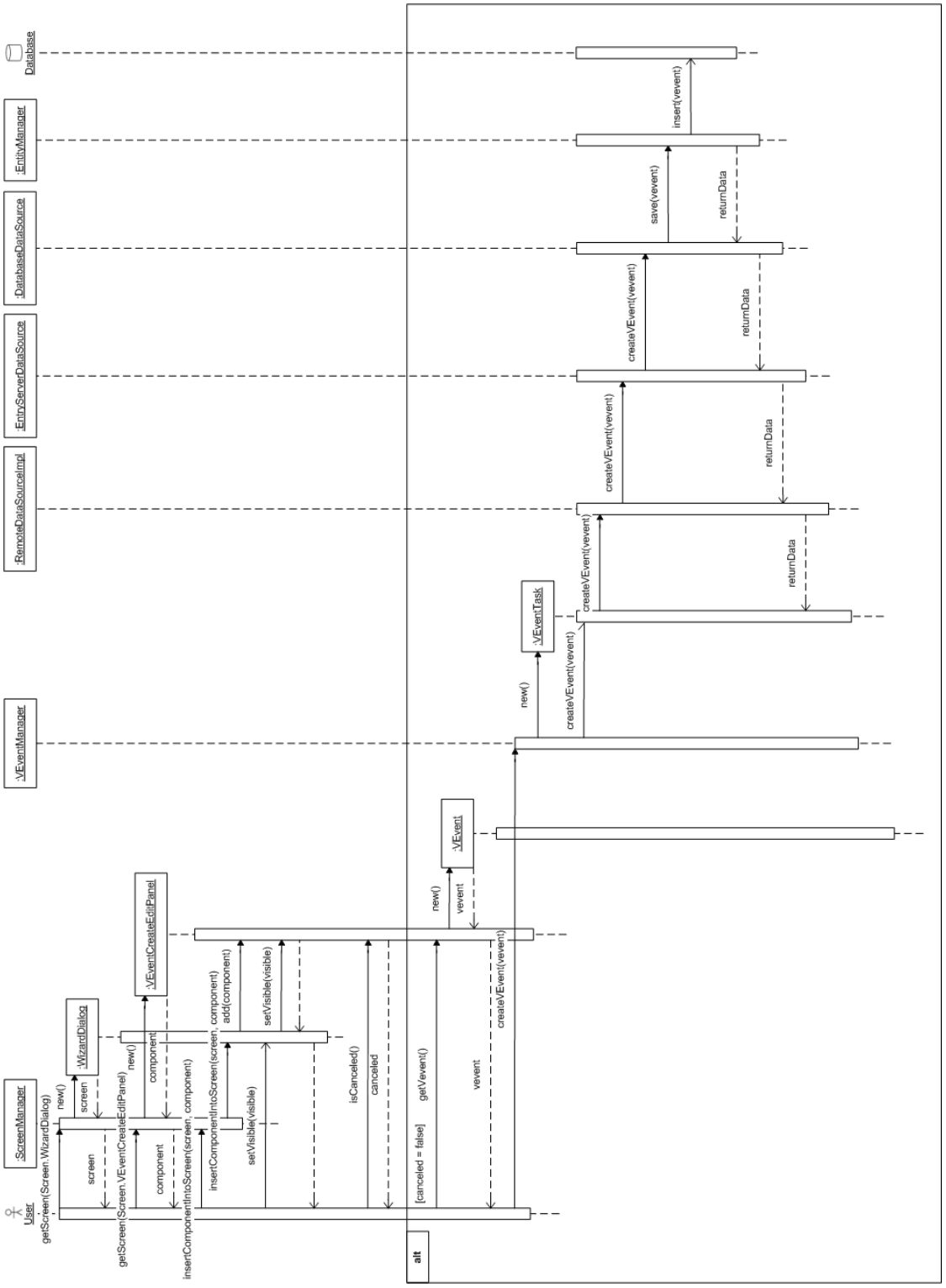
Vyčkáme, dokud uživatel dialog neuzavře a zeptáme se komponenty, zda bylo vytváření nové události zrušeno. Pokud vytváření nebylo zrušeno, tak si uživatel přeje vytvořit novou událost. Požádáme komponentu tedy o nově vytvořenou událost. V tento okamžik událost existuje pouze lokálně na straně klienta. Pokud bychom nově vytvořenou událost neuložily do persistentního úložiště, byla by událost po ukončení programu nenávratně ztracena. Z tohoto důvodu musíme událost uložit do persistentního úložiště.

Na objektem implementující rozhraní *VEventManager* zavoláme metodu *createVEvent()* a jako parametr ji předáme vytvořenou událost. *VEventManager* vytvoří objekt *VEventTask*, který se postará o uložení události do persistentního úložiště. Všechny metody volané do tohoto okamžiku byly volány synchronně. Nad objektem *VEventTask* je asynchronně zavolána metoda *createVEvent()* s parametrem nově vytvořené události. Uživatel tak může dál pokračovat v práci, aniž by byl ovlivněn zpožděním nutným pro přenos objektů po síti a jejich uložení do databáze.

Následující volání probíhají opět synchronně *VEventTask* zavolá metodu *createVEvent()* nad objektem třídy *RemoteDataSourceImpl*, který se postará o převedení volání metody na síťové volání. Na straně serveru je síťové volání zpracováno objektem *EntryServerDataSource*, který síťové volání převede opět na volání metody. Tento objekt zavolá *createVEvent()* s parametrem nově vytvořené události nad datovým zdrojem. V našem případě je datový zdroj reprezentovaný třídou *DatabaseDataSource*.

DatabaseDataSource obsahuje referenci na objekt třídy *EntityManager*. Následně objekt *DatabaseDataSource* zavolá metodu *save()* s parametrem nově vytvořené události nad *EntityManagerem*. *EntityManager* se postará o serializaci objektu události. *EntityManager* poté uloží objekt v serializované podobě pomocí SQL příkazu *INSERT*

do databáze. Na závěr je jako návratová hodnota propagován kód se stavem, jak požadovaná operace uložení do perzistentního úložiště skončila, a případně i načtená data z databáze. Návratová hodnota je propagována až k objektu *VEventTask,* který zajistí následnou synchronizaci lokálních dat.



Obrázek 11: Sekvenční diagram – Vytvoření nové události

4.8 Návrh uživatelského rozhraní

Vzhledem k omezenému prostoru jsem se rozhodl pro nastínění pouze návrhu hlavního okna aplikace. Člověk obvykle nejvíce věnuje pozornost oblasti v úrovni jeho očí. Typické a také doporučované umístění monitoru je kolmo proti očím uživatele. Z tohoto důvodu uživatel vnímá nejvíce střed monitoru. Na základě těchto faktů jsem se rozhodl hlavní okno aplikace rozdělit na čtyři samostatné části. Po levé a pravé straně okna se nachází sekundární funkce a funkce platné v každé situaci. Do střední části je vždy umístěna primární funkce aplikace. Poslední částí je pak menu, kde se nachází málo využívané funkce.

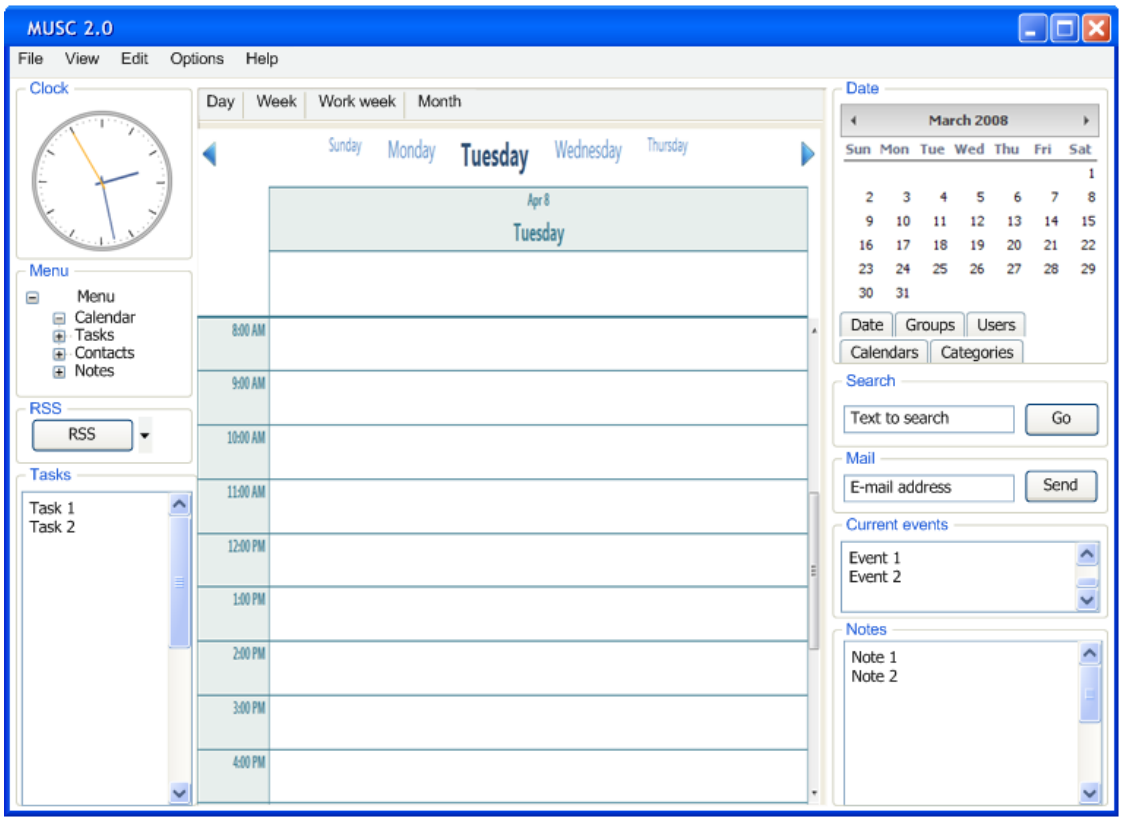
Ve střední části se vždy nachází aktuální primární funkce. Příkladem budiž např. tvorba časového plánu, správa kontaktů nebo poznámek. Samotné primární funkce musí být z pohledu uživatele co nejvíce intuitivní. Na základě toho jsem se rozhodl pro co největší podporu DnD v primárních funkcích. Příkladem je např. možnost přesunutí události mezi dny pouhým „chycením“ události a jejím upuštěním do nového dne. Dalším příkladem je změna počátku nebo konce události „chycením“ události a jejím protažením či zkrácením. Uživateli je nabídnuto několik pohledů na jeho časový plán k zvýšení přehlednosti práce s kalendářem.

Kontakty člověk používá pro rychlé zjištění základních informací o nějaké osobě. U kontaktů máme často přiřazenou celou řadu informací, ale ve většině případů potřebujeme znát pouze telefon nebo e-mail. Vzhledem k této skutečnosti jsem se rozhodl pro víceúrovňové zobrazení kontaktů. V nejvyšší úrovni jsou uživateli zobrazeny pouze nejpodstatnější informace, jako je jméno, preferovaný e-mail, telefon a datum narození. V druhé úrovni se může uživatel dozvědět již více údajů např. telefon a e-mail domů a do práce, zaměstnavatele, pracovní pozici či URL osobní stránky. Poslední úroveň nabízí kompletní seznam informací jako je seznam všech telefonů a e-mailů, adresa domů a do práce, informace o rodině atd.

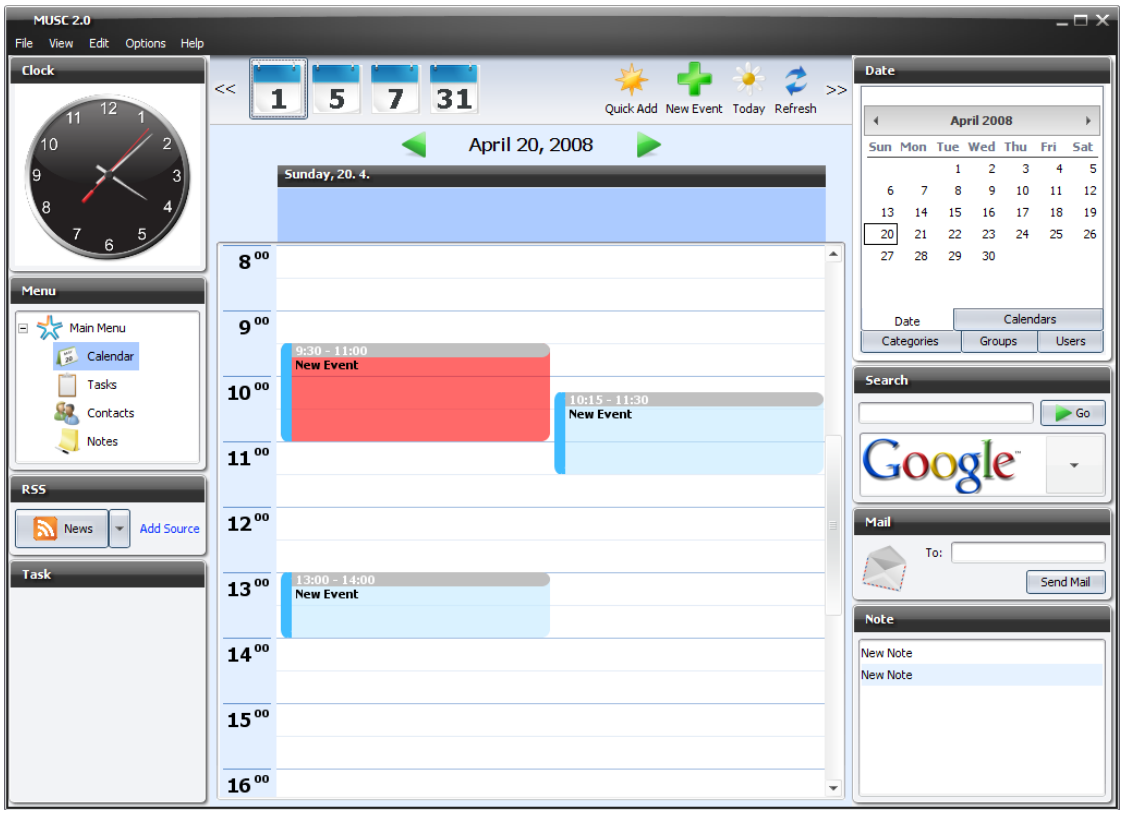
V horní části levého panelu jsou umístěny hodiny, aby uživatel vždy věděl kolik je hodin. Pod hodinami v úrovni očí se nachází jednoduché menu, které slouží pro přepínání primární funkce ve střední části. Pod menu se pak nachází jednoduchá RSS čtečka, kde jsou jednotlivé zdroje zobrazeny pomocí pop-up menu. Pod RSS čtečkou je pak volný prostor, který by mohl být využit pro přehledovou funkci, např. seznam úkolů.

V horní části pravého panelu se nachází záložkový panel. Tento panel slouží pro zprostředkování podpůrných funkcí pro primární funkci kalendáře. Zde lze najít např. mini kalendář, seznam skupin, kterých je uživatel členem, přehled definovaných kategorií atd. Tyto seznamy zároveň slouží jako filtry pro definování množství zobrazovaných informací v pohledu kalendáře. Pod tímto panelem se pak nachází panel pro rychlé vyhledávání na internetu. Poté následuje jednoduchý panel na poslání e-mailu. V dolní části se nachází opět volný prostor, který je možné využít opět pro přehledovou funkci. Zde by mohl být umístěn např. přehled poznámek nebo aktuálně probíhajících událostí.

Poslední částí je pak menu. V menu se nachází funkce pro specifické účely, jako je např. import či export kalendáře, správce RSS zdrojů atd. Návrh GUI je na obrázku 12. Na obrázku 13 je vidět design výsledné aplikace.



Obrázek 12: Návrh uživatelského rozhraní



Obrázek 13: Uživatelské rozhraní aplikace MUSC

4.9 Implementační prostředí

Důležitou otázkou při analýze návrhu je volba vhodných implementačních prostředí. V našem případě se jedná o programovací jazyk, aplikační server a databázový server.

4.9.1 Výběr programovacího jazyka

Otázka výběru vhodného programovacího jazyka, který bude následně použit pro implementaci, je velmi důležitá a nesmí být podceněna. Shrňme si nyní požadavky na aplikaci, které by mohli ovlivnit výběr jazyka.

Za prvé se má jednat o třívrstvou aplikaci – klient, server a databáze. Architektura by měla umožnit výměnu jednotlivých vrstev za nové a mít podporu pro budoucí vývoj rozmanitých prezentačních vrstev (desktop, web či mobilní telefon). V tento okamžik lze zcela jistě vyřadit jazyky jako je PHP nebo Ruby.

Za druhé a především se má jednat o multiplatformní aplikaci. Zde lze vyloučit jazyk jako je C či C++. Ačkoliv se jedná v základu o multiplatformní jazyky, není v nich možné vytvořit zcela multiplatformní aplikaci s GUI (Graphical user interface) a je nutné vytvářet vždy verze pro konkrétní platformy. Tento fakt tedy výrazně omezuje možnost tvorby desktopových a mobilních klientů.

V tento okamžik zde zbývají pouze možnosti v podobě interpretovaných jazyků jako je Java, C# či Python. Jazyk C# má však jednu podstatnou nevýhodu, a to že GUI aplikace je postavené na Windows Forms, které nejsou platformě nezávislé. V tomto případě přichází tedy v úvahu Open Sourcová implementace jazyka C# a to Mono. Mono využívá pro tvorbu GUI knihovny z projektu Gtk+. Běhové prostředí Gtk+ je přítomné dnes již snad ve všech distribucích Linuxu. Bohužel na operačním systému Windows se s ním lze setkat velmi sporadicky. Dalším důvodem pro zavrhnutí Mona a C# a je fakt, že implementace GUI v Monu není na dostatečné úrovni a trpí celou řadou technických problémů. S přihlédnutím k vlastním zkušenostem jsem se rozhodl pro programovací jazyk Java. Vzhledem k aktuálnímu zaměření společnosti Sun Microsystems na rozvoj technologie Swing, která je zaměřena na tvorbu GUI v jazyku Java, spatřuji tento krok za perspektivní do budoucna.

S ohledem na budoucnost a na aktuální stav platformy Java jsem se rozhodl pro verzi 1.6.0. Tato verze přinesla řadu vylepšení s ohledem na vývoj desktopových aplikací. Byla např. přepsána OpenGL pipeline pro referování Java2D, což přineslo velké výkonnostní zlepšení, větší přiblížení nativním aplikacím ať již vzhledem nebo funkcionalitou, např. tray icon, atd. Mnohem větší změny nás však čekají v updatu 1.6.0_10, kde bude přepsána Direct3D pipeline, nový deployment plugin nebo tzv. microkernel.

Na základě uvedených faktů se domnívám, že programovací jazyk Java je velmi vhodný pro implementaci této diplomové práce z několika důvodů. Tím hlavním je multiplatformnost. Dalším podstatným důvodem je zcela jistě typová kontrola při práci s objekty a bezpečnost při práci s pamětí. V neposlední řadě je to i vysoká efektivita programování. Jazyk Java má sice řadu nevýhod, např. rychlost běhu aplikací či paměťové

nároky, ale domnívám se, že klady tohoto jazyka hovoří dostatečně přesvědčivě pro jeho volbu.

4.9.2 Výběr databázového serveru

Na trhu s databázovými servery lze nalézt celou řadu produktů od společností jako je DB2, Microsoft atd. Tyto produkty jsou velmi kvalitní, ale mají také řadu nevýhod. Mezi jejich hlavní nevýhody patří vysoké náklady na pořízení nebo bývají svázány pouze s určitou platformou. V tento moment tedy přicházejí v úvahu Open Sourcová řešení jako je MySQL, Apache Derby nebo PostgeSQL. Na databáze jsou kladeny dva základní požadavky – multiplatformnost a vysoká výkonnost. Na základě srovnání uveřejněného na stránkách *Fermilab Computing Division* **[1]** jsem se rozhodl pro MySQL.

V případě MySQL se jedná o vysoce výkonný relační databázový server, který umožňuje pojmout velké množství dat, aniž by došlo k nějakému znatelnému snížení výkonu. Podstatnou výhodou je fakt, že není svázaný s žádnou platformou, a proto ho lze nasadit na všech doposud známých operačních systémech. Velmi příjemnou skutečností je, že MySQL server je šířen jako Open Source projekt, a proto ho lze pro nekomerční účely využívat zdarma. Tento produkt zcela jistě najde uplatnění všude, kde je potřeba výkonný databázový systém. V současné době se s tímto produktem lze nejčastěji setkat jako databázové řešení pro webové servery. Tato skutečnost je zcela logická, jelikož každá aplikace potřebuje úložiště dat, které bude rychlé, spolehlivé, dokáže pojmout velké množství dat bez znatelné ztráty výkonu a to vše za velmi příznivou cenu. Všechny výše zmíněné požadavky na úložiště dat MySQL databázový server zcela splňuje. V této oblasti si tento produkt již vydobyl významnou pozici.

S ohledem na to, že aplikace bude implementována v jazyku Java, je jistě kladem, že pro MySQL existuje kvalitní JDBC konektor. Konektor existuje ve dvou verzích 3.x a 5.x. Bohužel konektor verze 5.x není v současnosti příliš stabilní a je nutné prozatím využívat verzi 3.x. Mezi kladné vlastnosti určitě také patří snadná správa a konfigurace tohoto serveru. Výrobce navíc poskytuje zdarma kvalitní dokumentaci a příjemný prohlížeč záznamů v databázi.

4.9.3 Výběr aplikačního serveru

Na trhu existuje velká řada aplikačních server. Jelikož jsou všechny napsány v Jave, je zde splněna podmínka multiplatformnosti. Mezi nejznámější jistě patří WebLogic Server (BEA), JBoss, WebSphere AS (Application server) a GlassFish AS. Bohužel ne všechny jsou zdarma dostupné, a tak v potaz přichází pouze JBoss a GlassFish.

Aplikační server GlassFish je založen na Sun Java System AS. V současné době je GlasshFish Open Sourcový projekt podporovaný firmou Sun Microsystems a jedná se o referenční implementaci JEE. JBoss je také Open Sourcový projekt podporovaný naopak firmou IBM. JBoss je založen na webovém serveru Tomcat.

Jedním z požadavků na systém bylo vysoké zabezpečení. Oba dva servery samozřejmě podporují technologii SSL. Bohužel u JBossu se např. konfigurace SSL v rámci

technologie EJB měnila téměř v každé verzi. Naopak u AS GlassFish je konfigurace stabilní a neměnná. GlassFish navíc poskytuje příjemné webové rozhranní pro management samotného serveru. JBoss také nabízí webové rozhranní, ale to již není tak přívětivé na správu.

Další podmínkou výběru byla také implementace technologie EJB 3.0. Zatímco u GlashFishe byla implementace kompletní, u JBosse byla dlouhou dobu implementace nekompletní a ve fázi testování. Na základě těchto důvodů a vlastních zkušeností s oběma servery jsem se rozhodl pro AS GlassFish.

5 Implementace

V této kapitole se pokusím popsat problémy, se kterými jsem se setkal při implementaci projektu a nastíním jejich možné řešení. Nejdříve se pokusím osvětlit co je to datový zdroj a práci s ním. Dále se budu zabývat volbou síťové technologie a otázkami souvisejícími se zajištěním perzistence dat. Následně se pokusím vysvětlit techniku správy grafických komponent a obalujících objektů. Samostatná kapitola je pak věnována bezpečnosti a jejímu zajištění. Také se budu zabývat technikou vykreslování objektů, která je použita ve všech mnou navržených grafických komponentách. Následně se budu zabývat otázkou zajištění multiplatformnosti. Na úplný závěr se pokusím osvětlit práci s rozvrhem z KOSu.

5.1 Datové zdroje

Nejdříve je třeba definovat, co znamená pojem datový zdroj. Datový zdroj je takový objekt, který poskytuje předem definované služby. Tyto služby jsou definovány pomocí interfacu. Z logických důvodů je jasně, že jiné služby budou požadovány pro správu uživatelů a jiné pro správu uživatelem vytvořených dat, např. nový kontakt. Z toho důvodu existují dva základní datové zdroje – *UserSource*, který definuje služby pro správu uživatelů, a *DataSource*, který definuje služby pro správu uživatelských dat.

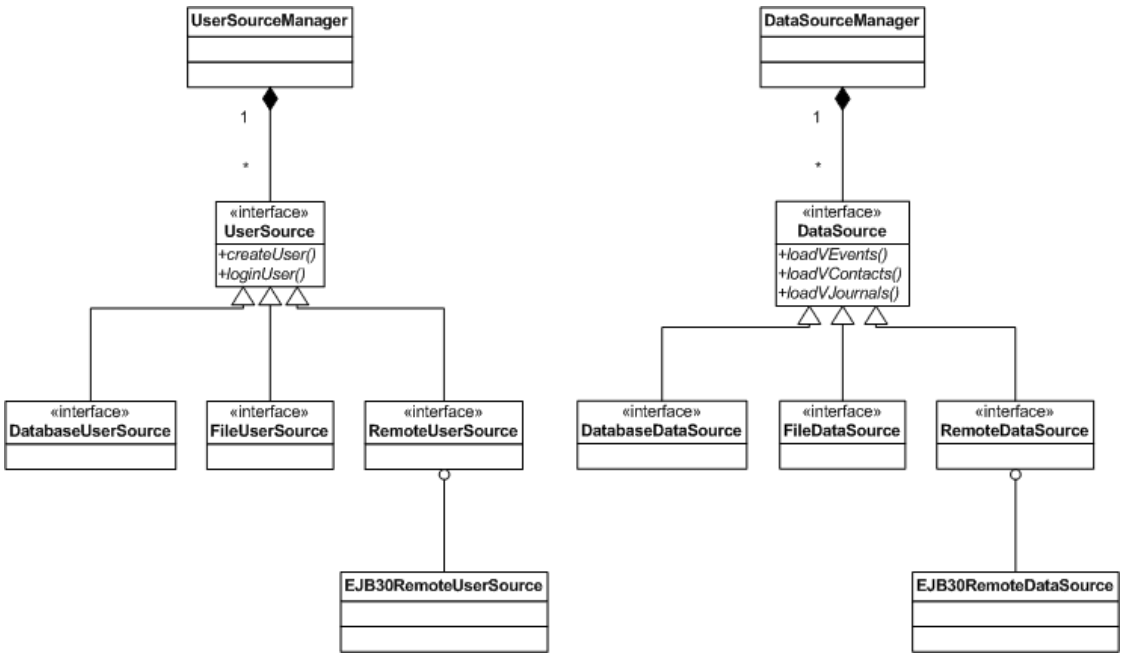
Je zřejmé, že může existovat celá řada zdrojů. Pro možnosti kategorizace zdrojů vznikla vrstva interfaců, která jednotlivé zdroje dělí, dle typu poskytovatele dat. Tyto interfacy jsou pak potomci definovaných zdrojů a nepřidávají žádnou další funkcionalitu. Existují tři základní kategorie poskytovatelů – databáze, soubor a síťový zdroj. Databázový zdroj může být např. embedded databáze nebo databázový server. Tato kategorie je reprezentována *DatabaseUserSource* a *DatabaseDataSource*. Souborovým zdrojem může být libovolný textový soubor, XML soubor atd. Tento typ zdrojů je reprezentován *FileUserSource* a *FileDataSource*. Posledním typem poskytovatele dat je síťový zdroj. Třída, která tento interface implementuje, pak bude zřejmě převádět lokální volání metod na síťová volání a postará se o serializaci a deserializaci objektů na straně klienta. Tato kategorie je reprezentována interfacem *RemoteUserSource* a *RemoteDataSource*. Jelikož tyto rozhraní definují pouze kategorie a nepřidávají žádnou funkcionalitu, je možné je zcela vynechat.

Nyní nastává fáze samotné implementace interfaců. V našem případě jsem se rozhodl pro distribuovanou architekturu a jako technologii jsem zvolil EJB 3.0, více v kapitole *Komunikace v distribuovaném prostředí*. Implementoval jsem tedy rozhraní *RemoteUserSource* třídou EJB30RemoteUserSource. Pro správu uživatelských dat jsem implementoval interface *RemoteDataSource* v třídě EJB30RemoteDataSource.

V rámci datového modelu existují oblasti, které nejsou spolu nijak pevně provázány, např. poznámky a události. V tento moment se nabízí otázka umožnění souběžného využívání více zdrojů pro jednotlivé oblasti. Možnost současného využívání více zdrojů umožňuje vytvoření záložních zdrojů pro případ výpadku. Protože existují dva základní typy zdrojů, je zcela namístě existence nezávislých správců zdrojů pro každý typ zdroje. To nám v budoucnu umožní provádět např. autentizaci uživatelů vůči LDAP serveru a data získávat naopak

z databáze. V našem případě se jedná o *UserSourceManager*, který se stará o správu uživatelských zdrojů, a *DataSourceManager*, který se stará o správu datových zdrojů.

Oba dva správci zdrojů obsahují objekt typu Map. Tento objekt slouží k uchovávání jednotlivých zdrojů. Každý zdroj je v rámci každého správce zdrojů jednoznačně identifikován klíčem. Klíčem může být libovolný objekt, který má definovanou hashovaní funkci *hashCode()*. Pro usnadnění práce se zdroji, poskytují správci zdrojů možnost definovaní tzv. defaultního zdroje v rámci každého správce. Diagram tří datových zdrojů je vidět na obrázku 14.



Obrázek 14: Diagram datových zdrojů

5.2 Komunikace v distribuovaném prostředí

Jako distribuovaný model byla zvolena varianta klient – server. Tento model byl upřednostněn před modelem peer-to-peer s ohledem na jednodušší implementaci a preferovanost v současných technologiích. Následně bylo nutné se rozhodnout pro technologii, která zajistí komunikaci mezi klientem a serverem. V tento okamžik se naskytly dvě varianty – vlastní implementace serverové části nebo vhodný middleware.

Pro první variantu, vlastní implementace serveru, hovořila především jednoduchost použitých síťových technologií. Zde se nabízejí technologie, jako jsou sockety nebo RMI. Protože by však bylo nutné provést implementaci zcela od začátku, byla by zde velká pravděpodobnost možného vzniku chyby při návrhu, která by mohla znemožnit rozšiřitelnost systému v budoucnu. Zároveň je třeba vzít v potaz stabilitu a robustnost serverové části s ohledem na velký počet souběžně aktivních klientů.

Druhou variantou pak bylo využití některé již existující middlewarové technologie. Zde se nabízí jednoznačně využití Javovského aplikačního serveru, více v kapitole *Výběr aplikačního serveru*. Tato varianta nám zabezpečí stabilitu a robustnost serverové části. Navíc tyto technologie jsou již připraveny na zpracování velkého množství souběžných požadavků,

mají implementovány technologie pro zabezpečení komunikace a obsahují podporu správy transakcí. Nevýhodou je naopak velká složitost těchto řešení a nutný čas pro jejich zvládnutí. Aplikační servery, ale nebývají spjaty s jednou technologií pro přenos dat, ale nabízejí více variant, např. HTTP, Corbu, WebServices atd, což nám umožní dobrou rozšiřitelnost v budoucnosti.

Na základě uvedených faktů byla vybrána druhá varianta. Nyní se však bylo nutné rozhodnout pro některou technologii pro přenos dat. Jelikož datové zdroje, více v kapitole *Datové zdroje*, pracují s objekty, byla hned zavrhnuta varianta komunikace pomocí čistého HTTP protokolu. V potaz přicházeli dvě varianty WebServices a EJB. Technologie WebServices v podobě definované v JAX-WS 2.0 (JSR 224) umožňuje posílání pouze primitivních datových typů. Nutností by tedy bylo využití některého frameworku nad základní implementací umožňují posílání objektů, jako je např. Axis2 či XFire. Nevýhodou této technologie, je její pomalost, protože objety jsou serializovány pro přenos do XML, které je následně přenášeno přes HTTP protokol.

Naproti tomu technologie EJB je vystavěna nad technologií Corba, která je primárně určena pro přenos objektů. Navíc data jsou přenášena v binární podobě, což zajistí menší objem přenášených dat a nižší síťové zpoždění. Proto byla pro přenos zvolena technologie EJB ve verzi 3.0.

V rámci technologie EJB jsou definována veřejná rozhraní, která umožňují následné vzdálené volání metod tohoto rozhraní na klientovi. Jako veřejné rozhraní pro správu uživatelů, byl definován interface *EntryServerUserSourceManagerRemote*, jehož struktura vychází z definice interfacu *UserSource*. Obdobně pro uživatelská data je definován interface *EntryServerDataSourceManagerRemote*,který vychází z rozhraní DataSource.

5.3 Práce s daty v distribuovaném prostředí

Protože se jedná o interaktivní program, není možné zanedbat síťové zpoždění. Toto zpoždění by se nejvíce projevovalo u práce s kalendářem, kdy např. každý přechod mezi týdny by představoval síťové volání a uživateli by další týden byl zobrazen až v okamžiku doručení dat ze serveru. Tato varianta je však neakceptovatelná.

Tento problém byl řešen lokální cacheí na klientovi s inteligentním před nahráváním dat. Program udržuje přednahraná data na určitý počet měsíců nazpět a dopředu od aktuálního měsíce. Při každém požadavku na data je specifikováno pro jaký časový interval jsou data požadována. Pokud jsou data pro dané období přítomna, jsou poskytnuta uživateli ihned. Zároveň jsou porovnány měsíce z krajních stran požadovaného intervalu s aktuálně přednahranými měsíci. Pokud se požadovaný interval blíží k maximálnímu nebo minimálnímu přednahranému měsíci, je vyslán asynchronní požadavek na server o nová data. Od serveru jsou požadována data pro nový interval, který vznikne posunutím aktuálního přednahraného intervalu dopředu či vzad podle toho, které krajní hodnoty intervalu bylo dosaženo. Tento systém umožňuje nerušenou práci uživatele, bez obtěžování zdlouhavým nahráváním dat. Nevýhodou této varianty je, že uživatel díky cachování může mít stará data. Uživatel má tedy vždy k dispozici možnost okamžité aktualizace, kdy jsou ze serveru vyžádána aktuální data.

Představme si následující situaci. Uživateli se po přihlášení defaultně v kalendáři zobrazí týdenní pohled na aktuální týden. Následně uživatel přepne na měsíční pohled. V tomto pohledu se postupně posune o dva roky napřed. Následně přepne na týdenní pohled. V tuto chvíli však nastává problém, protože v cachei jsou nahraná data na období za dva roky, ale týdenní pohled požaduje data na aktuální týden. V tento moment se zjistí, že požadovaná data nejsou v předem nahraném intervalu a ani posun vpřed, či vzad situaci neřeší. Na základě takto zjištěných skutečností je na server vyslán požadavek na data z uživatelem požadovaného intervalu. Po dodání dat ze serveru, je pohled přepnut a uživateli jsou poskytnuta aktuální data.

Přednahraná data jsou uchovávána ve třídě *DataManager.* Tato třída je tzv. *singleton*. Jednotlivé komponenty však nikdy nekomunikují s objektem této třídy přímo. Každá komponenta je určena pro zobrazování specifických dat, např. *VJournalPanel* slouží pro zobrazení objektů *VJournal* nebo *VEventMonthGridPanel* slouží pro zobrazování objektů *VEvent*. Na každou skupinu objektů jsou kladeny specifické požadavky na práci s nimi. Z tohoto důvodu byla vytvořena sada rozhraní pro každou skupinu objektů. Komponenty pak pracují s objekty implementující požadované rozhraní. Tyto rozhraní definují operace nutné pro správu konkrétního typu objektů. Např. pro správu události existuje *VEventManager* nebo pro správu kontaktů *VContactManager*. Třída, která jednotlivá rozhraní implementuje, musí zajistit odpovídající funkcionalitu. Rozhraní definují jak operace nad lokální cacheí, tak i nad datovými zdroji. V našem případě většinu těchto rozhraní implementuje již zmíněný *DataManager*. Existence rozhraní a to, že komponenty pracují právě s objekty implementující rozhraní, nám umožňuje v budoucnu vytvořit novou implementaci a nahradit, tak stávající implementaci v podobě *DataManagera*.

5.4 Perzistence dat

V současnosti jsou nejrozšířenějším typem databází tzv. relační databáze. Na základě faktů uvedených v kapitole *Výběr databázového serveru*, byla jako úložiště dat vybrána relační databáze MySQL. Zde však nastává problém ve způsobu práce s daty, zatímco databáze ukládá data do relací, v aplikaci se pracuje s objekty. Bylo tedy nutné řešit problém konverze relací na objekty. Existovala dvě možná řešení. Prvním je vlastní konverze relací na objekty a druhým je využití nějakého ORM *frameworku*.

První případ by znamenal zajištění komunikace přes JDBC. Data by byla z databáze získána ve formě relací. Následně by se musely vytvořit příslušné objekty a naplnit daty. Největší problém by však byl se zajištění vztahů mezi objekty plynoucích z doménové analýzy, viz *Doménová analýza a doménový model*. Dále by byly kladeny velké nároky na zajištění správy veškeré komunikace a transakčního zpracování.

Druhou variantou je pak využití ORM *frameworku*. Výhodou je, že *framework* zajistí nejen konverzi mezi relacemi a objekty, ale hlavně zajistí naplnění vazeb mezi objekty. Další nespornou výhodou je pak, že se postará o správu komunikace s databází. Zároveň nám umožní vytvářet dotazy nad daty v objektově orientovaném jazyce a nenutí nás užívat SQL. Nevýhodou pak může být výkonnostní ztráta daná vložení další vrstvy. Vezmeme-li však v potaz, že přemapování představuje většinu ztraceného času a skutečnost, že v případě

nepoužití *frameworku* by ORM mapování probíhalo také, ale ručně, je možné výkonnostní ztráty zanedbat.

Na základě uvedených fakt byla vybrána varianta s ORM *frameworkem*. Na trhu existuje celá řada produktů. Pokud bych se však rozhodl pro nějaký konkrétní, tak by to mohlo ovlivnit budoucí rozvoj aplikace, protože by se tak stala závislá na konkrétním produktu. Z tohoto důvodu jsem se rozhodl pro JPA (JSR 220). JPA je standardem definované API pro zajištění ORM. Programátor si pak může vybrat konkrétní implementaci, dle jeho přání. Tato volba mi umožní, využití ORM mapování a nestát se závislý na konkrétním produktu. Jako implantaci jsem zvolil nerozšířenější ORM *framework* Hibernate.

Na server přichází požadavky, které téměř vždy končí komunikací s databází. Často přijde velké množství požadavků současně. Jednotlivé požadavky jsou zpracovány v samostatných transakcích s využitím JTA (Java Transaction API). Pokud se sejde více požadavků na zápis stejné hodnoty, tak jsou požadavky serializovány v čase podle okamžiku jejich příchodu a poslední zapisující definuje novou hodnotu.

Distribuovaný charakter aplikace klade velká omezení na výhody dané ORM *frameworkem*. Jak již jsem zmínil, tak podstatnou výhodou je zajištění vazeb mezi objekty. Protože vazby mezi objekty mou být velmi rozsáhlé, je tato problematika řešena tzv. *lazy* inicializací. Příkladem budiž objekt *Osoba*, která má jako instanční proměnou kolekci svých dětí. Při nahrání objektu *Osoba* z databáze nejsou objekty jeho dětí nahrány. Teprve až při požadavku na ně jsou objekty typu *Dítě* nahrány. Toto dohrání dat je provedeno samotným ORM *frameworkem*. Existuje zde však i možnost nahrát všechna data na začátku najednou. To však může při složitých vazbách znamenat extrémně velké přenosy dat. S ohledem na distribuovanou povahu aplikace a skutečnost, že rozhraní mezi klientem a serverem je definováno technologií EJB s definovaným interfacem a ne samotným ORM nástrojem, je nutné vyřešit tento problém.

V zásadě existují dvě možná řešení – definování *proxy* objektu nebo omezení vazeb mezi objekty. Prvním řešením je definování *proxy* objektu. Požadovaný objekt by byl přenesen ze serveru na klienta s *lazy* inicializací některých jeho atributů. Při požadavku na atributy, které nebyly doposud načteny, by *proxy* objekt převedl lokální volání na síťové volání a zajistil by dotažení požadovaných objektů po síti. Toto řešení je nejčistší z pohledu návrhu a doménového modelu, ale má řadu nevýhod.

Za prvé by *proxy* objekt musel být definován pro každý typ objektu, který je ukládán do databáze. Spolu s nutností implementace, by to znamenalo velké svázání s konkrétní technologií pro síťovou komunikaci. Obě tyto sktečnosti by mohli vést k omezení rozšiřování v budoucnu nebo náhrady jednotlivých vrstev za jiné. Dalším problémem je zpoždění síťové komunikace nutné pro dotažení objektů, což by mělo vliv na dobu reakce na požadavek uživatele. V neposlední řadě by došlo k téměř úplnému potlačení serverové vrstvy a z třívrstvé aplikace by se stala pouze dvojvrstvá – klient a databáze.

Druhým možným řešením je omezení vazeb mezi objekty na nezbytné minimum. Zejména je nutné odstranění cyklických závislostí a ponechání pouze jednosměrný vztahů. Příkladem budiž objekt *VUser*. U tohoto objetu odstraníme všechny reference na objekty,

jako jsou jím definované kalendáře nebo vytvořené události. Naopak u objektu události *VEvent* musíme ponechat referenci na uživatele, který ji vytvořil. V podstatě simulujeme způsob tvorby referencí pomocí cizího klíče v relačních databázích. Takto upravené objekty, již lze načíst kompletně se všemi vztahy odkazující na další objekty. Takto načtené objekty pak lze kompletně odeslat ze serveru na klienta. Výhodou tohoto řešené je zachování serverové vrstvy. Nevýhodou pak je samozřejmě omezení vztahů mezi objekty a možná neaktuálnost objektů na straně klienta.

V okamžik, kdy jsou načtené objekty odeslány ze serveru na klienta, přecházejí ze stavu *persistent* do stavu *detached*. Stav *detached* nastává, pokud objekt s persistentním obrazem v databázi je odpojen od *session* s databází. Na klientovi se pak pracuje s objekty ve stavu *detached*, které je možné měnit. Změny se však samozřejmě nepromítají do databáze. Pokud je třeba objekt updatovat v databázi nebo ho smazat, tak je odeslán na server. Zde musí dojít nejprve k tzv. *reattached* s aktuální *session* do databáze. Poté může být objekt updatován nebo smazán z databáze.

Kromě stavu *persistent* a *detached* existuje ještě stav *transient*. V tomto stavu jsou objekty, které byly vytvořeny pomocí klíčového slova *new* a nemají tak přidělený identifikátor a svůj persistentní obraz v databázi. Toto je typická situace při vytvoření nové události. S ohledem na bezpečnost persistentních dat je nutné pro nově vytvořený objekt události vytvořit persistentní obraz. Zde se však opět objevuje problém se zpožděním síťové komunikace.

Tento problém jsem se rozhodl řešit pomocí asynchronního volání metod. Uživatel vytvoří nový objekt události. Následně je vyslán asynchronní požadavek na server pro vytvoření persistentního obrazu nového objektu. Na serveru je objektu přiřazen identifikátor a vytvořen persistentní obraz. Uživatel tak může dál pracovat, aniž by byl ovlivněn síťovým zpožděním. Uživatel pracuje s objektem bez přiřazeného identifikátoru. Co by se však stalo, pokud by se uživatel pokusil modifikovat, takovýto objekt? Pravděpodobně by v databázi vznikl nový objekt a původně uložený objekt by se nezměnil. To je však neakceptovatelné. Při vytvoření objektu na klientovi, je objektu přiřazen dočasný lokální identifikátor a objekt je označen za dočasný. Před odesláním je dočasný objekt uložen do *TempSynchronizationManagera*. Poté co je na serveru objektu přiřazen identifikátor a vytvořen persistentní obraz, je objekt odeslán zpět na klienta. Tam je pomocí *TempSynchronizationManagera* sesynchronizován stav mezi objektem na klientovi a příchozím objektem ze serveru.

V aplikaci probíhá celá řada asynchronních volání, což vede k zvýšeným nárokům na systémové prostředky v podobě práce s vlákny. Z tohoto důvodu vznikla třída *GlobalThreadPool*. Tato třída slouží jako pool pro vlákna a k spouštění asynchronních úkolů. V poolu jsou vlákna udržována po určitou dobu. Pokud není vlákno po určitou dobu použito, tak je uvolněno z poolu. V případě požadavku na vlákno, tak pokud je dostupné vlákno v poolu, tak se použije, jinak se vytvoří nové vlákno.

5.5 Správa grafických komponent

Grafické komponenty jsou jednou z paměťově nejnáročnějších částí aplikace. Dalším problémem je pak často velmi dlouhá doba jejich prvotní inicializace a vykreslení. Tento problém jsem se rozhodl řešit pomocí sdílení zdrojů a *lazy* inicializace.

Lazy inicializace nám zajistí vytvoření komponenty až v okamžik jejího skutečného použití. Díky tomu není nutné blokovat systémové prostředky. Druhým prostředkem je sdílení zdrojů. Za tímto účelem existuje třída *ScreenManager* a *enum Screen*. *Enum Screen* obsahuje seznam všech grafických komponent. Naopak *ScreenManager* slouží k tvorbě a správě komponent.

ScreenManager poskytuje dvě základní metody *getNewScreen(Screen screen)* a *getShareScreen(Screen screen)*. Metodě *getNewScreen()* je jako parametr předán typ požadované komponenty a na základě tohoto typu vytvoří vždy novou instanci požadované komponenty. Druhá metoda *getShareScreen()* opět vrátí požadovanou komponentu na základě požadovaného typu. Tato metoda ale vrací tzv. sdílené komponenty, což jsou komponenty, o které může požádat kdokoliv, ale vždy dostane tutéž instanci dané komponenty. Nejdříve se metoda podívá do *bufferu*, zda již požadovaná komponenta existuje. Pokud již existuje, tak je vrácena komponenta z *bufferu*. Jestliže však komponenta neexistuje, tak je nejprve vytvořena a vložena do *bufferu* a následně je vrácena objektu, který ji požadoval.

Tento systém má tři základní výhody – šetření paměti a dalších systémových prostředků, v kombinaci s *lazy* inicializací pouze jedinou zdlouhavou inicializaci komponenty a v neposlední řadě snadné provázání logiky mezi komponentami.

Další technikou využité při správě komponent je rozdělení komponent na komponenty, do nichž jsou komponenty vkládány a na ty co jsou vkládány. Interface *ScreenInterface* definuje sadu metod, které musí implementovat každá komponenta, do níž se vkládají jiné komponenty. *ScreenInterface* definuje následující metody:

public void insertComponentIntoScreen(Component component); – umožní vložení komponenty do komponenty implementující toto rozhraní

public void insertMenuBarIntoScreen(MenuBar menuBar); – vloží *MenuBar* do komponenty implementující toto rozhraní

public void insertJMenuBarIntoScreen(JMenuBar menuBar); – vloží *JMenuBar* do komponenty implementující toto rozhraní

public void removeAllFromScreen(); – odstraní veškerý obsah z komponenty implementující toto rozhraní

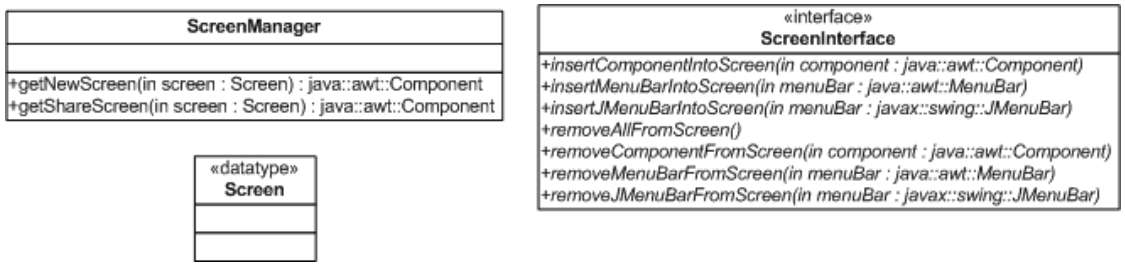
public void removeComponentFromScreen(Component component); – odstraní konkrétní komponentu z komponenty implementující toto rozhraní

public void removeMenuBarFromScreen(MenuBar menuBar); – odstraní *MenuBar* z komponenty implementující toto rozhraní

public void removeJMenuBarFromScreen(JMenuBar menuBar); – odstraní *JMenuBar* z komponenty implementující toto rozhraní

Příkladem budiž komponenta *WizardDialog*, která slouží pro zobrazování dialogů v aplikaci a implementuje rozhraní *ScreenInterface*. Pomocí *ScreenManageru* získám sdílenou instanci *WizardDialogu*. Nad touto instancí zavolám metodu *removeAllFromScreen()* a odstraním možný dříve vložený obsah této komponenty. Poté získám ze *ScreenManageru* sdílenou instanci požadovaného panelů, např. pro vytvoření nové události. Tuto komponentu pak pomocí metody *insertComponentIntoScreen()* vložím do *WizardDialogu*, který pak následně nechám zobrazit zavoláním metody *setVisible(true)* nad dialogem.

Díky kombinaci sdílení komponent, *lazy* inicializaci a interfacu *ScreenInterface* může uživatel velmi rychle pracovat v aplikaci, jelikož zobrazení požadovaného dialogu je otázkou okamžiku bez nutnosti inicializace dané komponenty a alokace systémových prostředků. Diagram tříd komponent pro správu grafických objektů je na obrázku 15.



Obrázek 15: Správa grafických komponent

5.6 Správa obalujících objektů

Obalující objekty slouží k přidání funkcionality entitním objektům. Entitní objekt je např. objekt třídy *VUser*, *VEvent*, *VCalendar* atd. Obalující objekty slouží k zjednodušení práce s entitními objekty. Např. entita *VCalendar* má definovanou barvu pomocí RGB+A složek pro zachování multiplatformnosti. Tento přístup však není vhodný pro práci v aplikaci a vhodnější by byl objekt typu *java.awt.Color*. Z tohoto důvodů obalující objekt *VCalendarWrapper* poskytuje objekt *java.awt.Color* definovaný na základě RGB+A složek z objektu *VCalendar*.

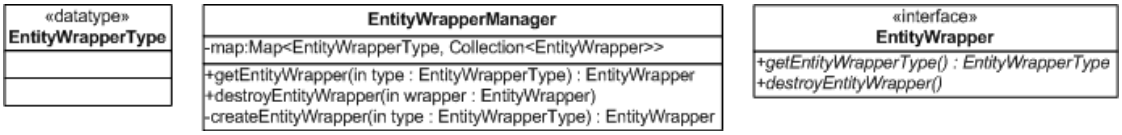
Je zřejmé, že během celého běhu aplikace bude potřeba velké množství obalujících objektů, protože obalující objekty musí být vytvořeny pro každou instanci entitních objektů. Často však entitní instance nemají dlouhou životnost a bylo by tedy vhodné umožnit znovu použitelnost obalujících objektů. Pro správu obalujících objektů slouží třída *EntityWrapperManager* a interface *EntityWrapper*. Každá obalující třída musí implementovat interaface *EntityWrapper*. Tento interface definuje tyto metody:

public EntityWrapperType getEntityWrapperType(); – vrací typ obalující třídy

public void destroyEntityWrapper(); – provede destrukci (vyčištění) obalující třídy na základě jejich specifických vlastností

Třída *EntityWrapperManager* pak slouží ke správě obalujících objektů. *EntityWrapperManager* poskytuje metodu *getEntityWrapper(EntityWrapperType type)*, která vrátí obalující objekt požadovaného typu. Metoda se nejdříve podívá, zda pro daný typ má v kolekci volný objekt obalující třídy. Pokud existuje volný objekt, tak je z kolekce odstraněn a vrácen objektu, který o něj požádal. Pokud však není k dispozici volný obalující objekt, tak je vytvořen nový objekt.

Při běhu aplikace nastávají okamžiky, kdy je zřejmé, že daný entitní objekt již nebude využíván a jeho obalující objekt je možné zničit. Typickým příkladem je přednahrání nových dat a zbavení se již nahraných objektů. V tento okamžik je nad *EntityWrapperManager* zavolána metoda *destroyEntityWrapper(EntityWrapper wrapper)*, kde je jako parametr předán obalující objekt, který má být zničen. Nad tímto objektem je následně zavolána metoda *destroyEntityWrapper()*, která způsobí vyčištění obalujícího objektu. Poté je objekt přidán do kolekce volných objektů daného typu obalujících objektů. Tato technika nám tedy umožní znovu využít obalující objekty bez nutnosti alokace dalších systémových prostředků a zrychlení přidělování obalujících objektů, jelikož již není nutný čas pro jejich vytváření. Na obrázku 16 je vidět diagram tříd objektů pro správu obalujících objektů.



Obrázek 16: Správa obalujících objektů

5.7 Technika vykreslování objektů

Jednou ze základních podmínek pro grafické komponenty je, aby byl jejich obsah plynule a co nejrychleji vykreslován. Zvláště je to nutné např. u DnD či při přechodu mezi týdny u týdenního pohledu v kalendáři. Základní technikou pro zamezení blikání při vykreslování je *double buffering*. Tato technologie je již součástí *frameworku* Swing pro tvorbu GUI v Jave. Podstatnější problém však byl s plynulostí a rychlostí vykreslování.

Při vykreslování jednotlivých objektů se často pracuje s průhledností např. v podobě stínů u vybraných událostí nebo aby byla viditelná časová mřížka, do které událost zapadá. Např. obrazec představující událost má zaoblené rohy, a tak je nutno mít zapnutý *anti-aliasing*. Protože chceme, aby vše působilo přirozeným dojmem a bylo zobrazeno co nejvěrohodněji, je nastavena nejvyšší kvalita renderování. Zároveň je třeba vzít v potaz, že se obvykle bude vykreslovat velké množství objektů. Toto jsou všechno faktory, které významně ovlivňují rychlost a následně i plynulost vykreslování.

Prvním pokusem při vykreslování bylo vykreslení celé plochy vždy znova. Tato technika je velmi jednoduchá, ale absolutně nevyhovující z hlediska výkonnosti. Z tohoto důvodu byla navržena technika kreslení ve vrstvách. Zobrazovaná plocha je vždy rozdělena do samostatných vrstev. Vrstva je pak znovu překreslena pouze v případě, že v ní došlo k nějaké změně. Jednotlivé vrstvy jsou pak složeny a dávají výsledný obrazec.

Uveďme si tuto techniku na konkrétním případu vykreslování týdenního pohledu v kalendáři. Tento pohled se skládá z pěti samostatných vrstev – časová mřížka (pozadí),

vrstva obsahující vykreslené události, DnD vrstva, vrstva pro interakci při pohybu myší po pohledu bez zmáčknutého tlačítka a vrstva při zmáčknutí tlačítka myši. Každá vrstva je kreslena na samostatný průhledný obrázek. Při každém požadavku na překreslení scény jsou tyto vrstvy (obrázky) složeny a vykresleny. Pokud dojde ke změně v nějaké vrstvě je nově překreslena pouze vrstva, v níž došlo ke změně, ostatní vrstvy zůstanou beze změny. Příkladem budiž přechod mezi týdny. Vrstva s pozadím zůstane vždy beze změny, protože události budu zapadat opět do stejné mřížky. Naopak vrstva s událostmi se musí překreslit, protože v novém týdnu jsou i nové události.

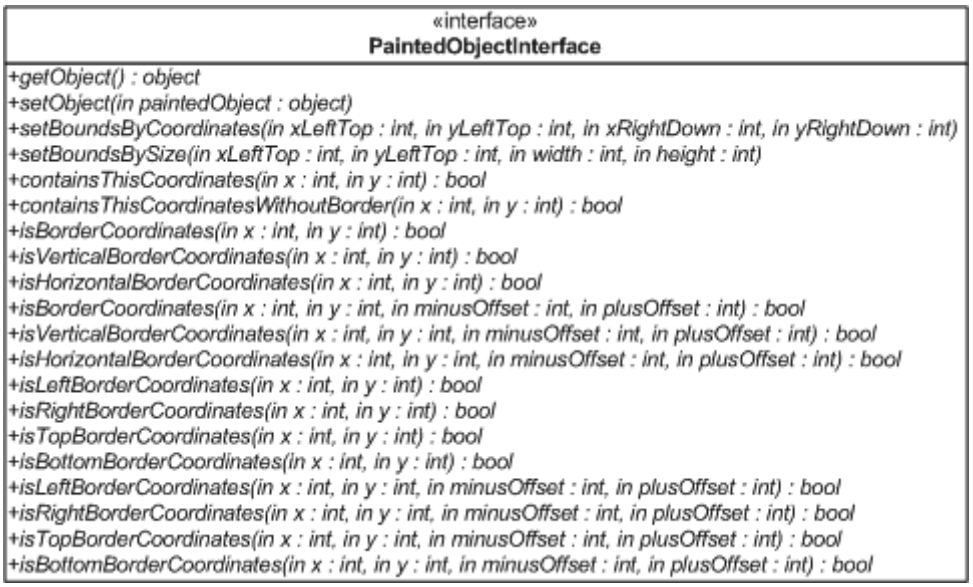
Při pohybu myší po pohledu jsou detekovány jednotlivé události a je zobrazen symbol pro změnu události. V tento moment se nově překresluje pouze vrstva pro pohyb myší po ploše bez stisknutého tlačítka myši a ostatní vrstvy se pouze zkopírují a použije se jejich stará verze. Další příkladem je např. výběr události a zobrazení stínu kolem události symbolizující výběr. Zde opět dochází pouze k překreslení vrstvy pro zmáčknutí tlačítka myši. Pokud však dojde ke změně velikosti celého pohledu (např. maximalizace okna), je nutné překreslit vrstvu pozadí, která je pak následně opět znovu využívána.

U DnD dochází k úplnému znovu využití všech vrstev. Událost, jejíž poloha je měněna, je vykreslena na samostatný obrázek, který vytváří novou vrstvu. V tento moment je nutné zajistit odstranění dané události z vrstvy pro zobrazení událostí. Zde jsou dvě možnosti. První je znovu tuto vrstvu vykreslit, což by však bylo časově velmi náročné. Druhou a lepší variantou je pouze vyčištění plochy, kterou daná událost zabírala ve vrstvě událostí. Tato technika práce s vrstvami je pak použita ve všech grafických komponentách, které byly v projektu vytvořeny.

Další otázkou je pak organizace událostí pro vykreslování, tak aby události ve vykreslovaném plánu vyplňovali prostor co nejkompaktněji a vypořádání se s problematikou souběžně probíhajících událostí. Za tímto účelem byly vytvořeny tzv. *PaintedObjectLine*. Celý princip si vysvětlíme na nejjednodušším příkladu časového plánu pro jeden den.

Časový plán pro jeden je rozdělen do *n* paralelních časových rovin. Tyto roviny slouží pro následné zobrazení paralelně probíhajících událostí. Tato paralelní rovina je reprezentována objektem třídy *PaintedObjektLine*. Na počátku je tato rovina prázdná a poslední potenciálně vložená událost skončila v 0:00. Při pokusu o přidání nového objektu do roviny se musí nejdříve zjistit, zda přidávaná událost začíná až poté co skončila, poslední předchozí přidaná událost. Pokud ano, tak se událost vloží a změní se koncový čas posledně přidané události pro tuto rovinu, pokud ne tak se událost nepřidá do této roviny. Počet těchto rovin v jeden den je určen maximálním počtem souběžně probíhajících událostí. Tyto časové roviny jsou pak sloučeny do *PaintedObjectLines*. Při pokusu o přidání nové události jsou postupně procházeny jednotlivé časové roviny v předem daném pořadí. Pokud lze událost vložit, tak je vložena. Pokud ji však nelze vložit, tak se pokusím událost vložit do další dostupné paralelní roviny. Jestliže už neexistuje další dostupná paralelní roviny, tak je vytvořena nová paralelní rovina. Tento mechanismus zajistí co nejkompaktnější časový plán. Tyto denní linie je možné dále sdružovat např. do týdenních pohledů. Pro měsíční pohled nebyla jako elementární linie zvolena linie pro jeden den, ale celý týden.

Do těchto linií se pak vkládají objekty, jež implementují interface *PaintedObjectInterface*, např. třída *PaintedObject*. Tento interface definuje metody pro uložení vykreslovaného objektu, souřadnic kde je vykreslen, metod pro detekci jeho pozice, pro dané souřadnice *x* a *y*, určí, zda spadají do oblasti, kde byl objekt vykreslen nebo je jeho okraj atd. Charakteristika interfacu *PaintedObjectInterface* je vidět na obrázku 17.

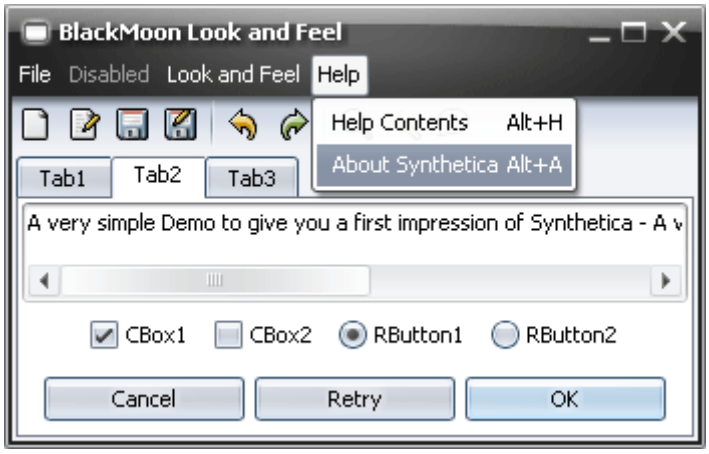


Obrázek 17: PaintedObjectInterface

5.8 Multiplatformnost

Zajištění multiplatformnosti byl jeden ze základních požadavků kladených na celý projekt. Z tohoto důvodů byl zvolen multiplatfomní jazyk Java, aplikační server Glassfish a databázový server MySQL. Na první pohled by se zdálo, že se podařilo celý problém vyřešit, bohužel není tomu tak. Největší problém spočívá totiž v samotné Jave. Existují tu v zásadě dva hlavní typy problémů – platformě závislé chyby nebo rozdílné chování na jednotlivých platformách.

Prvním problémem bylo vytvoření multiplatformní klientské desktopové aplikace, která bude na všech platformách vypadat a chovat se stejně. Aby aplikace vypadala na všech platformách stejně, je v podstatě nesplnitelný úkol a je možné pouze docílit velmi podobného vzhledu. Zde je již základní problém v otázce volby LaF (Look and Feel). Buďto můžeme zvolit variantu, kdy aplikace bude mít na nativní LaF dané platformy nebo použijeme tzv. *cross-platform* LaF. Udělat aplikaci, která by měla nativní vzhled a stále byla multiplatformní, je v podstatě nemožné a skončilo by to pravděpodobně samostatnými verzemi pro jednotlivé platformy. Jako vhodnější se tedy jevila varianta *cross-platformního* LaF. Přímo v Jave je přítomný *cross-platform* LaF Metal, ten však již dnes nesplňuje požadavky na moderní aplikace. Další možnou volbou je LaF Nimbus, který má být nástupcem LaF Metal. Tento LaF má však podstatné nevýhody a to, že je stále ve vývoji a LaF je uplatněn pouze na vnitřek okna a rám okna je vykreslen ve stylu dané platfomy. To např. na OS Windows XP působí velmi neesteticky a rušivě na zbytek aplikace, kdy jako by rám okna přestával být součástí naší aplikace. Z tohoto důvodu jsem zamítnul Nimbus a zvolil LaF Synthetica Black Moon. Tento LaF splňuje všechny zmíněné požadavky. Ukázka LaF Synthetica Black Moon je na obrázku 18.



Obrázek 18: Look and Feel Synthetica Black Moon

Tento LaF dobře fungoval na platfomně Windows XP a Vista. Bohužel na platfomně Linux se objevil problém se zhledem komponenty *javax.swing.JSpinner*. Tento problém způsoboval vznik výjimky, která znemožňovala použití této komponenty. Vzhled je definován pomocí souboru *synth.xml*. V tomto souboru bylo nutné změnit u stylu pro *spinner* atribut *„Synthetica.spinner.arrowButton.width“* na *„Synthetica.spinner.arrowButton.size“*. Po této opravě komponenta již fungovala i na platformě Linux.

Další problém byl však již v samotné Javě a její implementaci pro platformu Linux. Spuštění aplikace vždy vyústilo v tuto zprávu *„xcb_xlib.c:50: xcb_xlib_unlock: Assertion 'c->xlib.lock' failed.“* a aplikace se nespustila. Tento problém je evidován jako chyba číslo 6532373 na http://bugs.sun.com. Je zde problém, že AWT (Abstract Window Toolkit) není připraveno pro použití s XCB (X C Binding). XCB je rozhraní pro jazyk C pro programování X Window Serveru. Tento problém v Jave existuje již od verze 1.4.2. Naštěstí tento problém je již opraven v připravovaném updatu 1.6.0_10b9 a v nové verzi Javy 1.7.0b22.

Větším problémem je však nepodpora *java.awt.Desktop* API ve window manageru KDE (K Desktop Environment), viz chyba číslo 6486393, což mělo za následek nefunkčnost posílání e-mailu či vyhledávání na internetu. V současnosti není známá oprava pro tento problém. Důsledkem je, že v současnosti nelze zajistit podporu platforem s KDE.

Celý systém je vyvíjen na platfomně Windows Vista v anglické jazykové verzi. Při přenosu aplikace na stejnou platformu, ale v české jazykové verzi, došlo k nekorektnímu chování aplikace při práci s datem, např. špatný výpočet týdnů v měsíci. Na obou systémech běžela stejná verze VM (virtual machine) 1.6.0_05. Jak se následně ukázalo, implementace abstraktní třídy *java.util.Calendar* se chovala rozdílně v závislosti na jazykové verzi.

V anglicky mluvících zemích se často za první den v týdnu bere neděle, zatímco v Čechách je jím pondělí. I přes tento fakt má měsíc březen v roce 2008 v obou variantách šest týdnů. Třída *Calendar* nevrací aktuální počet týdnů pro aktuální měsíc, ale pouze minimální a maximální číslo týdne. V anglické jazykové verzi je minimální týden 0. a maximální 6. týden. Naproti tomu v české verzi je minimální týden 0. a maximální 5. Na základě této skutečnosti musel být upraven způsob výpočtu počtu týdnů.

5.9 Rozvrh z KOSu

Rozvrh z KOSu je každý den exportován do XML souboru. Serverová část aplikace má umožněn přístup k tomuto souboru. Tento soubor je následně s využitím SAX parseru (Simple API for XML) převeden do databáze. Více o schéma databáze v kapitole *Návrh datového úložiště rozvrhu z KOSu*. Aby byla práce s importem nového rozvrhu z XML souboru co nejvíce zjednodušena, byla vytvořena speciální komponenta. Tato komponenta je součástí serverové části a pravidelně v předem daných časových intervalech zajistí získání nového rozvrhu. V okamžiku aktivování komponenta stáhne nový rozvrh a importuje ho do databáze. Následně se naplánuje další spuštění komponenty. Pokud nemůže být komponenta v předem určený čas spuštěna, je dodatečně spuštěna v okamžik, kdy to již bude možně. Tato komponenta je implementována třídou *KOSDataTimerServiceBean* s využitím API *Timer Service* v technologii EJB. Tato komponenta je inicializována a poprvé spuštěna prvním uživatelem, který se přihlásí na server.

Exportovaný rozvrh je organizován z pohledu studentů, což se projevuje, tak že přiřazení studentů vůči jednotlivým rozvrhovým lístkům existuje pouze pro studenty. Toto přiřazení však již neexistuje z pohledu pedagogů. To je však nepřijatelné, jelikož je nutné, aby aplikace byla schopná zobrazit jak rozvrh studentů, tak i pedagogů. Naneštěstí u každého rozvrhového lístku je uvedeno *id* pedagoga, který danou hodinu vyučuje. Na základě této informace jsem byl schopen doplnit chybějící přiřazení učitelů na jimi vyučované hodiny.

Aplikace umožňuje zobrazit rozvrh z KOSu všech členů uživatelsky definované skupiny. Při tomto požadavku je nejprve získán seznam všech uživatelů, kteří jsou členy dané skupiny. Tím získám jejich uživatelské jméno, které odpovídá školnímu uživatelskému jménu. Následně je proveden dotaz nad databází, která vrátí všechny hodiny pro všechny požadované osoby. Pokud daný uživatel není student ani pedagog, tak pro něho není vrácen žádný rozvrh. Každá takto získaná hodina odpovídá jedné opakující se události. Podle toho zda se jedná o hodinu každý týden či jen jednou za 14 dní, je upraveno pravidlo pro opakování. Pokud se jedná o událost pouze v liché týdny, tak začíná v prvním školním týdnu. Jestliže se však jedná o událost pouze v sudém týdnu, tak je datum prvního výskytu posunuto na druhý vyučovací týden.

Součástí dat z KOSu bohužel není datum začátku semestru. Z tohoto důvodu má každý uživatel definovanou objekt *VProperty* s nastavením *„kos.semester.start“*, který obsahuje začátek semestru v UTC (Coordinated Universal Time) formátu. Uživatel jsi tak sám definuje začátek semestru. Tato proměnná je samozřejmě uložena v databázi a její změna je nutná vždy pouze jednou na začátku nového semestru. Pokud si nechá uživatel zobrazit rozvrh uživatelské skupiny, je jako počátek semestru zvoleno datum uživatele, který o rozvrh požádal.

Defaultně je uživateli jeho školní rozvrh zobrazen pouze s právem *„read“*, kdy si rozvrh může nechat zobrazit, ale nemůže ho měnit. Uživatel však má možnost přijmout školní rozvrh za vlastní a ten se tak stane součástí jeho kalendáře. V ten moment může školní rozvrh plně modifikovat, ale je také sám zodpovědný za jeho odstranění např. s příchodem nového semestru.

6 Bezpečnost

Otázka bezpečnosti patří dnes k jednomu z nejpalčivějších témat, proto byla problematice bezpečnosti věnována velká pozornost. Bezpečnost v rámci aplikace lze rozdělit na několik samostatných částí – zabezpečení přístupu k uživatelským účtům, okruh povolených uživatelů, zabezpečení síťové komunikace, zabezpečení uživatelských hesel a ze systému uživatelských práv. Jednotlivé oblasti jsou probrány v následujících kapitolách.

6.1 Zabezpečení přístupu k uživatelským účtům a heslům

Pro zabezpečení přístupu k uživatelskému účtu byla zavedena klasická technika pomocí uživatelského jména a hesla. Uživatelské jméno a heslo je uchováváno v databázi. Ačkoliv heslo se nachází v databázi, která není volně dostupná, nelze nikdy zcela vyloučit nebezpečí nepovolaného přístupu k databázi. Z tohoto důvodu nejsou v databázi uložena hesla v originální podobě, ale pouze jejich hashe. Tento hash vzniká v okamžiku vytváření uživatelského účtu na klientské aplikaci. Originální heslo tedy existuje v aplikaci pouze v okamžik vytvoření účtu a poté je nenávratně ztraceno. Navíc originální heslo nikdy neopustí klientskou aplikaci. Jako hashovaní funkce je použito SHA 512.

Jistě všichni víme, že i sebelepší šifra či hash se dá prolomit ať již hrubou silou nebo např. chybou v dané šifře či hashy. Z tohoto důvodu vznikla třída *HashUtilities*. Tato třída slouží k vytváření hashů z daných objektů. Podstatnou výhodou této třídy je skutečnost, že typ hashovaní funkce je zde pouze jako jeden z parametrů a je tedy možné zvolit pro daný okamžik libovolnou hashovaní funkci.

6.2 Autentizace uživatelů z povoleného okruhu uživatelů

Zvláště v prostředí ČVUT je vhodné omezit okruh potenciálních uživatelů pouze na zaměstnance školy a studenty. Možným řešením je provést autentizaci uživatele při vytváření nového účtu proti nějaké autentizační službě. Z tohoto důvodu interface *UserSource* poskytuje metodu *authetication*AgainService(String username, String password)*. Objekt, který tuto metodu implementuje, pak po jejím zavolání provede autentizaci uživatele pomocí uživatelského jména a hesla proti jím zvolené autentizací službě, např. LDAP serveru.

Bohužel na Katedře počítačů v současné době neexistuje žádná veřejná autentizační služba. Z tohoto důvodu je autentizace prováděna přístupem na zabezpečené stránky některého serveru ve škole, např. serveru HardWeb. Uživatel tedy při vytváření nového účtu zadá své uživatelské jméno a heslo na server HardWeb. Po ověření autentičnosti uživatele je z poskytnutého hesla vytvořen hash a heslo v systému již dále neexistuje v původní podobě. Výhodou tohoto řešení je, že uživatel může v rámci aplikace používat stejné uživatelské jméno a heslo jako ve škole a nemusí si tak plavat další uživatelské jméno a heslo.

6.3 Zabezpečení síťové komunikace

Pro zajištění bezpečnosti síťové komunikace existovaly tři varianty – použití symetrických šifer, použití asymetrických šifer nebo jejich kombinace. S ohledem na snadnost správy bylo zvoleno pro zajištění bezpečnosti síťové komunikace asymetrické šifrování v podobě technologie SSL (Secure Sockets Layer). Jako implementace SSL na straně klienta byla zvolena implementace z Java SE. Na straně serveru je poskytovatelem implementace SSL aplikační server GlassFish.

Při vytváření nového účtu je nejprve provedena autentizace uživatele vůči autentizační službě. Z klienta je poslán požadavek na aplikační server přes SSL. Autentizace se provádí na serverové části, tak že je stáhnuta stránka z webu HardWeb, která vyžaduje uživatelské jméno a heslo. Komunikace mezi aplikačním serverem a serverem HardWeb je nezabezpečená. Výsledek autentizace je odeslán nazpět z aplikačního serveru na klienta pomocí spojení, které je chráněno SSL. Veškerá následující komunikace mezi klientskou aplikací a serverovou částí je chráněna pomocí SSL.

XML soubor s exportem rozvrhu z KOSu je stahován serverovou částí. Stáhnutí rozvrhu probíhá přes nezabezpečené spojení. Stažený rozvrh je následně importován do databáze. Uživateli je rozvrh poskytnut prostřednictvím serverové části a komunikace je tedy zabezpečena pomocí SSL.

6.4 Uživatelská práva

V systému lze nalézt několik modulů v rámci níž lze definovat práva nezávisle na ostatních modulech. V současnosti v systému existují dva moduly s vlastními právy – globální uživatelský modul a modul skupin.

Každý uživatel má po vytvoření účtu přiřazeno tzv. právo *„Standard user“*. Uživatel s těmito právy má nejnižší úroveň oprávnění. Jedná se o tzv. *„normálního uživatele“*, který nemá právo vytvářet skupiny. Tento uživatel se může sám stát členem pouze veřejných skupin nebo být administrátorem přiřazen do privátní skupiny. Ze skupin se může samostatně odhlašovat.

Uživatel s právy *„Group master“* vlastní všechna práva jako *„Standard user“*. Tento uživatel má navíc právo vytvářet skupiny, jak veřejné tak i privátní. Dále může přiřazovat uživatele do privátních skupin a určuje práva jednotlivých uživatelů v rámci skupin.

Posledním typem uživatele je *„Administrátor“*. *„Administrátor“* má všechna práva jako *„Group master“*. Tento uživatel má úplnou kontrolu nad systémem.

Každý uživatel má v rámci každé skupiny, které je členem, definovaná vlastní práva. Existují dva základní typy – *„read“* a *„edit“*. Pokud má uživatel právo *„read“*, tak může události a úkoly v dané skupině pouze prohlížet, nemůže je měnit ani mazat. Pokud má uživatel právo *„edit“* a událost nebo úkol nevytvořil, může si ji prohlížet nebo měnit, nesmí ji však smazat. Pokud má uživatel právo *„edit“* a událost či úkol vytvořil, tak je její vlastník, pak si událost smí prohlížet, může ji měnit nebo smazat.

Pokud se uživatel sám přiřadí do veřejné skupiny, je mu automaticky přiděleno právo *„read“*.

7 Testování

V rámci testování jsem se zaměřil na dvě hlavní oblasti a tou je testování multiplatformnosti a uživatelského rozhraní.

7.1 Testování multiplatfomnosti

Testování se zaměřilo jak na rozdílné softwarové platformy v podobě OS, tak i na hardwarové. Test proběhl na třech OS – MS Windows XP SP2 (česká), MS Windows Vista (anglická) a Linux SUSE 10.3 s GNOME (česká). Primární platformou byla MS Windows Vista, na které probíhal vývoj. Hardwarová konfigurace:

a) MS Windows XP SP2 (česká):
   CPU AMD Athlon XP 2000, 256 MB RAM, 40 GB HDD (7200 ot/min), Nvidia GeForce 2 (64 VRAM), Java 1.6.0_5
b) MS Windows Vista (anglická):
   CPU Intel Pentium M (1,7 GHz), 1GB RAM, 60 GB HDD (5400 ot/min), ATI Radeon 9700 (64 MG VRAM), Java 1.6.0_5
c) Linux SUSE 10.3 s GNOME (česká):
   CPU Intel Pentium M (1,7 GHz), 1GB RAM, 60 GB HDD (5400 ot/min), ATI Radeon 9700 (64 MG VRAM), Java 1.6.0_5[1]

Protože aplikační server GlassFish i databázový server MySQL jsou multiplatformní, soustředilo se testování zejména na desktopového klienta.

Na platformě SUSE 10.3 s GNOME se napoprvé nepodařilo desktopového klienta spustit a byla nutná změna Javy na verzi 1.6.0_10b14, více v kapitole *Multiplatformnost*.

Dalším problém pak byl s implementací komponenty *javax.swing.JSpinner* opět pod Linuxem, která znemožňovala její použití. Více opět v kapitole *Multiplatformnost*. Posledním problémem pak bylo rozdílné chování implementací abstraktní třídy *java.util.Calendar* pod českou a anglickou jazykovou verzi Windows. Pod Linuxem v české verzi však bylo chování stejné jako pod anglickými Windows.

Aplikace v Jave byly vždy náročné na paměť, což se potvrdilo i zde. Doporučenou hodnotou je jistě 256 MB RAM. Tato náročnost byla dána především použitím LaF Synthetica, který pracuje velmi často s gradienty při vykreslování GUI. Závěrem však lze říci, že desktopovou aplikaci se nakonec podařilo úspěšně nejen spustit, ale i provozovat na všech testovaných platformách. Bohužel svět Linuxu je příliš fragmentovaný a neumožňuje zaručit funkčnost na všech běžných distribucích, viz nepodpora *java.awt.Desktop* API v KDE[2].

[1] Později použita Java verze 1.6.0_10b14.
[2] Následně se ukázalo, že na linuxové distribuci Debian 4.0r3 Etch s KDE 3.5.9 byla pro spuštění nutná Java verze 1.7.0b25. I přesto se však nepodařilo odstranit problém s nepodporou java.awt.Desktop API pro KDE.

7.2 Testování uživatelského rozhraní

Test uživatelského rozhraní byl proveden dvěmi samostatnými metodami – kognitivní průchod a hodnotící heuristika.

7.2.1 Kognitivní průchod

Test použitelnosti byl proveden jako kombinace techniky person a kognitivního průchodu. Pro určení a charakterizaci typického uživatele jsem použil metodu person. Tato technika mi umožní získat celkový pohled na typického uživatele. Výhodou je pak, že nám stačí relativně malý počet person pro správnou charakterizaci typických uživatelů. Dále mi persona umožní lepší „sžití“ s naším potenciálním uživatelem.

Pro otestování uživatelského rozhraní jsme vybrali techniku kognitivního průchodu. Vstupem této techniky je model případně prototyp artefaktu, sada úkolů a správný postup při řešení úkolů. Prostřednictvím simulace uživatele se snaží expert (člověk znalý dané problémové domény) nalézt nedostatky zkoumaného artefaktu. Simulovanému uživateli jsou postupně zadány jednotlivé úkoly a zkoumá se, jestli je jich schopen s pomocí artefaktu dosáhnout. Při vyhodnocování se vychází ze srovnání chování simulovaného uživatele s předpokládaným, předem daným a optimálním, průchodem úkoly. Technika kognitivního průchodu se tak dokonale doplňuje s metodou person, kdy na jejich základě získáme potřebné informace o uživateli a následně není nutná jeho přítomnost u testování. Persona pak umožní expertovi vžit se do skutečného uživatele. Dále nám kognitivní průchod umožní odhalení nejzávažnějších problémů spojených s usabilitou.

7.2.1.1 Persona

„Moje motto zní: Život je příliš krátký na to, abych ho promarnil. Snažím se toho držet a užít si jak jen to jde, kdykoliv a kdekoliv. Někdy je prostě potřeba si trochu zvýšit hladinu adrenalin (úsměv). Jo a říkejte mi Honzo, Jan je moc formální (úsměv).“

Honza vystudoval čtyřleté gymnázium v Plzni a následně začal studovat výpočetní techniku na FEL, ČVUT v Praze. „Škola je potřeba, ale prachy jsou dobrá věc, a tak jsem se rozhodl, že si najdu nějakou práci při škole.“ Už dva roky pracuje jako softwarový analytik v jedné zahraniční společnosti v Praze. Vzhledem k jeho pozici ve firmě probíhá všechna firemní komunikace v angličtině a jeho čas je organizován s přesností na minuty. „Je to tam celkem dobrý, plat není špatný a práce mě baví.“ Honza se chová v zaměstnání racionálně a odpovědně. Součástí jeho práce jsou často setkání s kolegy a analýza aktuálního stavu vývoje produktu. „A teď si představte, jak to mám všechno skloubit se školou. Ještě že existují elektronické kalendáře, jinak nevím, co bych dělal.“

Ve svém zaměstnání často pracuje s moderní technikou. „Bez mého mobilního telefonu a počítače si už dnes nedokážu představit život (úsměv).“ Honza rád cestuje, a tak si nedávno pořídil nový digitální fotoaparát. Líbí se mu na něm, že se snadno ovládá a bez velké námahy může udělat pěkné fotografie. „Fakt nesnáším, když si koupím novou věc a musím si nejdřív přečíst nějaký manuál. Já to chci jen vyndat z krabice a hned používat.“ Digitální

fotografie jsou podle něho dobrá věc, protože si je může prohlédnout kdykoliv a kdekoliv chce (fotoaparát, počítač, televize, mobil atd.).

Honza má rád moderní techniku a snaží se jít s dobou. „Nedávno jsem v jednom obchodě viděl televizi s Full HD a byla to fakt bomba. Už vím na co šetřit (úsměv).“ S penězi si Honza ale hlavu nedělá: „Když na to nemám, tak si to prostě koupím na splátky.“

Honza je svobodný, ale má přítelkyni Janu. S Janou se seznámil před dvěma roky na kolech. „Zatím se nechci nijak vázat, nejdřív si přece musím pořádně užít (úsměv) a dodělat školu.“ Společně jezdí aspoň jednou za měsíc na kole do přírody. Rád tráví dovolenou poznáváním cizích zemí. Někdy si ale zajede s kamarády pod stan. V zimě jezdí Honza na hory lyžovat. Honza také vlastní řidičský průkaz skupiny B, ale příliš často autem nejezdí.

Když se chce Honza pobavit, tak si zajde s Janou do kina či zatancovat do klubu nebo surfuje po internetu. „Rád se kouknu na nějaké video na netu nebo pokecám s kamarády přes ICQ. Net je super, najdu tam prostě všechno. Například když jsem jel s kámošema pod stan, tak jsme tam našli kemp, jaký bude počasí atd.“ Honza není pověrčivý, ale když uvidí na webu nějaký horoskop, tak se rád podívá.

Honza nosí brýle na čtení. Brýle si bere jen doma, když jde někam ven, tak si bere vždy kontaktní čočky. Honza rád nosí pěkné a moderní oblečení.

7.2.1.2 Artefakt

Artefakt je systém, který bude zkoumán. V tomto případě se jedná o použití desktopového klienta aplikace MUSC s připojením na server. Aplikace MUSC byla podrobně popsána v předchozích kapitolách této práce.

7.2.1.3 Otázky

Během kognitivního průchodu byly v každém kroku kladeny otázky. Tyto otázky slouží k odhalení slabých míst aplikace a identifikaci potenciálních problémů v uživatelském rozhraní. Jedná se o následující otázky:

a) Ví uživatel, kde se v aplikaci nachází (v jakém je stavu) a jak bude pokračovat?
b) Najde uživatel správnou funkci k řešení úkolu?
c) Pochopí uživatel, že tato funkce je ta pravá pro další postup v plnění úkolu?
d) Porozumí uživatel zpětné vazbě po provedení akce?

7.2.1.4 Testovací úlohy

Při samotném testu uživatel plní postupně sadu předpřipravených úloh. Tyto úlohy jsou vykonávány s využitím artefaktu. V každém kroku při plnění úlohy odpovídá uživatel na otázky z předchozí kapitoly. Pro kognitivní průchod byla definována sada úkolů, které se snažily postihnout všechny významné případy užití. Z důvodu omezeného prostoru zde budou prezentovány pouze některé úlohy:

a) Vytvořit opakující se událost. Událost začíná 5. 5. 2008 v 9:00 hodin a končí 5. 5. 2008 v 10:00 hodin. Událost má titulek „Schůzka“ a opakuje se tři dny za sebou.

b) V měsíčním pohledu přesunout událost s titulkem „Návštěva“ na následující den.
c) Provést import kalendáře do aplikace MUSC.
d) Nalézt telefonní číslo a jména dětí osoby se jménem „Josef Novák“.
e) Vytvořit poznámku s titulkem „Nákup“ a textem „1 mléko“. Následně změnit text poznámky na „1 mléko, 10 rohlíků“.

7.2.1.5 Vyhodnocení testovacích úloh

V této kapitole nebude probráno provádění jednotlivých průchodů, ale pokusím se naznačit správný průchod úlohou a identifikovat možné problémy v aplikaci.

Add úloha a:

Uživatel se nachází na kalendáři s týdenním pohledem. Pokud zobrazený týden neobsahuje den 5. 5. 2008, tak pomocí šipek v horní části pohledu přejde na týden, který požadovaný den obsahuje. Poté uživatel 2krát klikne do kalendáře a otevře se okno pro vytvoření nové události. Druhou možností je, že uživatel klikne na tlačítko *„New Event“* a otevře se okno pro vytvoření nové události. Pokud je třeba, tak uživatel upraví čas začátku a konce události. Do políčka *„Summary“* zapíše titulek. Následně přejde na záložku *„Recurrence“*. Na této záložce klikne na *„Enable Recurrence“*, čímž se mu stane dostupným nastavení opakování. Zde v poli *„Number of Times“* nastaví číslo tři. Nakonec uživatel kline na *„Ok“*. Událost se zobrazí v kalendáři s ikonkou opakování.

Pokud uživatel kliknul v okolí linky spojené s popisem *„9:00“*, tak se počáteční čas nastavil na požadovanou hodnotu a nemusela být již upravována. Informace jsou logicky rozděleny do záložek a uživatel ihned pochopil, kde se nastavuje opakování. Aplikace používá standardizované názvosloví a jednotné ovládání v celé aplikaci. Tato skutečnost přispěla k tomu, že uživatel neměl problém se splněním úkolů. Do budoucna by však bylo možné vytvořit ještě jednodušší variantu GUI ve stylu funkce *„Quick Add“*.

Add úloha b:

Uživatel se nachází na týdenním pohledu. Musí tedy nejprve přejít na měsíční pohled s pomocí tlačítka zobrazující měsíční pohled, kde je připravena událost s titulkem „Návštěva“. Uživatel na událost klikne a změní se kurzor ze šipky na kurzor kříže. Uživatel drží tlačítko myši a posune událost na další den.

Přesun události na další den nepůsobil uživateli žádné potíže. Jediný zádrhem nastal v okamžiku přechodu na měsíční pohled, kdy si uživatel nejprve nebyl jistý symbolem pro měsíční pohled.

Add úloha c:

Uživatel v menu *„File“* vybere položku *„Import Calendar“*. Následně se uživateli zobrazí dialog pro import. Uživatel si vybere z nabízených formátů pro import. Poté uživatel napíše umístění souboru s kalendářem nebo ho vybere pomocí dialogu. Nakonec uživatel klikne na tlačítko *„Import“* a počká, než se zobrazí dialog informující o tom, zda se podařilo import provést.

V této úloze neměli uživatelé žádné problémy a lehce ji splnili.

Add úloha d:

Uživatel přejde v stromovém menu na položku *„Contacts“*. S pomocí záložek přejde na záložku s písmenem *„N“*. Následně se zobrazí seznam osob, jejichž příjmení začíná na písmeno *„N“*. Zde uživatel uvidí v seznamu položku se jménem „Josef Novák“. U položky je vidět telefonní číslo. Následně uživatel klikne na položku se jménem „Josef Novák“. Uživateli se zobrazí vizitka osoby „Josef Novák“. Následně uživatel klikne na tlačítko *„Detail“* a otevře se okno s detailními informacemi o osobě. Na záložce *„Family“* vidí uživatel jména dětí požadované osoby.

Tato úloha proběhla ze strany uživatelů zcela bez problémů a byla oceněna možnost rychlého zjištění kontaktních informací (telefonního čísla a e-mailu). Hierarchické dělení informací o kontaktu bylo přijato kladně.

Add úloha e:

Uživatel přejde v stromovém menu na položku *„Notes“*. Pomocí tlačítka *„New Note“* se otevře dialog pro vytvoření nové poznámky. Do políčka *„Sumary“* uživatel vepíše titulek a do políčka *„Description“* text. Uživatel klikne na tlačítko *„Ok“*. Po vytvoření poznámky uživatel najede na text poznámky a kurzor myši se změní ze šipky na kurzor editace. Uživatel klikne do textu a upraví jeho obsah. Při kliknutí do textu se poznámka vybere. Nakonec uživatel klikne mimo poznámku, která již není vybrána, a přestane tak být editována.

Splnění této úlohy proběhlo úspěšně. Uživatelé velmi ocenili možnost přímé změny textu poznámky.

7.2.2 Hodnotící heuristika

Jakob Nielsen ve svém článku *Ten Usability Heuristics* **[3]** definoval sadu základních otázek, které se snaží odpovědět na otázku, zda je uživatelské rozhraní správně navrženo. Zde se využívá heuristických metod.

Visibility of system status (aplikace by měla dávat uživateli najevo, v jakém stavu se nachází) – aplikace poskytuje informaci o tom v jakém je stavu (aktuální primární funkce) pomocí malého stromového menu po levé straně aplikace. Zde je vždy zvýrazněna aktuální primární funkce (kalendář, kontakty, poznámky). Pokud existují další stavy nějaké primární funkce, např. jednotlivé pohledy u kalendáře, je tento stav zobrazen pomocí tzv. *„toggle“* tlačítka (stlačené 1 tlačítko z *n*). Bohužel dialogy byly často bez titulků.

Match between system and the real world (aplikace by měla používat pojmy z reálného světa ve stejném významu) – v aplikaci jsou použity termíny z běžného světa. Uživatel potřebuje pouze základní znalost anglického jazyka. Aplikace neobsahuje žádné speciální nebo nesrozumitelné výrazy. Aplikace používá standardní názvosloví používané v obdobných aplikacích. Trochu matoucí bylo označení jednotlivých pohledů, což bylo následně upraveno.

User control and freedom (vždy by měla existovat cesta zpět) – uživatel se může v aplikaci volně pohybovat. V každém dialogu a při každé akci je přítomno tlačítko *„Cancel“*, které

umožní danou akci ukončit bez jakýchkoliv následků. Většina akcí je vratná, výjimkou je pak např. nemožnost změny neopakující se události na opakující se.

Consistency and standards (jednotné rozhraní v aplikaci a dodržení obecných standardů) – v rámci celé aplikace existuje jednotné rozhraní. Jednotné rozhraní je dodrženo i napříč všemi platformami. Pro dodržení konzistence s ostatními aplikacemi podobného charakteru, je zvolen pro import a export kalendáře standard iCalendar a pro kontakty standard vCard. Tato skutečnost pomáhá k dodržování jednotné terminologie nejen v rámci aplikace, ale i s ostatními aplikacemi.

Recognition rather than recall (uživatel by měl vždy mít v dosahu potřebné ovládací prvky) – všechny ovládací prvky jsou vždy dostupné a viditelné. Ovládací prvky jsou logicky roztříděny. Navíc Informace i ovládací prvky jsou hierarchicky roztříděny, což přispívá k lepší orientaci.

Flexibility and efficiency of use (efektivita používání) – uživatel je schopen velmi rychle a snadno dosáhnout svých cílu. Příkladem budiž použití DnD nebo rychlé vytvoření události pomocí funkce *„Quick Add“*.

Error prevention (předcházení chyb ze strany uživatelů) – v aplikaci existují místa, kde by bylo vhodné upozornit na následky způsobené danou akcí.

Aesthetic and minimalist design (dialogy by měly obsahovat pouze podstatné a srozumitelně uvedené údaje) – dialogy vždy obsahují pouze podstatné informace. Tyto informace jsou v některých případech navíc rozděleny pomocí záložkových panelů. Popisky jednotlivých ovládacích prvků jsou jasné a srozumitelné. Jediný problém byl s označení jednotlivých pohledů v kalendáři.

Help users recognize, diagnose, and recover from errors (srozumitelnost chybových hlášení) – existují dva typy chyb. Prvním jsou chyby, které znemožňují práci, jako např. špatné heslo či nenavázání spojení se serverem. Chyby tohoto charakteru jsou uživateli zobrazeny pomocí informačního dialogu, který musí být potvrzen. Druhým typem chyb je např. nedostupnost RSS zdroje. Tyto chyby jsou před uživatelem skryty.

Help and documentation (nápověda a dokumentace) – aplikace v současné době neosahuje žádnou nápovědu.

8 Závěr

V rámci diplomové práce vznikl víceuživatelský systém pro správu času, kontaktů a poznámek. Aplikace poskytuje snadný a rychlý přístup k informacím prostřednictvím RSS čtečky či vyhledávání na internetu. Uživatelé mají možnost jednoduchého sdílení časových plánů v rámci uživatelsky definovaných skupin. Uživatelé, kteří náleží do akademické obce FEL, ČVUT v Praze, mají navíc možnost pracovat se školním rozvrhem z KOSu a zahrnout ho, tak do svých plánů. Aplikace podporuje export a import kalendáře do formátu iCalendar a kontaktů do formátu vCard.

Architektura systému je založena na klasické třívrstvé architektuře – klient, server a databáze. Na základě otevřené a vrstevnaté architektury je možné systém dále snadno rozvíjet či měnit. V budoucnu je tak možné vytvořit celou řadu klientů. O kvalitě architektury zcela jistě vypovídá fakt, že systém lze pouhou konfigurací přeměnit na čistě desktopovou aplikaci s embedded databází. Vysokého zabezpečení celého systému je dosaženo s pomocí asymetrických šifer, hashovacích funkcí a systému práv pro uživatele a pro členství ve skupinách. Ačkoliv jazyk Java je multiplatformní, tak jeho API pro tvorbu GUI obsahuje celou řadu chyb a nedodělků, což znemožňuje úplnou multiplatformnost klienta. I přes tuto skutečnost se však podařilo zajistit podporu řady platforem.

Design klientské aplikace byl navržen s ohledem na usabilitu. Díky tomu je práce s klientem rychlá, příjemná a zcela přirozená. Uživatelské rozhraní klientské aplikace bylo testováno technikou kognitivního průchodu s využitím metody person. Následně byla ještě provedena hodnotící heuristika dle Jakoba Nielsena. Ani jedna ze zmíněných technik neodhalila žádné závažné nedostatky, které by bránily ve snadném používání aplikace.

Při porovnání s konkurencí aplikace nabízí služby na srovnatelné úrovni. Naopak konkurenci zcela jistě překonává jak ve formě prezentace informací, tak i ve způsobu práce se samotnou aplikací, která je zcela přirozená. S ohledem na architekturu poskytuje systém mnohem větší možnosti na další rozvoj oproti konkurenci. Na rozdíl od konkurence je zde snazší možnost sdílení plánů mezi více uživateli a je zde přítomna i podpora školního rozvrhu z KOSu.

Projekt MUSC byl Open Sourcován na SourceForge.net (http://sourceforge.net/projects/musc) pod BSD licencí. Systém by bylo jistě vhodné do budoucna obohatit o řadu nových klientů, např. webový či pro mobilní telefony. Další možností je pak rozšíření funkcionality o úkoly, přímé posílání e-mailů, informace o počasí či vyhledávání a přehrávání videí z YouTube. S ohledem na interoperabilitu se nabízí možnost podpory Google Calendar, MS Exchange Serveru či dalších formátů pro export a import.

9 Seznam použité literatury

[1] Fermilab Computing Division. *PostgreSQL or MySQL?* [online]. Poslední revize 2005-2-15 [cit. 2007-11-13]. <http://www-css.fnal.gov/dsg/external/freeware/pgsql-vs-mysql.html>.

[2] Macich, J. *Webové kalendáře velké trojky* [online]. Datum publikování 2007-9-14 [cit. 2007-10-5]. <http://www.lupa.cz/clanky/webove-kalendare-velke-trojky/>.

[3] Nielsen, J. *Ten Usability Heuristics* [online]. Datum publikování 1994 [cit. 2008-2-10]. <http://www.useit.com/papers/heuristic/heuristic_list.html>.

[4] Hall, M. *Java servlety a stránky JSP*. Neocortex spol. s r.o., Praha, 2001.

[5] Spell, B. *Java Programujeme profesionálně*. Computer Press, a.s., Praha, 2002.

[6] Herout, P. *Java – grafické uživatelské prostředí a čeština*. Kopp, České Budějovice, 2001.

[7] Herout, P. *Java – bohatství knihoven*. Kopp, České Budějovice, 2003.

[8] Herout, P. *Java a XML*. Kopp, České Budějovice, 2007.

[9] Dawson, F. *vCard MIME Directory Profile, RFC 2426* [online]. Datum publikování 1998-9 [cit. 2007-11-1]. <http://tools.ietf.org/html/rfc2426>.

[10] Dawson, F. *Internet Calendaring and Scheduling Core Object Specification (iCalendar), RFC 2445* [online]. Datum publikování 1998-11 [cit. 2007-11-10]. <http://tools.ietf.org/html/rfc2445>.

[11] Sun Microsystems, Inc. *Java SE Tutorials* [online]. Poslední revize 2008-3-14. <http://java.sun.com/docs/books/tutorial/index.html>.

[12] Jendrock, E. – Ball, J. – Carson, D. – Evans, I. – Fordin, S. – Haase, K. *The Java EE 5 Tutorial* [online]. Datum publikování 2007-9 [cit. 2007-12-3]. <http://java.sun.com/javaee/5/docs/tutorial/doc>.

[13] MySQL AB. *MySQL 5.1 Reference Manual* [online]. Datum publikování 2006-09-07. <http://dev.mysql.com/doc/refman/5.1/en/index.html>.

[14] JBoss Inc., *HIBERNATE – Relational Persistence for Idiomatic Java, Hibernate Reference Documentation 3.2.2* [online]. <http://www.hibernate.org/hib_docs/v3/reference/en/html>.

[15] Cvrček, P. *Začínáme s MySQL* [online]. Datum publikování 2001-8-24 [cit. 2007-11-13]. <http://www.zive.cz/default.aspx?section=21&server=1&article=102589>.

[16] Sun Microsystems, Inc. *Java™ Platform, Standard Edition 6 API Specification* [online]. Datum publikování 2006. <http://java.sun.com/javase/6/docs/api>.

A Seznam použitých zkratek

A

Ajax – Asynchronous JavaScript and XML

AS – aplikační server

AWT – Abstract Window Toolkit

C

CSV – Comma-separated values

D

DnD – Drag-and-drop

E

EJB – Enterprise Java Beans

G

Gtk+ – Gnu ToolKit

J

JEE – Java Platform, Enterprise Edition

JPA – Java Persistence API

JSR – Java Specification Request

JTA – Java Transaction API

K

KDE – K Desktop Environment

L

LaF – Look And Feel

LDAP – Lightweight Directory Access Protocol

N

MSN – The Microsoft Network

M

MUSC – Multiuser Scheduling Calendar

O

ORM – Object-relational mapping

OS – operační systém

R

RMI – Remote Method Invocation

RSS – Really Simple Syndication

S

SAX – Simple API for XML

SSL – Secure Sockets Layer

U

URI – Uniform Resource Identifier

URL – Uniform Resource Locator

UTC – Coordinated Universal Time

V

VM – virtual machine

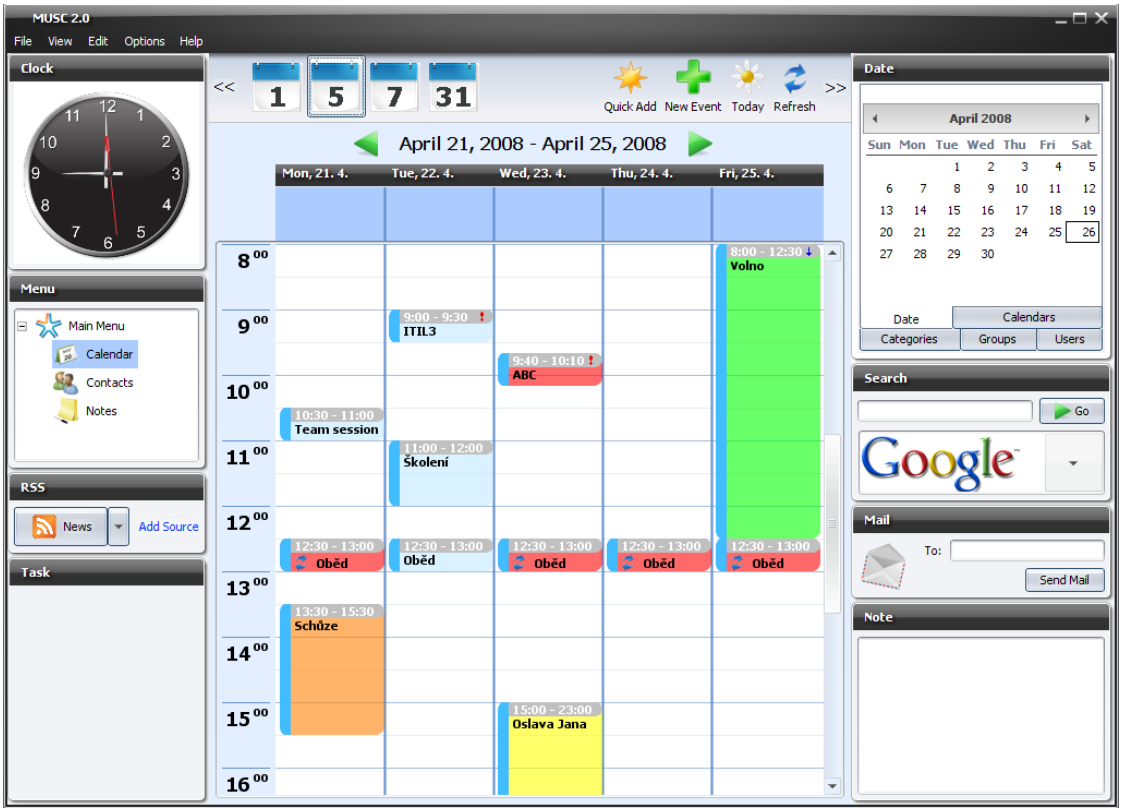
X

XCB – X C Binding

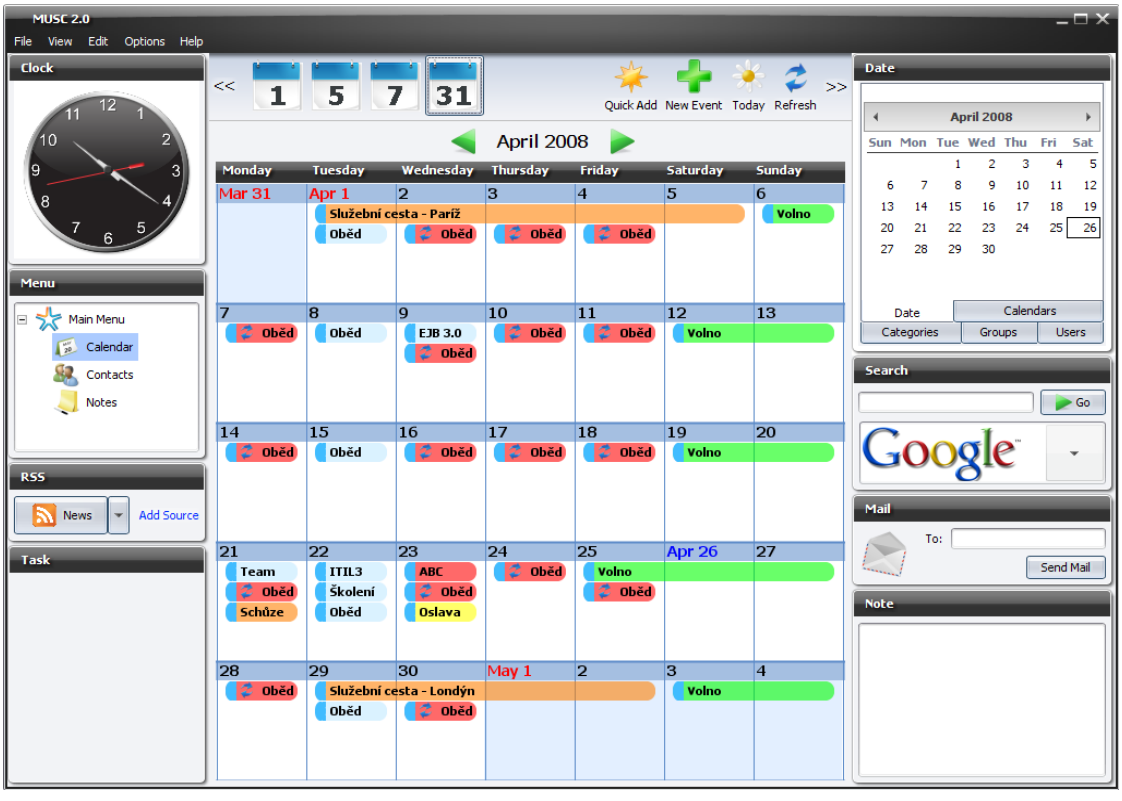
B Obsah přiloženého DVD

- *data*\ – adresář obsahující podpůrné programy pro instalaci systému
  - *glassfish-installer-v2ur2-b04-windows.jar* – instalátor aplikačního serveru
  - *jdk-6u6-windows-i586-p.exe* – instalátor JDK
  - *jre-6u6-windows-i586-p.exe* – instalátor JRE
  - *mysql-essential-5.0.51b-win32.msi* – instalátor databázového serveru
- *doc*\ – adresář obsahující dokumentaci
  - j*avadoc*\ – adresář obsahující *javadoc*
  - *UserGuide_MUSC_Outulny.pdf* – uživatelská příručka
- *exe*\ – adresář obsahující instalační data
  - *MUSC-2-Client*\ – adresář obsahující instalační data klientské části aplikace
  - *MUSC-2-Server*\ – adresář obsahující instalační data serverové části aplikace
  - *UserGuide_MUSC_Outulny.pdf* – uživatelská příručka
- *src*\ – adresář obsahující zdrojové kódy aplikace
  - *MUSC-2-Client*\ – adresář obsahující projekt klientské části aplikace pro NetBeans 6.0
  - *MUSC-2-Server*\ – adresář obsahující projekt serverové části aplikace pro NetBeans 6.0
- *text*\ – adresář obsahující text diplomové práce
  - *AbstractEN.docx* – abstrakt v anglickém jazyce ve formátu Docx
  - *AbstractEN.pdf* – abstrakt v anglickém jazyce ve formátu PDF
  - *AbstraktCZ.docx* – abstrakt v českém jazyce ve formátu Docx
  - *AbstraktCZ.pdf* – abstrakt v českém jazyce ve formátu PDF
  - *DP_MUSC_Outulny_text.docx* – text diplomové práce ve formátu Docx
  - *DP_MUSC_Outulny_text.pdf* – text diplomové práce ve formátu PDF
  - *UserGuide_MUSC_Outulny.docx* – uživatelská příručka ve formátu Docx
  - *UserGuide_MUSC_Outulny.pdf* – uživatelská příručka ve formátu PDF
- *AbstractEN.pdf* – abstrakt v anglickém jazyce
- *AbstraktCZ.pdf* – abstrakt v českém jazyce
- README.txt – soubor *readme*

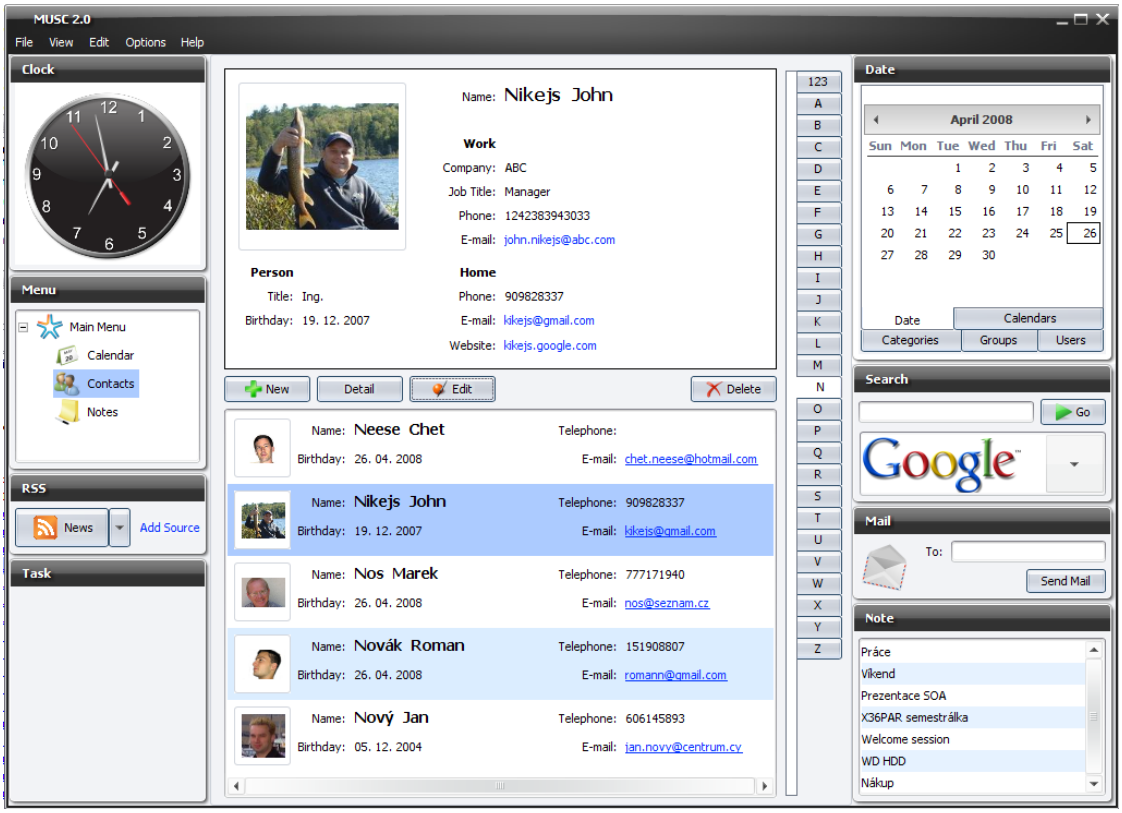
C Obrázky aplikace



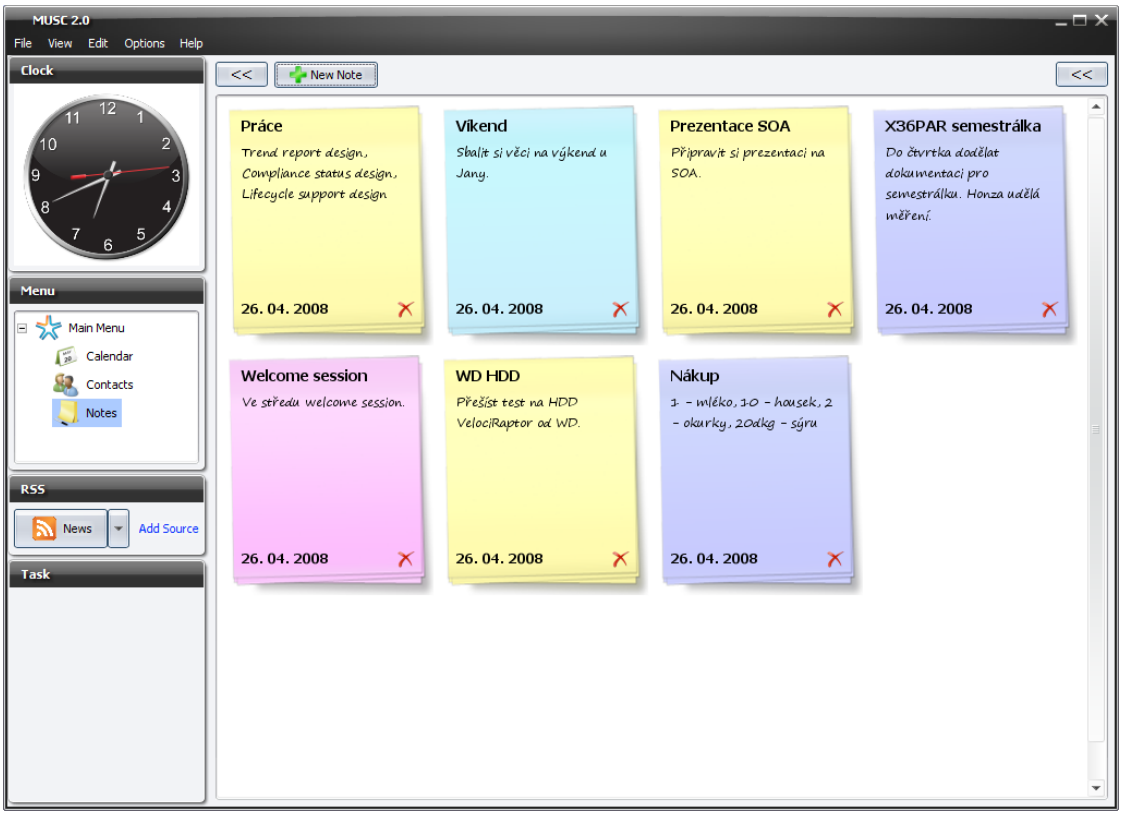
Obrázek 19: Aplikace MUSC – Kalendář, pohled pracovní týden



Obrázek 20: Aplikace MUSC – Kalendář, měsíční pohled



Obrázek 21: Aplikace MUSC – Kontakty



Obrázek 22: Aplikace MUSC – Poznámky