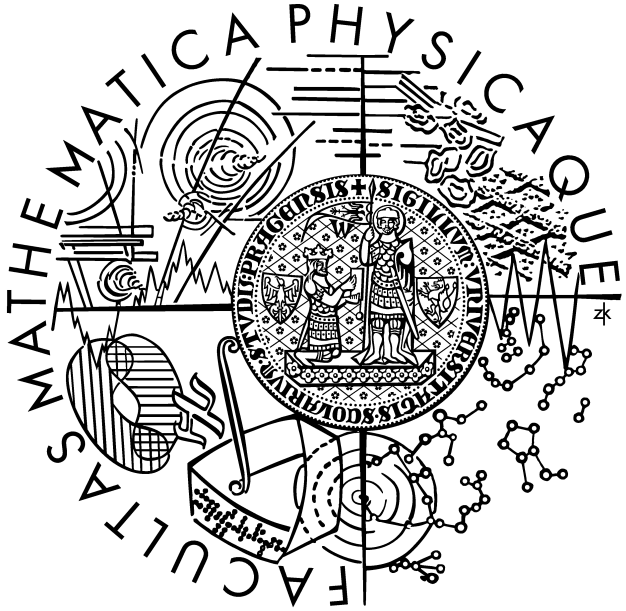
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE



Nguyen Son Tung

E-mailový klient pro J2ME zařízení

Ústav formální a aplikované lingvistiky

Vedoucí bakalářské práce: Mgr. Pavel Machek
Studijní program: Informatika, programování

2007

Rád bych na tomto místě poděkoval vedoucímu této bakalářské práce Mgr. Pavlovi Machkovi za ochotný přístup po celou dobu trvání tohoto projektu.

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

V Praze dne 29. 7. 2007 Nguyen Son Tung

Obsah

Název práce: E-mailový klient pro J2ME zařízení
Autor: Son Tung Nguyen
Katedra: Ústav formální a aplikované lingvistiky
Vedoucí bakalářské práce: Mgr. Pavel Machek
Email vedoucího: Pavel.Machek@mff.cuni.cz

Abstrakt: Cílem projektu je vytvořit emailového klienta umožňujícího práci s emaily s ohledem na rychlost a optimalizaci na malých J2ME zařízení. Program bude schopen načítat emaily z POP3 i IMAP serverů a posilat email pomocí SMTP protokolu. Klient bude podporovat MIME přílohy ke zprávám vybraných základnich formátů příloh - txt, jpg, png a jiných multimediálních formátů v závislosti na konkrétnim mobilním telefonu. Zprávy budou organizovány ve standardních složkách (Inbox, Outbox, Rozepsané, Koš...). Zprávy mohou být řazeny ve složkách podle uživatelem zvolených kritérií (odesílatel, subject, datum, velikost). Podpora automatické kontroly nových zpráv. Operace, u kterých se předpokládá dlouhá doba zpracování jsou přerušitelné a jejich stav je zobrazitelný. Inteligentni T9-like adresář.

Klíčová slova: Emailový klient, J2ME program, sofware pro mobilní zařízení

Title: E-mail client for J2ME devices
Author: Son Tung Nguyen
Department: Institute of Formal and Applied Linguistics
Supervisor: Mgr. Pavel Machek
Supervisor's e-mail address: Pavel.Machek@mff.cuni.cz
Abstract: The aim of the project is to create a optimized client for using on mobile J2ME devices capable of work with emails. Both POP3 and iMAP servers will be supported to retrieve mails. SMTP protocol will be used for sending emails. The client will also support MIME type attachments and display basic supported formats like: txt, jpg, png and other multimedia formats that are supported by user's mobile device. Standard mail boxes like Inbox, Outbox, Drafts, Trash... will be supported as well. Emails can be ordered by various user’s criterias : by subjects, date, size and senders or recipients. The program will support many email accounts. Automatic new emails checking. Operations that take longer time can be halted and whose progress will be displayed. Inteligent T9-like addressbook.

Keywords: Email client, J2ME program, software for mobile devices

Kapitola 1

Úvod

S rozšířením mobilních zařízeních souvisí i rozšíření a dostupnosti softwaru pro tato zařízení. Též používání emailu se stalo běžným způsobem komunikace. Avšak na trhu je stále malý počet softwaru umožňujícího práci s emailem pro tato mobilní zařízení. Zvláště volného funkčního či dokonce uživatelsky atraktivního softwaru. Některé drahé a novější mobilní telefony sice mají zabudovaný emailový klient, ale ty nejsou open-source aproto obsahují-li chyby či chybí-li jim nějaká funkce, uživatel nemá možnost to nějak změnit či ovlivnit.

Nejběžnější softwarovou platformou pro mobilní zařízení je platforma J2ME (či s novým názvem Java Micro Edition) od firmy Sun Microsystems.

Proto při hledání tématu bakalářké práce mě velmi zaujal projekt vypsaný Mgr. Pavlem Machkem – Emailový klient pro zařízení podporující platformu J2ME. Protože když jsem dostal před 2 roky svůj první mobil podporující J2ME a chtěl jsem si do něho nainstalovat emailový klient, tak jsem nemohl najít žádný funkční, stabilní a volně rozšiřitelný program a tak se mi zdálo téma užitečné i mně samotnému.

Tento projekt – MujMail se vyvíjel relativně dlouho. Projekt byl uveden v roce 2003 jako open-source vyvíjený Petrem Špatkou a po nějaké době se do něho zapojil i můj vedoucí projektu – Pavel Machek. Program byl navržen jako velmi jednoduchý klient umožňující práci s emailem přes POP3 a SMTP protokoly. V roce 2005 se do něho zapojil i Martin Štefan, který se mnou pracoval v týmu. V roce 2006 jsem se rozhodl s Martinem přepsat MujMail a vytvořit jiný návrh, aby mohl splnit požadavky zadaní bakalářské práce, jelikož původní program byl navržen s jiným cílem a požadavky. Jelikož na „novým” MujMailu se mnou pracoval Martin Štefan a bylo využito velmi malé procento původního projektu z roku 2003, aproto v tomto textu se budu věnovat detailně pouze částem, na kterých jsem pracoval či se podílel na vývoji. U každého modulu, kerý není čistě moje práce, explicitně upozorním a popíšu pro představu jen velmi stručně.

Kapitola 2

Elektronická pošta

Je služba, která může být realizována různými způsoby a na různých platformách (MS Mail, Mail602…). V dnešním Internetu se používá tzv. **SMTP-pošta**, která je založena na bázi protokolu SMTP a RFC 822. Aproto je tento formát elektronické pošty je otevřený a nikdo ho nevlastní. Dnešní email má určité výhody oproti jiným způsobům komunikace: je levný, rychlý a „umí toho hodně“ – lze do něho připojit přílohy různých formátů jiných než prostý text.

2.1 Filozofie a architektura emailu

Samotný formát emailu byl definován již velmi dlouho (RFC 822 je z roku 1982). Prvotním cílem bylo posílat pouze jednoduché zprávy ve **formě textu**, obsahující pouze 7bitové ASCII znaky. S postupným vývojem byly na něj kladeny nové požadavky jako je podpora národních abeced a možnost posílání netextových, multimediálních souborů. Tyto požadavky byly vyřešeny postupným rozšiřováním formátu o jiné standardy.

Architektu emailu je založena na **modelu Klient/Server**.

Poštovní Server je entita zajišťující přenos a uchovávání emailových zpráv. Kdežto **Mailový klient** je entita zprostředovávající uživatelské rozhraní a umožňuje uživateli s emailovou zprávou pracovat a zpracovávat jednotlivá zprávy.

Server a klienti bývají od sebe odděleny, tzn. že leží na různých místech, aproto bylo potřeba vyvinout jednotné prostředky pro komunikaci mezi komunikujícími stranami – tzv. **protokoly**.

Struktura emailu

Každá zpráva je rozdělena na tzv. **hlavičku** a **tělo.**

Hlavička je tvořena jednotlivými ***položkami*** a tyto položky můžou ležet na jedné řádce či rozloženy na více řádků (v tomto případě každá další řádka položky začína mezerou či tabulátorem). Každá položka je uvozena klíčovým slovem zakončeným dvojtečkou, za níž následuje vlastní obsah položky. Syntaxe obsahu některých položek je přesně definován (třeba u položky pro datum, tvar emailové adresy) pro správné jednotné strojové zpracování, avšak jiné položky určené pro uživateli či vnitřní použití daného serveru mužou mít libovolnou syntaxi(přemět zprávy, spam-rating..). Pořadí položek v hlavičce též není pevně dán a všechny nejsou povinné.

Tělo zprávy, které je odděleno od hlavičky prázdným řádkem obsahuje vlastní sdělení zprávy. Obsah těla podle původního formátu definovaného v RFC 822 je tvořen pouze jedním textem zprávy, avšak s rozšířením formátu o standard ***MIME*** je tělo s vícemi částmi (přílohami) podporováno.

Příklad emailové zprávy:

Hlavička:

To: tomas.dvorak@seznam.cz
From: "Martin" <martin.dvorak@seznam.cz>
Date: Mon, 12 Dec 2007 01:20:00 +0100
Subject: Dulezity

Tělo:

<prázdný řádek>
Nazdrar!
Kdy mi vratis ty penize?

Rozšíření formátu

Jelikož původní formát dle RFC 822 zavádí 7-bitové omezení na přenášené znaky, tak přenos 8-bitových znaků a binární dat (národní znaky, formátovací znaky a přílohy) bylo nemožné a tak lidi museli najít nějaké řešení umožňující přenos 8-bitových znaků. Principilně řešení spočívá v převedení 8-bitových na 7-bitových a potom po přenosu se zase všechno vrátí do původní podoby. Pro tyto účely byl zaveden standard **Multipurpose Internet Multimedia Extensions** (**MIME**).

MIME

V MIME je definováno jak se má:

1) ***Převádět*** 8-bitová data na 7-bitová - pomocí kódování Quoted Printable a Base64

2) ***Typovat data*** - zavádí se tzv. MIME type. Řetězec, ze kterého je možné odvodit typ přenášených dat. Př.: image/jpg

3) ***Rozšiřovat formát zprávy*** - zavádí se nové položky v hlavičce mailu (content-type…)
   - definování struktury těla zprávy pro podporu příloh, viz. níže

V rámci rozšíření formátu je tělo multi-složkové zprávy (zpráva s přílohami) rozděleno na jednotlivé části tzv. **bodyparts** (každá bodypart odpovídá jednotlivé příloze). Bodyparts jsou ve textové ASCII formě**,** avšak jejich skutečná interpretace a formát nemusí být pouze text. Tyto bodyparts vždy začínají textovým řetězcem, tzv. **boundary**, které je definováno v položce content-type. Jednotlivé bodyparts jsou od sebe odděleny od sebe prázdnou řádkou.

2.2 Mechanismus přenosu zpáv

Doručování zpráv

Dnešní emailová adresa se zkládá ze dvou částí:

Alias – část, která se nachází před znakem zavináč

Domény – část která se nachází po znaku zavináč

Alias určuje, kterému uživateli se má zpráva doručit. Přesněji do které poštovní schránky se má zpráva doručit. Rozhodovácí mechanismus pro výběr schránnky, do které se má zpráva doručit, řídí SMTP server příjemce. A tento SMTP server je vždy jednoznačně určen podle MX záznamů DNS domény.

SMTP protokol

Pro doručování zpráv se používá, jak již bylo řečeno výše, mechanismus definovaný protokolem RFC 821 – Simple Mail Transfer Protocol. Tento protokol je použit jak mezi klientem a SMTP serverem odesílatele, tak i mezi SMTP severy ležící na cestě mezi SMTP serverem odesílatele a SMTP serverem adresáta.

Obvykle komunikující strany, není-li jinak domluveno (třeba pro zabezpečení přípojení), navážou spojení přes port 25. Poté dochází ke vzájemnému dialogu, ve kterém si strany předávají identifikační údaje, údaje důležitá pro přenos a pak je přenesena vlastní zpráva.

Jak SMTP příkazy, tak i přenášená data mají textový charakter. Každý text se skládá ze 7-bitových ASCII znaků a tento text je členěn na řádky pomocí CR+LF.

Stahování a manipulace se zprávami

Aby si mohl klient stáhnout a manipulovat se zprávy na serveru, musí tu existovat nějaký mechanismu určující, jak to celé bude probíhat. V dnešním světě se pro tyto účely nejběžněji používají ***protokoly***:

- **Post Office Protocol verze 3 (POP3)**
- **Internet Message Access verze 4 revize 1 (IMAP4)**

Též tu exitují 3 základní přístupy(***modely***) pro manipulaci se zprávami, podle níž se klient a server chovají:

- **Offline**
- **Online**
- **Disconnected**

Modely

Offline

U tohoto přístupu se klient chová tak, že si po připojení k serveru klient stáhne nové zprávy a odpojí se od serveru. Manipulace se zprávami probíhá pouze lokálně na počítači, kde leží klient.

Online

Zde klient manipuluje přímo se zprávami (či se schránkami) ležícími na serveru. Proto musí být klient během práce stále připojen k serveru a neukládá si žádná data lokálně. Tak se chová např. webový klient.

Disconnected

Tento přístup je na rozhraní dvou výše uvedených. Nejprve si stáhne nové zprávy a manipuluje si s nima lokálně [offline přístup]. Poté promítne tyto změny se zprávami ležícími na serveru [online přístup]. Synchronizace zpráv mezi klientem a serverem probíhá díky existenci speciálního identifikátoru každé zpávy.

Naše aplikace se může chovat podle Offline nebo Disconnected modelu v závislosti na nastavení.

Protokoly

U obou protokolů POP3 a IMAP4 mají data a příkazy textový charakter. Text se skládá ze 7-bitových znaků a je členěn na řádky zakončenými CR+LF.

POP3

Ze dvou protokolů je POP3 považován za ten jednodušší. Je definován RFC 1939. Komunikující strany se domlouvají a identifikují standartně přes port 110. Protokol POP3 nabízí spíše offline přístup, kdy si klient stáhne zprávy ze serveru a zároveň je smaže ze serveru. Avšak klient má i možnost stáhnout si jenom hlavičky zpráv a ponechát si zprávy na serveru a mít k nim přístup i později. Proto tyto účely pozdějšího přístupu je každá zpráva opatřena unikátním identifikátorem.

IMAP4

IMAP4, definovaný RFC 3501, vznik později než POP3 a je mnohem propracovanější a složitější. Komunikující strany se domlouvají a identifikují standartně přes port 143. Klient a server se můžou přes IMAP4 chovat podle všech tří modelů, ikdyž se většinou předpokládá online model. Nabízí nejenom základní práce se zprávami jako je stahování a mazání, ale i stahování hlavičky (toto nemusí podporovat všechny POP3 servery), struktury těla, přístup k jednotlivým bodypart. Dále nabízí i práci se příznaky zpráv – označit zprávu za přečtenou, smazanou..atd, též je tu možnost vyhledávat zprávy podle nejrůznějších kritérií, či dokonce manipulace se samotnými schránkami. Nabízí i možnost zabepečeného připojení.

Kapitola 3

Technologie

Technologie použítá v tomto projektu je speciální platforma Java technology od firmy Sun Microsystems nazvaná **Java 2 Micro Edition** (**J2ME**), nověji má název Java Micro Edition.

J2ME

Přesněji je J2ME souhrn specifikací a technologií tvořící podmnožinu standardní Java platformy (která se skládá z jazyka Java, compileru jazyka, knihoven, virtuální stroj…). Tedy J2ME definuje, mimo jiné, kolekci Java API, knihoven, run-time prostředí pro vývoj běh softwaru na zařízeních s hardwarovými omezeními.

Avšak zažízení podporující J2ME se od sebe velmi liší hardwarově, třeba velikostí RAM, výkonem CPU, úložným prostorem..(vezme si například pager a satelitní přijímač), proto se J2ME rozděluje dále na ***konfigurace.*** Konfigurace určuje základní sadu knihoven, avšak nic neříká o uživatelským interface,vstupu a výstupu…, protože zařízení v jedné konfiguraci se značně liší třeba velikostí grafického výstupu, uživatelským vstupem, způsobem práce s ukládáním dat a přípojení k síti. Aproto konfigurace jsou ještě rozděleny na ***profily***, které tyto vlastnosti přesněji specifikují.

Konfigurace

Jelikož zařízení jsou hardwarově velmi omezené a odlišné, došlo ve J2ME ke značné redukci a změnám v knihoven oproti standardní edici. Též musel být navržen jiný obecnější model připojení, jiný způsob ukládaní dat a jiný způsob verifikace kódu. Redukcím se ani nevyhnul virtuální stroj.

J2ME se dozděluje na 2 konfigurace:

1) **Connected Limited Device Configuration (CLDC)**
2) **Connected Device Configuration (CDC)**

CLDC

Je konfigurace určená pro zařízení s velmi omezenými možnostmi hardwaru. CLDC specifikuje minimální nároky na uložný prostor – 160kB a paměť RAM 32kB.

CDC

Je určena pro výkonější zařízení oproti CLDC. Zařízení musejí mít RAM o velikosti aspoň 2MB RAM a 32-bitový CPU. A některé i implementují kompletní virtuální stroj jako Java jazyk.

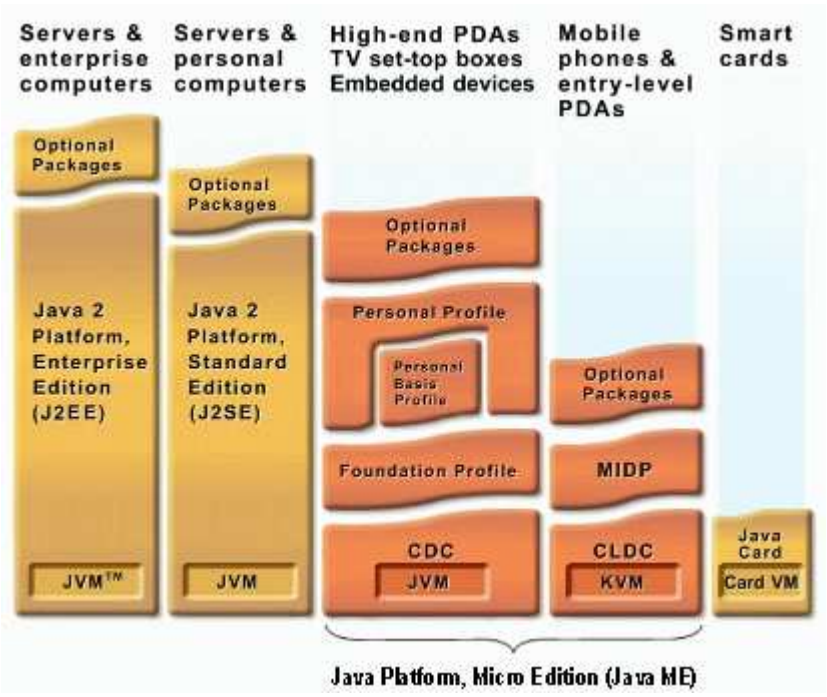
Profily

Dnes existují 2 základní profily pro každou konfiguraci ***Foundation a Mobile Information Device Profile (MIDP)***. Profily mohou být doplněny dalšími knihovnami a přesněji specifikovány ještě menšími profily.

MIDP

MIDP je profil určený pro použití na nejmenších zařízeních, jako jsou mobilní telefony. K hardwarové specifikaci přidává požadavek na minimální velikost displeje 96 x 54 pixelů a na možnost ovládat zařízení klávesami nebo dotykovým displayem. Také navíc vyžaduje aspoň 8 kB perzistentní paměti pro ukládání dat aplikací. Aplikacím určeným pro zařízení podporující MIDP se říká **midlet**. Právě tato platforma se těší největší pozornosti a naše aplikace bude implementována v tomto profilu.

Následující obrázek ze serveru[1] Sun.com ilustruje rozdělení Java platformy:



MIDlet

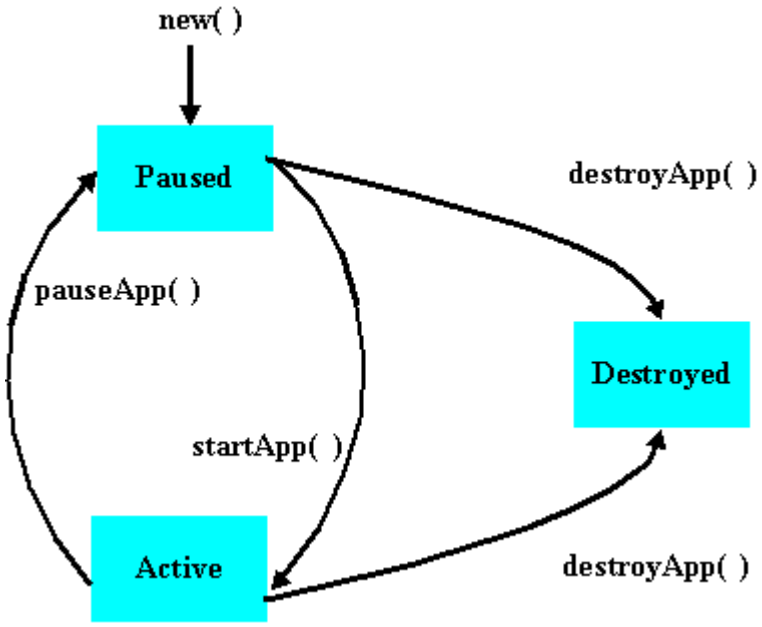
MIDlet je aplikace určená pro MIDP profil. Všechny class soubory této aplikace jsou zabaleny do jednoho JAR archívu. Tento archív, který může být doprovázen manifest souborem obsahující základní informace o aplikaci (velikost aplikace, autor, verze..), může být nainstalován do zařízení kabelem či bezdrátovou sítí (tzv. Over The Air způsob).

O běh MIDletu a její přechod mezi stavy, ve kterých se může MIDlet nacházet, se stará ***Aplikační manažer***. MIDlet musí obsahovat základní třídu, rozšiřující abstraktní třídu javax.microedition.midlet.MIDlet. Při spouštění aplikace je tato třída vytvořena zavoláním jejího veřejného konstruktoru bez parametrů.

Stavy MIDletu

MIDlet se některými vlastnostmi podobá apletu a to tím, že se aplikace během svého života se může nacházet v různých stavech. Nejdříve se aplikace po zavolání konstruktoru nachází v *pasivním stavu*. V tomto stavu by neměla vlastnit či používat žádné sdílené zdroje. Pro přechod z pasivního stavu do *aktivního stavu* je zavolána metoda startApp(). V této metodě aplikace inicializuje své potřebné zdroje. A při volání metody pauseApp() se aplikace vrací do pasivního stavu a měla by tyto zdroje opět uvolnit. Při ukončení aplikace zavolá aplikační manažer metodu destroyApp(boolean unconditional), kterou přejde aplikace do *stavu zrušeného*.

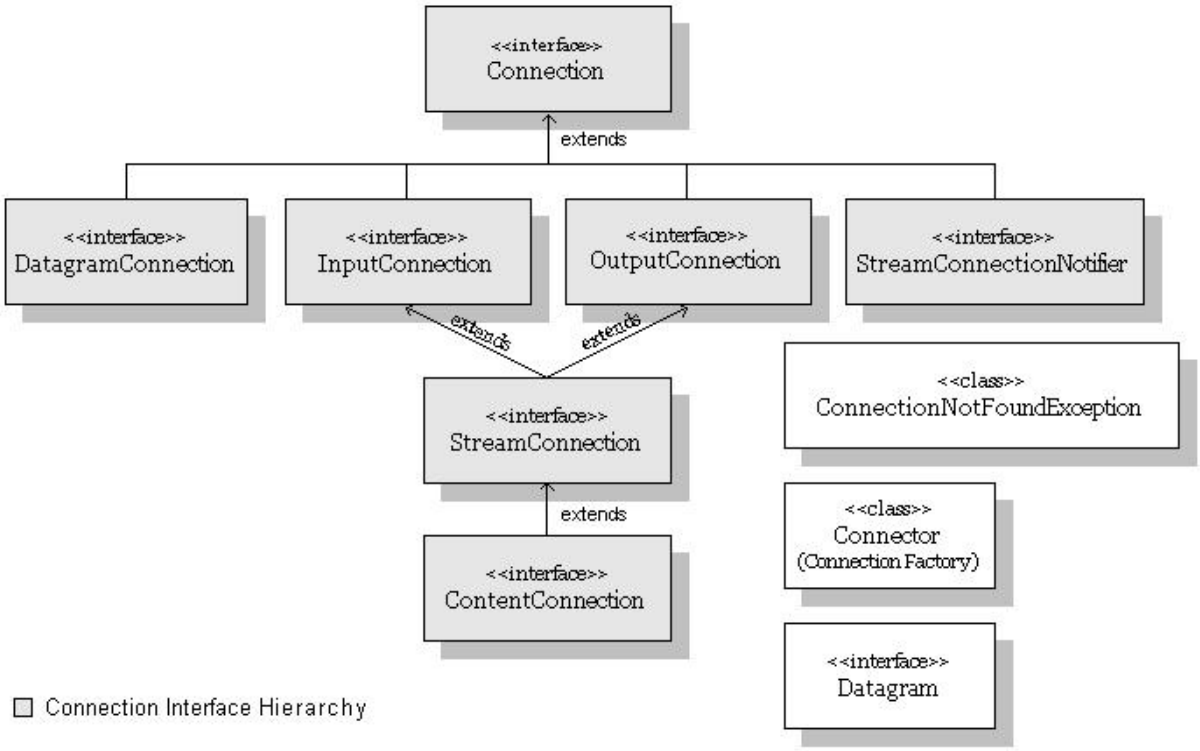
Stavy ilustruje tento obrázek[2] ze serveru Sun.com:



Způsob připojení

J2ME se liší od standardní edice J2SE způsobem připojení. Byla pro ní navržena speciální architektura, tzv. ***Generic Connection Framework (GCF)***. GCF zobecňuje a zjednodušuje nejen způsob připojení, ale i I/O systém díky základním společným rozhraním pro všechny tyto druhy připojení. Proto po další rozšíření GCF je J2ME schopné podporovat od sebe různé druhy připojení jako je: HTTP, Sockets, Datagrams, Serial Ports, SMS, Bluetooth, Files…

Následující obrázek[3] ze serveru Sun.com ilustruje základní hierarchii GCF:



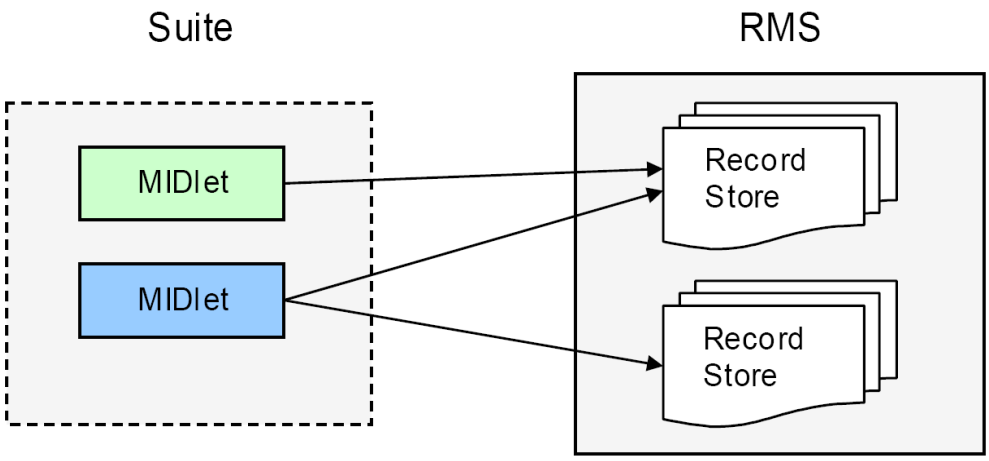
Způsob ukládání dat aplikací

Jelikož zařízení se od sebe velmi liší a není zaručeno, že budou podporovat nějaký souborový systém, byl proto navržen jednoduchý způsob ukládání dat do tzv. ***záznamů***. Tato architektura se technicky jmenuje ***Record Management System*** (RMS).

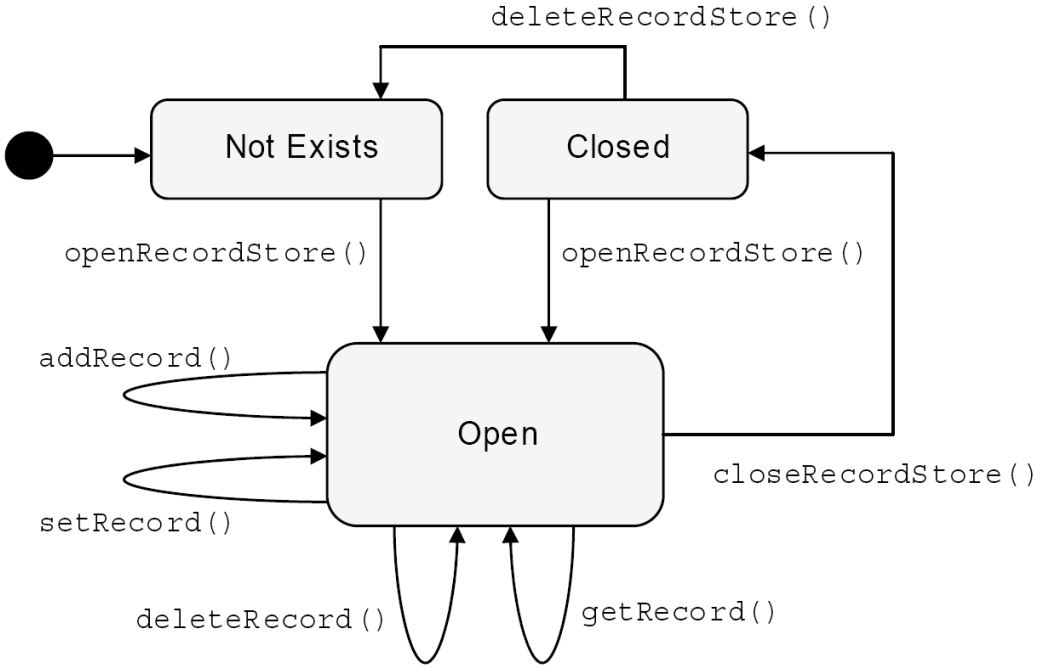
Úložiště záznamů neboli ***databáze*** se skládá ze seznamu záznamů. Každý záznam je tvořeny z pole bytů a je unikátně indexovány od 1. Za zachování konzistence a integrity databáze jak při práci s ní, tak během nestandartních situací jako je selhání zdroje energie, odpovídá implementace J2ME (tento požadavek je velmi težký splnit a proto u mnoha implementací není překvapivé, že po delší době dochází k poškození konzistence).

Sada může z bezpečnostních důvodů manipulovat pouze s databází patřící jí. V novější verzi MIDP2 je sdílení databáze možné, pokud si to sada vytvářející databázi explicitně přeje a nadefinuje. Při odstranění sady MIDletů ze zařízení jsou Aplikačním manažerem automaticky odstraněny i všechny její databáze.

Následující obrázek[4] ze serveru Sun.com ilustruje vztah sady MIDletů k RMS:



A zde pro ilustraci obrázek[5] ze serveru Computerbase.de operace a stavy ve kterých se databáze může nacházet:



User-Interface

Uživatelský interface je logicky rozdělen na 2 sady API:

1) **High-level**
2) **Low-level**

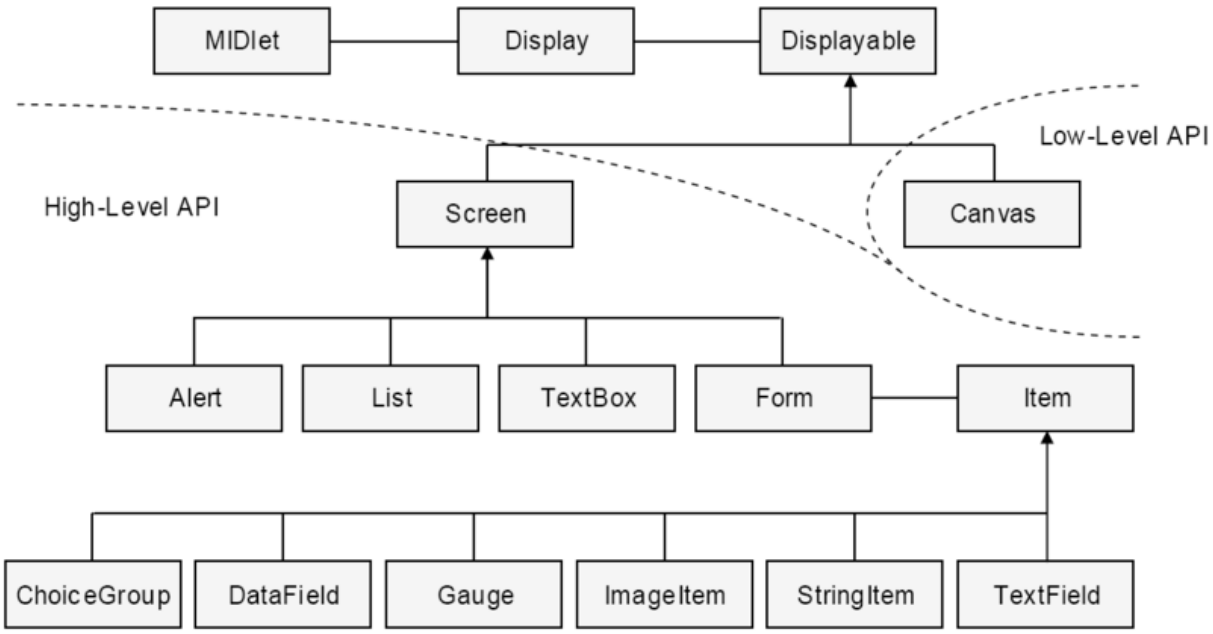
High-level

Je mnohem přenositelnější, díky větší abstrakci. O kreslení a interakci s uživatelem je zodpovědná konkrétní implementace MIDP.

Low-level

Je zase mnohem flexibilnější. Aplikace má přímou kontrolu nad vzhledem, primitivními událostmi a interakcemi s uživateli, ale vzhled aplikace na jednom zařízení nemusí být stejné jako na druhém (kvůli velikosti displaye, počtu barev…)

Pro názornost uvádím obrázek[6] ze Sun.com ilustrující hierarchii tříd User Interface v MIDP:



Kapitola 4

Návrh a implementace

Aby projekt reflektoval požadavkům zadání, museli jsme v MujMailu navrhnout struktury, které měli za úkol:

1. **reprezentovat strukturu a obsah emailové zprávy**
2. **logicky rozčlenit zprávy a nabídnout manipulace se zprávami**
3. **nabídnout práci s databázi**
4. **přípojení k síti**
5. **protokoly pro stahování zpráv**
6. **protokoly pro posílání zpráv**
7. **propojit účty, protokoly a zprávy**
8. **graficky znázornit zprávy**
9. **správu kontaktů**
10. **systém hlášení výjímek a lokalizace, nastavení**
11. **grafické propojení modulů**
12. **propojit všechny moduly, inicializace a ukončení aplikace**

Poznámka: Vzhledem k délce některých kódů nebudu vždy vkládat kód, který je podrobně okomentován, ale popíšu pouze princip a koncept jednotlivých částí.

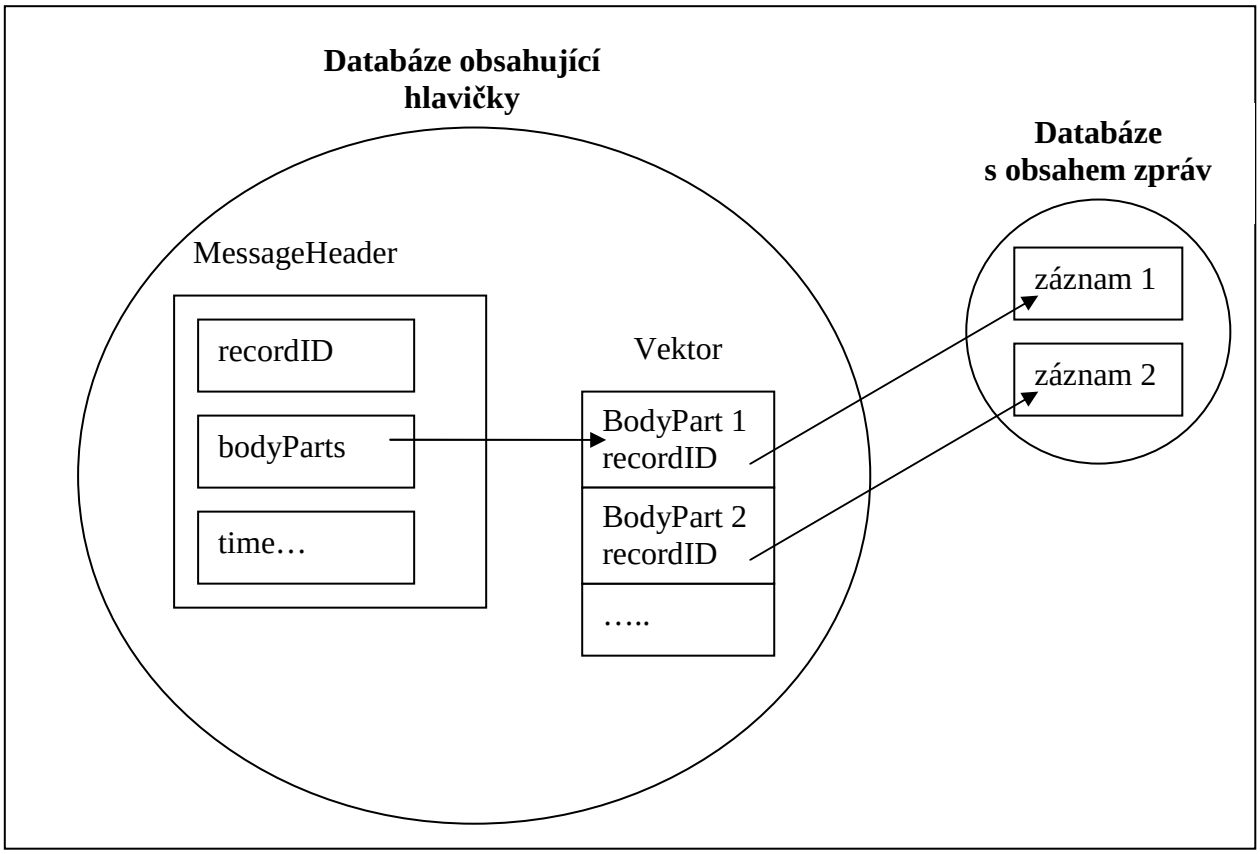
Jak reprezentovat strukturu a obsah emailové zprávy

Pro reprezentaci zprávy slouží třída ***MessageHeader***.

Tedy objekt této třídy obsahuje důležité informace o charakteru emailu jako je čas, velikost, odesílatel...atd. Nedrží však vlastní obsah zprávy. Dále třída MessageHeader obsahuje vnitřní třídu **BodyPart**. Jak již z názvu vyplývá, každý objekt BodyPart odpovídá jednotlivé části těla (příloze). Ale ani tento objekt nedrží vlastní obsah zprávy, ale pouze další podrobnější informace o jednotlivých částech zprávy jako je jméno, znaková sada, velikost..atd. MessageHeader obsahuje jako atribut vektor **bodyParts**, kde jsou uloženy jednotlivé objekty BodyPart. Všechny tyto informace se ukládají do jednoho záznamu databáze(viz. dále v).

A kde tedy leží vlastní obsah jednotlivých částí? Tento obsah je uložen v záznamech jiné databáze. Každý BodyPart se odkazuje na 1 korespondující záznam atributy: **DBFileName** a **recordID**. DBFileName nám říká, v jaké databázi leží záznam a recordID index tohoto záznamu.

Následující obrázek[7] ilustruje tuto situaci:



Asi jsme si všimli, že i MessageHeader obsahuje atribut **recordID**, to je však index záznamu, ve kterém jsou uloženy informace hlavičky MessageHeader.

Pro přístup k jednotlivým atributům a BodyPart slouží jednoduché metody třídy Message. Za zmínku stojí metoda sloužící pro přístup k obsahu zprávy *String MessageHeader.getBodyPartContent()*. Tato metoda zavolá další pomocnou metodu MailDB.loadBodypart(), která načte z databáze obsah záznamu a vrátí jej:

Kód 0:

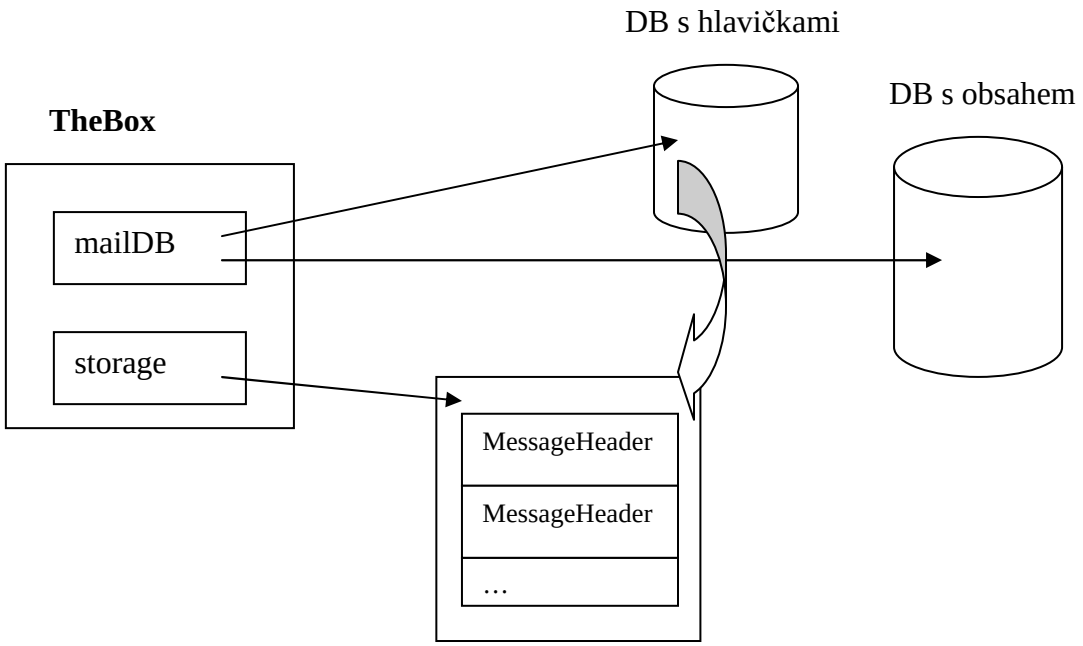
```
public String getBodyPartContent(byte index) {
        try {
                BodyPart bp = (BodyPart)bodyParts.elementAt(index);
                return MailDB.loadBodypart(bp);
        } catch (Exception ex) {
                return null;
        }
}
```

Jak logicky rozčlenit zprávy a nabídnout manipulace se zprávami

Nejen pro tyto účely, ale i pro grafické znázornění hlaviček MessageHeader a znázornění probíhaných processů slouží třída ***TheBox*** a její odvozené třídy ***InBox, OutBox, SentBox, Trash***. Tyto struktury jsou známy i v jiných klientech pod názvem složky.

Každá složka obsahuje hlavičky zpráv, které do ní logicky patří. Proto každá složka má na svoji databázi, kde má uložené svoje hlavičky; a jinou databázi, kde je uložen vlastní obsah zpráv. Na tyto databáze složka odkazuje objektem **mailDB** instancí třídy MailDB. Po inicializaci programu se hlavičky načtou z databáze a vloží se do **vektoru storage**. Vlastní obsah zpráv se z důvodu šetření RAM nenačítájí z databáze a přístupuje se k nim jen na požadek uživatele, kdy si chce přečíst zprávu, viz. přechozí povídání.

Následující obrázek[8] ilustruje tuto situaci:



Složky nabízejí pro práci se zprávami metody:

1. pro mazání zprávy jak z vektoru storage, tak i z databáze
2. pro přidávání zpráv do databáze a vektoru storage

Pro mazání slouží metody *void delete(MessageHeader), void deleteNow()*. Kvůli minimalizaci přístupů k databázi se nemažou zprávy po jednom, ale naráz všechny, co jsou označené jako smazané metodou delete(MessageHeader). Faktické mazání se stane, když se zavolá metoda deleteNow(), která pro práci s databázi zavolá metodu mailDB. deleteMails().

Pro ukládání zpráv do vektoru storage a databáze slouží metoda *MessageHeader storeMail(MessageHeader)*. StoreMail() po uložení všech těl a hlaviček metodou mailDB.saveBodypart(BodyPart), resp. mailDB.saveHeader(MessageHeader), uloží hlavičky do vektoru.

Zobrazování hlaviček probíhá tak, že při přístupu do složky se metodou *void paint(Graphics)* projedou všechny hlavičky ve vektoru storage (ovšem do té doby než se vejdou na výšku displaye) a graficky znázorní jejich důležité informace jako je předmět, čas, odesílatel..atd, ale také ikonky odpovídající stav této zprávy, jestli je zpráva přečtená, poslaná, označená pro smazání…atd.

Složky mohou fungovat také jako grafické znázornění progress stavu operací probíhaných nad zprávami v této složce a to, když se hlavičky načítají, mazají z databáze nebo když se stahují a mažou ze serveru. Probíhá to tak, že pokud se taková to činnost zdetekuje metodou *boolean isBusy()* (která je u každého typu složek jiná) tak složka nakreslí progressbar metodou *void paintProgress(Graphics)*. Stav progressbaru se řídí atributy **int actual, int max** a pro detailní popis prováděné akce slouží atribut **String activity**. Stavy těchto činností můžou jiné objekty (protokoly, databáze..) měnit metodami setProgress(String activity, int max, int actual), updateActual(), updateMax(). Metodou *report(String report)* můžou jiné objekty i ohlásit změnu svoji činnosti či nastalap-li chyba během činnosti. Chyby a vyjímky se rozpoznávají podle počátečních znaků argumentu *report*, a to: ERROR, ALERT, WARNING. Po detekci nestandatní situace se spustí systém pro znázornění výjimek (viz dále).

Trash

Jak název napovídá, složka slouží k obnovení smazaných zpráv z jiných složek. Každá hlavička je opatřena atributem **char MessageHeader.orgLocation** , který identifikuje, kde byla zpráva umístěna (je-li to znak I, tak je to InBox, O jako OutBox…). Metoda *void restoreNow()* se tak podle toho řídí a uloží zprávy zpět přes metodu TheBox.storeMail(MessagHeader).

K jiným složitějším složkám jako je InBox a OutBox se vrátíme pozdějí, až si řekneme něco o protokolech.

Jak pracovat s databází

Databáze je reprezentovana třídou **MailDB**, která nabízí všechny potřebné funkce pro práci s databází. Na vzniku této třídy se podílel i Martin, sice byla během vývoje přepsána, nicméně část metod saveHeader(), loadHeaders() je kreditována jemu.

Jak již bylo řečeno, každá složka je propojena na 2 databáze – jedna obsahuje hlavička, druhá vlastní obsahy.

Jelikož operace probíhané nad databází může trvat dlouho, proto tyto operace běhají vždy v novém vlastním threadu. A to jmenovitě operace pro hromadné načítání hlaviček – privátní metodou *void loadHeaders()*; hromadné mazání hlaviček privátní metodou *void _deleteHeaders()*. Proto složka která chce vykonat tyto činnosti zavolá metody *void loadDB()*, resp. *void deleteMails()* . Tyto metody nastaví atribut **byte runMode** na příslušné hodnoty RUNMODE_LOAD, resp. RUNMODE_DELETE, vytvoří nové vlákno a metoda run() podle atributu zavolá buď metodu loadHeaders() či _deleteHeaders(). Zde je kousek kódu metody run() :

Kód 1:

```
busy = true;
switch (runMode) {
        case RUNMODE_LOAD:
                try {
                        loadHeaders();
                …
                } catch (MyException ex) {...}
                //init the map of already downloaded mails in the Inbox
                if (callBox == callBox.mujMail.inBox)
                        ((InBox)callBox).initHash();
                //its needed to resort the box according to the settings
                callBox.resort();
                break;

        case RUNMODE_DELETE:

                try {
                        _deleteMails();
                …
                } catch (MyException ex) {…}
                break;
        }
busy = false;
```

Načítání hlaviček probíhá tak, že se nejdříve otevře příslušná databáze složky.Vytvoří se ze všech záznamů seznam a pro každý záznam se vytvoří jedna hlavička, načtou se atributy z streamu a přidá do vektoru storage příslušné složky.

Kód 2:

```
RecordStore headerRS = Functions.openRecordStore(callBox.DBFile+"_H", false);
RecordEnumeration enumeration = headerRS.enumerateRecords(null,null,false);
byte[] data = new byte[250];
DataInputStream inputStream = new DataInputStream( new
                                        ByteArrayInputStream(data) );
…
while (enumeration.hasNextElement()) {
    try {
        id = enumeration.nextRecordId();
        …
        headerRS.getRecord( id, data, 0 );
        MessageHeader header = new MessageHeader();//create a new header
        //read the inportant information from the stream
        header.recordID = id;
        header.orgLocation = inputStream.readChar();;
        header.from = inputStream.readUTF();
        header.recipients = inputStream.readUTF();
        ….

        //now create a new bodypart and read its information
        bodyPartsCount = inputStream.readByte();
        for (byte k=0; k < bodyPartsCount; k++) {
            MessageHeader.BodyPart bp = new
                                            MessageHeader.BodyPart();
            bp.DBFileName = inputStream.readUTF();
            bp.recordID = inputStream.readInt();
            bp.name = inputStream.readUTF();
            …
            header.addBodyPart(bp);//add the bodypart to vector bodyParts
        }
        //change the counter of unread mails of the Inbox
        if (isInBox && header.readStatus == MessageHeader.NOT_READ)
            ((InBox)callBox).changeUnreadMails(1);

        hdrRefer.addElement(header); //add the header to the appropriate box
        callBox.updateActual(); //update the progress
}
```

Opačný process k načítávání je mazání hlaviček. Nejdříve se otevřou obě databáze – jedna pro hlavičky a jedna pro těla mailů. A při procházení vektoru storage složky se zkontroluje, jestli je hlavička označená jako smazaná či ne – nastaven flag **boolean MessageHeader.deleted**. Hlavičky a těla se mažou tím, že se smažou záznamy jím odpovídající – dle atributu **int recordID**.

Kód 3:

```
headerRS = RecordStore.openRecordStore(callBox.DBFile+"_H", false);
bodyRS = RecordStore.openRecordStore(callBox.DBFile, false);
MessageHeader header; MessageHeader.BodyPart bp;
//lock the container as were gonna to modify it, so callBox.repaint() and other things will not crash
synchronized (hdrRefer) {
    ...
    for (int i = hdrRefer.size()-1; i >=0; --i) {
        header = (MessageHeader)hdrRefer.elementAt(i);
        if (header.deleted) {
            //move it the trash if it's appropriate
            if (!Settings.safeMode && Settings.moveToTrash && callBox !=
                                                    MujMail.mujmail.trash)
                MujMail.mujmail.trash.moveToTrash(header);

            for(byte j = (byte)(header.bodyParts.size()-1); j >= 0; --j ) {
            …
                bp = (MessageHeader.BodyPart)header.bodyParts.elementAt(j);
                    //was it saved to safemode file or standard box file?
                    if ( bp.DBFileName.equals(callBox.DBFile) ) {
                        if (bodyRS != null) //delete bodypart's record
                            bodyRS.deleteRecord(bp.recordID);
                    }
                    else {
                        MailDB.clearDb(safeModeDBFile);
                        break;
                    }
            …
            if (headerRS != null)  //delete header´s record
                headerRS.deleteRecord(header.recordID);

            hdrRefer.removeElementAt(i);
            --callBox.deleted;
            if (callBox.deleted == 0) //we deleted all marked mails
                break;
        }
    }
    callBox.resort();// we sorted by recordID before the iteration
    //we added new mails to the trash so let's resort it
    if (!Settings.safeMode && Settings.moveToTrash && callBox !=
                                                    MujMail.mujmail.trash)
        MujMail.mujmail.trash.resort();
```

O ukládání hlaviček a těl se starájí metody *int saveHeader(MessageHeader)*, resp. *int saveBodypart(String body, boolean safeMode)*.

SaveHeader() jednoduše otevře databázi se hlavičkami, projde všemi atributy hlavičky a uloží tak obsah hlavičky do jednoho záznamu a vrátí index záznamu do, kterého se informace uložily:

Kód 5:

```
RecordStore headerRS = Functions.openRecordStore(callBox.DBFile+"_H", true);
        ByteArrayOutputStream byteStream;
        DataOutputStream outputStream;
        try {
            byteStream = new ByteArrayOutputStream();
            outputStream = new DataOutputStream( byteStream );

            //if it was stored then this is update procedure
            boolean update = header.DBStatus == header.STORED;
            if (header.orgLocation == 'X') //used for restoring from the trash
                header.orgLocation = callBox.DBFile.charAt(0);
            outputStream.writeChar(header.orgLocation);
            outputStream.writeUTF(header.subject);
            if(header.boundary == null)
                header.boundary = header.messageID;
            outputStream.writeUTF(header.boundary);
            header.DBStatus = header.STORED;
            …
            // save also all bodypart headers
            byte size = header.getBodyPartCount();
            outputStream.writeByte(size);
            for (byte j=0; j < size; j++) {
                outputStream.writeUTF(header.getBpDBFileName(j));
                outputStream.writeInt(header.getBpDBIndex(j));
                …
            }
            …
            if (update)
                headerRS.setRecord(header.recordID,  byteStream.toByteArray(),
                0, byteStream.size() );
            else
                header.recordID =
                    headerRS.addRecord( byteStream.toByteArray(), 0,
                    byteStream.size() );
                 }
            …
            outputStream.close();
            byteStream.close();
        } catch (Exception ex) {…}
Functions.closeRecordStore(headerRS);
return header.recordID;
```

int SaveBodypart(String body, boolean safeMode) jednoduše převede textové tělo (existuje i přetížená varianta SaveBodypart, která ukládá tělo v binární formě. Ta funguje podobně.)

Kód 6:

```
public int saveBodypart(String body, boolean safeMode) throws MyException{
        int index = -1;
        //we will try to minimize using DB by recycling common record store safeModeDBFile for mails
        String file = safeMode? safeModeDBFile: callBox.DBFile;
        RecordStore bodyRS = Functions.openRecordStore(file, true);
        try{
                index = bodyRS.addRecord(body.getBytes(), 0, body.length());
        } catch (Exception ex) {…}
        Functions.closeRecordStore(bodyRS);
        return index;
}
```

Pro načtení obsahu těl slouží metoda StringloadBodypart(MessageHeader.BodyPart) . Tato metoda se podívá na atribut recordID konkrétní bodypart a vrátí příslušný záznam.

Kód 7:

```
        public static String loadBodypart(MessageHeader.BodyPart bp) throws MyException
        {
                String body = null;
                RecordStore bodyRS = Functions.openRecordStore(bp.DBFileName, false);
                try {
                        body = new String(bodyRS.getRecord(bp.recordID));
                } catch (NullPointerException npex) {
                                body = "";
                } catch (Exception ex) {…}
                Functions.closeRecordStore(bodyRS);
                return body;

        }
```

Jak se připojit k síti

Ačkoliv se naše aplikace připojuje k Internetu socketovým připojením, způsob připojení byl navržen tak, aby podporoval i případné HTTP připojení, kdyby se to hodilo v budoucnu. Pro tyto učely vznikla třída abstraktní třída **BasicConnection**, kterou třída **SocketConnection** rozšiřuje o socketové možnosti připojení.

Třída BasicConnection sdílí podobný působ načítání a posílání dat ze serveru s původním MůjMailem – metodu getLine() a sendCRLF().

Tato třída obsahuje abstraktní metody void write(byte[] data), int read(byte[] buffer, int offset, int length), int available(), void close(), void open(String url, boolean ssl), které musejí rozšiřující třídy naimplementovat.

Jinak, jelikož všechny protokoly, jak pro příjem, tak i pro posílání mailů řádkově orientované, jsou metody *String getLine()* a *void sendCRLF* společné v pro všechny druhy implementovany pro všechny připojení stejně.

SendCRLF() jednoduše převede textový příkaz na bytes a pošle jej serveru zavoláním *write((command+CRLF).getBytes())*.

Metoda getLine() má za úkol načíst ze serveru vždy 1 řádek a ten vrátit – to dělá tak, že načítá ze bufferu, dokud nenarazí na znak LF. Načítání dat ze serveru lze přerušit pokud si to uživatel přeje – to je nastaví se příznak **boolean quit** na true metodou *stop();* nebo pokud dojde k nějaké chybě během komunikace – předčasné ukončení, dlouho trvající odezva.. Níže je kód této metody:

Kód 8:

```
public String getLine()  throws MyException {
        if (backMark) { //a flag telling that we want to read the last line again
                backMark = false;
                return lastLine.toString();
        }
        lastLine.delete(0, lastLine.length());
        int max=0;
        char ch;
        int timeout = 0;
        try {
                while (!quit) {
                        while (pos < len) { //read everything thats in chr[]
                                ch = (char) buffer[pos];
                                pos++;
                                lastLine.append(ch);
                                if (ch == '\n') {
                                        return lastLine.toString();
                                }
                        }
                        //wait until get some data from server

                        while ( (max=available()) == 0) {
                        if (timeout > Settings.timeout)
                                throw new
                                MyException(MyException.COM_TIMEOUT);
                        Functions.sleep(100);
```

```
                        timeout += 100;
                    }
                    if(max>buffer.length)
                        max = buffer.length;
                    len = read(buffer, 0, max);
                    pos = 0;
                    timeout = 0;
                }
            } catch (IOException e) {
                throw new MyException(MyException.COM_IN,e.toString());
            }
            if (quit) //user pressed the stop button
                throw new MyException(MyException.COM_HALTED);

            return "";
        }
```

Třída **SocketConnection** rozšiřuje BasicConnection zjevně o možnosti socketového připojení implementací metod void write(byte[] data), int read(byte[] buffer, int offset, int length), int available(), void close(), void open(String url, boolean ssl).

Metoda *void open()* nejdříve vytvoří spojení se serverem pomocí metody Connector.open(), což je knihovní metoda pro vytvoření jakéhokoli připojení, načež se toto připojení přecastuje na packetově orientované připojení, vytvoří se z packetového připojení vstupní a výstupní proudy, s nimiž potom metody *void write()* a *int read()* pracují – zapíšou, resp. z nich přečtou byty. Metoda *close()* toto spojení a proudy naopak uzavře.

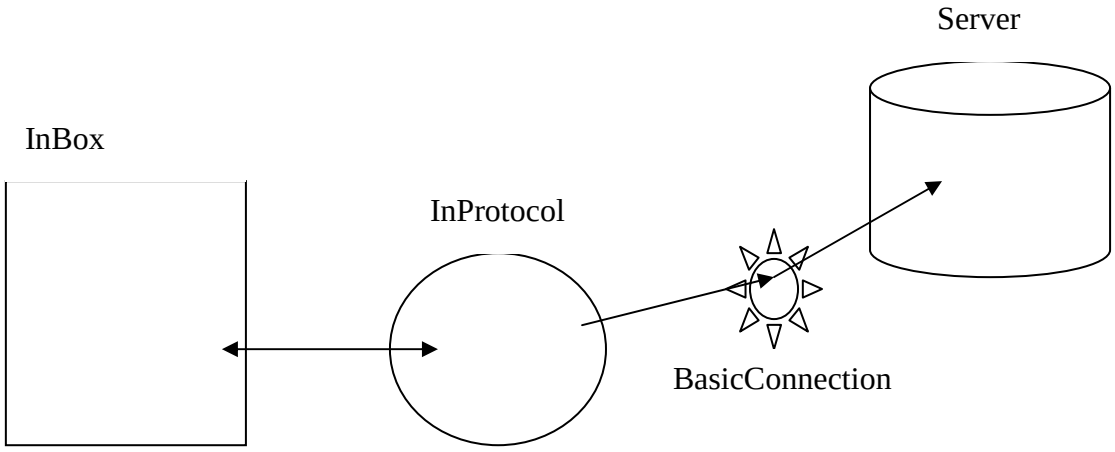
Níže je *kód[9]* metody open():

```
public void open(String url, boolean ssl) throws IOException {
    if (ssl)
        streamConnection = (StreamConnection) Connector.open("ssl://" + url);
    else
        streamConnection = (StreamConnection) Connector.open("socket://" + url,
        Connector.READ_WRITE, true);
    inputStream = streamConnection.openInputStream();
    outputStream = streamConnection.openOutputStream();
    …
}
```

Jak stáhnout a manipulovat se zprávami

Abstraktní třída **InProtocol** nabízí základní rozhraní pro manipulace a stahování zpráv. Každý objekt InProtocol je svázán se složkou **InBox** přes atribut **TheBox theBox**, aby objekt InProtocol mohl do ní přidávat a manipulovat se zprávami uložené v InBoxu a naopak InBox je svázan s InProtocolem, aby mohl zavolat příslušné metody pro práci se serverem. InProtocol komunikuje se serverem přes atribut **connection**, což je instancí třídy BasicConnection.

Následující obrázek[9] zhruba ilustruje tuto situaci:



InProtocol svoji funkce zprostředkovává přes důležité metody:

- *void getNewMails()* – stáhne nové maily ze serveru
- *void getBody(MessageHeader)* – stádne tělo konkrétní zprávy ze serveru
- *void regetBody(MessageHeader, byte mode)* – znovu stáhne poškozené tělo nebo její část.

  Kde mode je index části těla, která se má stáhnout.
- *void removeMsgs()* – smaže označené zprávy v InBoxu ze serveru
- *void poll()* – pravidelně po určeném intervalu zkrontroluje nové zprávy
- *void close()* – uzavře spojení se serverem

tyto metody z důvodu délky trvání musejí běhat v samotném threadu, proto vždy nastaví atribut **byte runMode** na příslušné hodnoty GET_NEW_MAILS, RETRIEVE_BODY, REDOWNLOAD_BODY, REMOVE_MAILS, POLL, CLOSE a spustí abstraktní metodu **void run()**. Potom už závisí na konkretní implementaci protokolu, jak tuto metodu naimplementují. Jelikož uživatel může chtít provádět několik operací najednou se serverem, tak tyto metody musejí být synchronizovány a musí existovat způsob exklusivního zdílení zdrojů (to se provede metodou lock() a unlock()).

dále pro své rozšiřující třídy jako je POP3 a IMAP4 nabízí metody:

- *void parseHeader(MessageHeader)* – z dat od serveru zparsuje důležité info o hlavičce
- *void parseBody(MessageHeader)* – z dat od serveru zparsuje tělo zprávy

ParseHeader(MessageHeader header) načítá ze serveru řádky a podle jejich syntaxí nastaví odpovídající položky hlavičky header (předmět zprávy, čas, boundary…atd). A zjistí zda tělo zprávy je rozděleno na více části (má přílohy) či ne. To pozná, jestli je ze vstupu řádek boundary či ne a nastaví atribut **MessaHeader.messageFormat** na příslušnou hodnotu MessageHeader.frt_MULTI, resp. MessageHeader.frt_PLAIN. I hodnota **MessageHeader.boundary** se změní podle příslušné vstupní řádky. parseHeader() načítá ze vstupu dokud nenarazí na prázdný řádek indikující konec hlavičk zprávy, nebo dojde k nějaké chybě či přerušení.

ParseBody(MessageHeader header) je metoda, jejíž hlavní náplní je rozpoznat ze vstupu požadované části těla a uložit je do databáze složky theBox a vektoru MessageHeader.bodyParts.
Chce-li uživatel stáhnout celé tělo zprávy, ParseBody() bude pracovat tak, že:

- U zpráv bez příloh načte ze vstupu tu část, která začíná prázdnou řádku a končí znakem, indikující konec těla.
- U zpráv s přílohami se snaží rozpoznat jednotlivé části tím, že části začínají prázdnou řádkou a končí řetězcem MessageHeader.boundary

V každým případě se pokusí uložit získané části do databáze a následně do vektoru.
Během tohoto procesu může ovšem nastat velmi nepříjemné situace, se kterými musíme počítat:

- A to, že zpráva je příliš velká a nevešla se do RAM, aby se mohla uložit. V tom případě se zjistí metodou *MessageHeader.canBePartiallySaved(BodyPart)*, jestli se formát přílohy je zobrazitelný (většinou to bývá text), pokud se příloha nestáhla celá a podle toho nastaví flag Me na příslušný **byte BodyPart.bodyState** MessageHeader.BS_PARTIAL (to je že je zobrazitelný) nebo MessageHeader.BS_EMPTY (to je že není zobrazitelný). Pokud je aspoň částečně zobrazitelný, tak se obsah přílohy snažíme uložit do databáze.
- Při ukládání obsahu přílohu může dojít místo v databázi a my ji nemůžem uložit celou. O ukládání se stará metoda *saveBodyPart(MessageHeader.BodyPart bp, StringBuffer content)* . Tato metoda opět zjistí, jestli formát přílohy je zobrazitelný. Pokud ano, tak zkrátí jeho obsah na tolik, kolik zbylo místo v databázi.

V těchto chvílích je, ale část těla považována za poškozenou. Avšak naše aplikace jako správná aplikace pro omezené zařízení dokáže poradit. A to tak, že nabídne uživateli možnost znovu si stáhnout celý mail a poškozené části nahradit nově nepoškozenými nebo znovu stáhnout jenom konkrétní poškozenou přílohu a opravit si jí.

Chce-li si uživatel znovu stáhnout konkretní poškozenou část těla, aplikace přikáže serveru, aby znovu poslal celé tělo (protokol IMAP umí dokonce i poslat konkrétní část těla). Najde se ta část, která se má znovu stáhnout – to se dělá tak, že u zpráv s 1 přílohou jsme hned u požadované části; u zpráv s více přílohy spočítáme kolikrát jsme řetězec boundary přečetli, zjístíme tím u kolikáté přílohy jsme a když jsme u požadované přílohy, začneme

proces parsování. U protokolu IMAP se dostaneme hned k požadované příloze a nemusíme zbytečně pročítat nechtěná data.

U znovu stahování celého těla je situace jednodušší, jelikož nemusíme hledat požadovanou část. Ale parsujeme každou část.

Níže uvedený *kód[9]* ilustruje postup:

```
while (!(line=connection.getLine()).startsWith(END_OF_MAIL)
                                                && !line.startsWith(endBoundary)) {
        ++n;
        bf = new StringBuffer();
        MessageHeader.BodyPart bp = new MessageHeader.BodyPart();

        ////if it's not imap and redownloading a particular bodypart
        if ( !(account.type == MailAccount.IMAP && reDownloadMode != -1) ) {
                //skip everything to the boundary
                while (!line.startsWith(bodyBoundary)) {
                        line = connection.getLine();
                }
                //but if its also endBoundary then its the end of the mail
                if (line.startsWith(endBoundary))
                        break;

                if (runMode == REDOWNLOAD_BODY) {
                        //find an incomplete body part
                        if (reDownloadMode == -1) //redownload complete mail
                                nextIncomplete = n;
                        else {//redownloading a particular part
                                nextIncomplete = reDownloadMode;
                        }

                        //if we want to redownload a body part but it's not the actual one
                        if (n < nextIncomplete) {
                                //skip the line to the body part
                                continue;
                        }
                        //if all incomplete parts have been downloaded, stop the downloading operation
                        if (n > nextIncomplete) {
                                //close(false); //let's save the connection and save the bandwidth
                                break;
                        }
                }

                //parse body parts headers
                //it it has different bodypart's boundary than the actual boundary then its encapsulated email type
                String tmpBoundary = parseBodyPartHeader(bp);
                if (tmpBoundary != null && !tmpBoundary.equals(bodyBoundary))
                //change the actual boundary. this can happen as many time as many encapsulated emails are within
                        bodyBoundary = tmpBoundary;
```

```
        } else {//if its imap and redownloading a particular bodypart
                //then by executing BODY"+"["+reDownloadMode+"] we reach to the right bodypart
                n = reDownloadMode;
                //make a copy, dont call parseBodyPartHeader(bp) as imap protocol reach to datas directly already
                bp = new MessageHeader.BodyPart( header.getBodyPart(n) );
        }
```

Když jsem již doparsovali požadované části, musíme zjisit, jestli nově stáhnuté části jsou lepší než ty staré poškozené – to je že se stáhlo a podařilo uložit víc data. To se provede jednoduchým srovnáním velikostí částí. Pokud je nově stáhnutá část lepší, nahradíme ní starou částí. Nebo pokud se nová část vůbec nevyskytla v předešle stáhnutém mailu, znamená to, že ji uživatel smazal, tak ji přímočaře uložíme.

Kód [10] popisuje tento postup:

```
bp.size = bf.length();
bp.order = n;
if (
        (  (runMode == REDOWNLOAD_BODY && //redownloading the mail
                ( !isInMail(header, bp) //and redownloading deleted body part
                || isBigger(header, bp) ) ) //or the same bodypart is better, bigger than the old one
        || runMode != REDOWNLOAD_BODY ) //or it's just simple downloading
        && bp.bodyState <= MessageHeader.BS_PARTIAL //but check it if it's at least partial
        && !saveBodyPart(bp, bf) ) { //afterall lets try to save it to the DB

                //something gone wrong during the saving process
                box.report(… );
}

//redownloading and the new bodypart that was successfully saved to the DB and replacing
the old bodypart
if ( runMode == REDOWNLOAD_BODY && bp.bodyState <=
                                MessageHeader.BS_PARTIAL && isBigger(header, bp) ) {
        byte old = 0;
        try {
                //delete the old bodypart from DB
                for (old = (byte)(header.getBodyPartCount()-1); old >= 0; --old )
                        if (header.getBpOriginalOrder(old) == bp.order ){
                                MailDB.deleteBodypart( header.getBodyPart(old) );
                                break;
                        }
        } catch (MyException ex) {…}
        header.bodyParts.setElementAt(bp, old); //replace the old one in vector bodyParts
}
//just sipmle downloading or adding a deleted bodypart
else if (runMode != REDOWNLOAD_BODY || !isInMail(header, bp) ) {
        header.bodyParts.insertElementAt(bp, n);  //accept any bodypart - partial, empty, complete
}
```

POP3

POP3 rozšiřuje InProtocol a implementuje metodu *void run()* o možnosti stáhnutí nových mailů, těla a mazání zpráv ze serveru.

Jelikož všechny tyto operace můžou být uživatelem vyvolány najednou, musíme nejdříve uzavřít přístup k sdílenýmu zdroji – atributu **BasicConnection connection**, metodami *lock()* a *unlock()* (ty jsou převzaty ze starého MujMailu). Potom provedeme operace dle nastaveného atribut **byte runMode** (který nabývá hodnot GET_NEW_MAILS, RETRIEVE_BODY, REDOWNLOAD_BODY, REMOVE_MAILS, POLL, CLOSE).

Chce-li si uživatel stáhnout nové emaily:
Nejdříve musíme zjisit kolik emailů aktuálně leží na serveru. To provedeme odesláním příkazu *STAT* serveru. Potom budeme chtít získat identifikátor a index jednotlivých zpráv příkazem *UIDL*. A po získání těchto identifikátorů se podíváme, jestli byly tyto zprávy někdy stáhnuty. Metoda *InBox.wasOnceDownloaded(header.messageID)* se podívá do své hash tabulky a vydá nám výsledek. Zjistíme-li, že zpráva ještě nebyla přítomna v InBoxu, pošleme serveru buď příkaz *TOP index* nebo *RETR index*, abychom získali data hlavičky, resp. celé zprávy. A zavoláním metody *parseHeader(MessageHeader)* a *parseBody(MessageHeader)* provedeme vlastní parsování a ukládání obsahu zprávy. Po parsování nezapomeneme uložit nové informace do databáze metodou InBox.mailDB.saveHeader(MessageHeader). Metodou *void InBox.addToStorage(MessageHeader)* novou zprávu uložíme do vektoru InBox.storage a uložíme i její identifikátor metodou *void InBox.addToMsgIDs(MessageHeader)*, abychom pro příště rozpoznali, že byla stáhnuta.

Chce-li si uživatel stáhnout či znovu stáhnout tělo zprávy:
Zjistíme na jakém indexu zpráva na serveru metodou *String getMsgNum(MessageHeader header)*. Po získání indexu pošle jednoduše serveru příkaz *RETR index* a předá metodě *parseBody(MessageHeader)* hlavičku zprávy, která se má stáhnout.

Chce-li si uživatel smazat emaily ze serveru:
Musí nejdříve zjistit pro každou označenou zprávu v hash tabulce **deleted** její index na serveru. Proto se pošle příkaz *UIDL*, kterým zjistíme identifikátor a index všech zpráv na serveru. Potom porováme identifikátor označených emailů, pokud identifikátor matchuje, pošleme příkaz *DELE + získaný index*. Nezapomeneme se odpojit od serveru, aby server akci DELE provedl definitivně.

Pravidelná kontrola nových emailů:
Probíhá principiálně stejně jako přikontrole nových mailů. Jenom se liší tím, že narazí-li na 1 zprávu nepřítomnou v InBoxu, upozorní uživatele či zavolá getNewMails().

IMAP4

Implementace IMAP4 principilně funguje stejně jako u POP3. Jen se samozřejmě liší syntaxí a tím, že některé její příkazy umožňují rychlejší přístup k datám. Třeba jmenovitě: pro přístup k identifikátorům nepřečtených zpráv může rovnou poslat příkaz *UID SEARCH UNSEEN*; nebo při mazání, známe-li identifikátor označené zprávy, můžeme rovnou poslat příkaz *STORE Identifikátor FLAGS (\\Deleted)* a potrvdíme akci mazání příkazem *EXPUNGE* bez toho, abychom se museli odpojit ze serveru, jako u pop3. Pro získání hlavičky či celé tělo slouží příkazy FETCH index (RFC822.HEADER), resp. FETCH index_zprávy (RFC822). A pro velmi rychlý přístup k požadované části těla provedeme příkaz *FETCH index_zprávy BODY [index_části_těla]*.

Za zmínku stojí upozornit, že serveru IMAP může čas od času poslat jen tak od sebe nějakým data jako upozornění, že došlo ke nějaké změně na serveru, jelikož IMAP podporuje, aby se více lidí připojilo na 1 společný účet. Proto se ke každému příkazu serveru musíme přípojít nějaký tag a zpracovávat jenom odpovědi začínající na stejný tag. Pro tyto účely jsou přizpůsobeny metody *String execute(String command, String arguments) a String execute(String command, String arguments)*.

Jak poslat zprávu

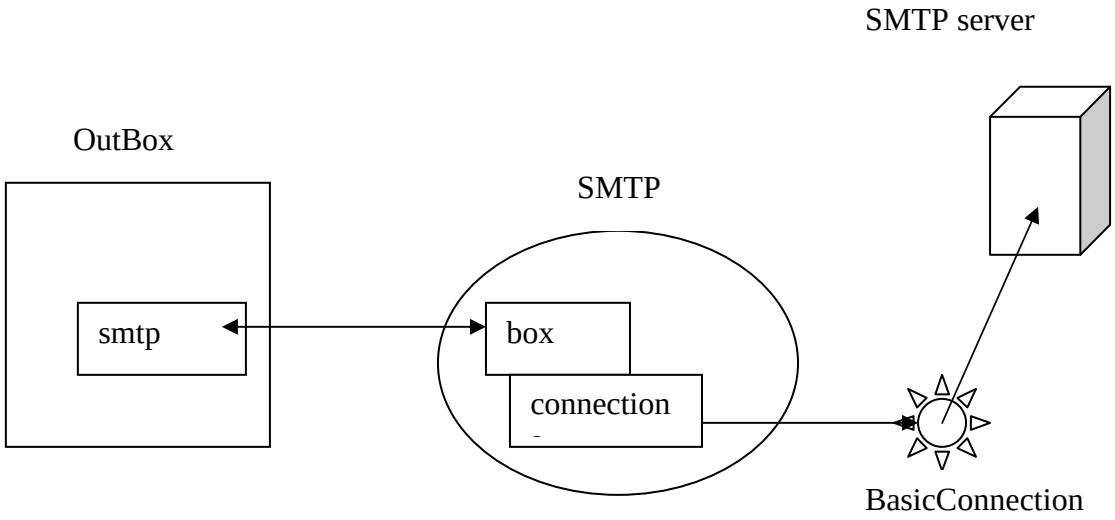
Nejdříve pro vytvoření zprávy tu máme třídu **SendMail**. SendMail slouží jednak jako formulář pro vytvoření zprávy, jednak jako třída reagující na požadavky uživatele, zda chce přeposlat zprávu, odpovědět či editovat zprávu. Metody replyAll(MessageHeader header), reply(MessageHeader header), edit(MessageHeader header), initForward(MessageHeader header) změní vzhled formuláře podle požadovanéhi tvaru pro uskutečnění konkrétní akce. U každé prováděné akce se změní atribut **byte mode** na příslušné hodnoty NORMAL, EDIT, REPLY, FORWARD.

Pro posílání zpráv a komunikaci se serverem slouží třída SMTP.

SMTP

Třída SMTP implementující protokol SMTP je, podobně jako u tříd pro příjem zpráv, navzájem svázana se složkou **OutBox** přes atribut **TheBox box**. OutBox je zas svázan se SMTP přes svůj atribut **SMTP smtp**. Dále obsahuje atribut **BasicConnection connection**, pro komunikaci se serverem.

Následující obrázek[10] nám znázorňuje situaci:



tak jako jiné dlouho trvající operace i posílání zpráv může trvat dlouho. Proto třída STMP implementuje rozhraní Runnable, aby operace posílání probíhala ve samostatném threadu.

Třída SMTP jednoduše realizuje posílání zpráv tím, že po připojení se k serveru projede jednotlivé zprávy požadované k odeslání ležící ve **vektoru storage** složky OutBox a SMTP dialogem se serverem pošle důležité atributy zprávy a vlastní obsah zprávy. Potom požádá o server o odpověď zda posílání proběhlo v pořádku. Začíná-li odpověď na řetězec “250 OK”, potom vše proběhlo v pořádku, zpráva se označí flagem **byte MessaHeader.sentStatus** na hodnotu MessageHeader.SENT, jinak se zpráva označí hodnotou MessageHeader.FAILED_TO_SEND; a pokračuje se v odesílání další zprávy

v pořadí. Tento způsob batch-odesílání zpráv z fronty, který dělá MujMail, je uspornější na data a rychlejší, než kdyby se připojovalo, odeslalo se jednu zprávu, odpojilo se a zase by se připojovalo pro odeslání další zprávy, jak to dělají někteří klienti.

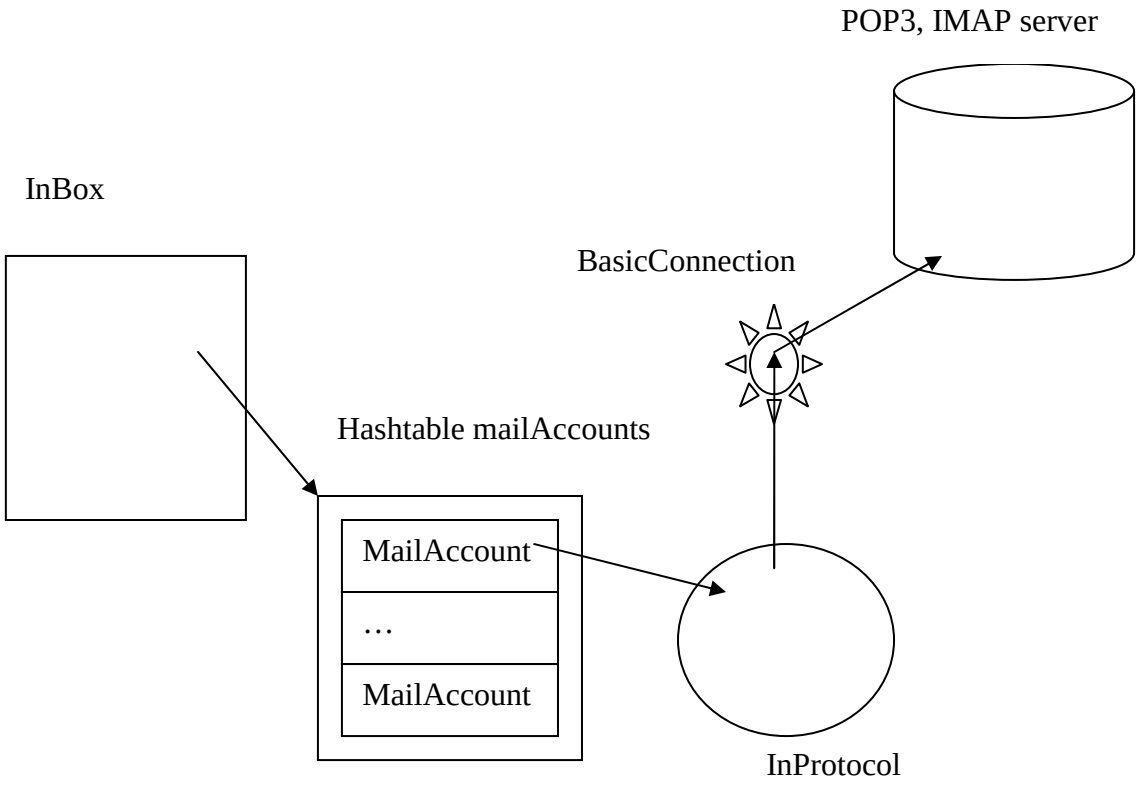
Jak propojit účty, protokoly a zprávy

Účty

Každý POP3 nebo IMAP4 účet je reprezentován třídou MailAccount, obsahující důležité informace o účtu jako je uživatelské jméno, heslo, typ účtu…atd. Avšak o správu účtu se stará třída AccountSettings, kterou jsme s Martinem vytvořili (Martin se podílel na části, kde s pracovalo s databází). AccountSettings jednak zprostředkovává formulář se, kterým se vytvoří nový účet, jednak umí pracovat s databází. Slouží především k uložení (*void saveAccount(String ID)*), smazání (*void deleteAccount(String ID)*) a načtení účtů (*void loadAccounts()*)z této databáze. Principiálně funguje velmi podobně jako MailDB – každý účet je uložen do jednoho záznamu databáze a k těmto záznamům se přistupuje přes atribut **MailAccount.recordID**. Po načtení účtu z databáze se účty uloží do **hashtable mailAccounts**, kde klíčem je emailová adresa a hodnota je objekt MailAccount. Uložení účtů do hash tabulky má výhodu, že se s ní snadno najde, ke kterýmu účtu(resp. serveru) náleží konkrétní zpráva. Každý objekt MailAccount se odkazuje na svůj server přes atribut **InProtocol protocol**.

InBox

Po delší odbočce se zas vracíme k InBoxu. InBox rozšiřuje TheBox o možnost provázání zpráv se účty a následně se servery.
Následující obrázek[11] upřesňuje obrázek[9]:



Chce-li uživatel provést nějakou akci se serverem, musí se to provést přes InBox a následně se správným účtem a serverem. Např:

- Chce-li si stáhnout nové maily ze všech účtů, musíme zavolat metodu *void InBox.retrieve()*, která projde hash tabulkou **mailAccounts** a u každýho aktivní účtu se přes referenci **MailAccount.protocol** zavolá metoda InProtocol.getNewMails().

- Chce-li si stáhnout/znovu stáhnout tělo zprávy, musíme zavolat metodu *void getBody(MessageHeader)*, resp. *void regetBody(MessageHeader)*. Ty najdou pro danou hlavičku správný účet a referenci na server (využíjí atribut **String MessageHeader.accountID,** což je emailová adresa serveru, kde byla hlavička stáhnutá. A s tímto atributem accountID se podívájí do hash tabulky mailAccounts). Potom získání reference MaiAccount.protocol se zavolá metoda InProtocol.getBody(MessageHeader).

- Mažou-li se zprávy z InBoxu metodou *void deleNow()*, musíme nejdříve se podívat na každou označenou zprávu (jako smazanou) a podobně zjistit její správný účet a server. A až potom u získaného objektu InProtokol uložíme do **vektoru InProtokol.deleted** danou hlavičku, která se smazat. Provedeme-li tento krok se všemi hlavičkami, můžeme inicializovat u každého serveru process mazání metodou InProtocol.removeMsgs().

- Pravidelná automatická kontrola zpráv se provede díky vytvoření **Timeru pollTimer**. Ten po dané periodě zavolá process **Polling** (instance třídy Timertask), který projde hash tabulkou mailAccounts a u každého účtu se přes referenci **MailAccount.protocol** zavolá metoda InProtocol.poll().

Inbox navíc má 2 užitečné hash tabulky. Jedna se jmenuje **msgIDs**, která obsahuje identifikátory zpráv – atribut **String MessageHeader.messageID**. A tímto způsobem tabulka mapuje všechny emaily, které jsou nýní uloženy v InBox (přesněji jejich hlavičky jsou ve vektoru InBox.storage). Díky této hash tabulce se vyhneme stáhnutí jedné stejné zprávy dvakrát.
Druhá hash tabulka se jmenuje **onceDownloadedMails**, která přes identifikátory mapuje všechny zprávy, které byly kdy stáhnuty do zařízení od začátku instalace programu. Ta zabraňuje, pokud si to přeje uživatel, aby se stáhla nějaká zpráva, kterou uživatel již viděl, ale smazal ze zařízení.

OutBox

V textu o SMTP jsme se zmínili o OutBoxu. Teď si o něm řeknem něco více. OutBox má jednak metody *void sendAll()* či *void sendSingle(MessageHeader)*, které se přes atribut **SMTP smtp** spustí proces posílání zpráv metodou *SMTP.sendMails()*. OutBox ale navíc umí vytvořit ze formuláře SendMail zprávu. To provádí díky metodě *MessageHeader addOutMail(SendMail sendMail)*. Metoda addOutMail() z jednotlivých položek formuláře vytvorí předmět, adresáta a tělo zprávy. A podle hodnoty **byte SendMail.mode** se rozhodne, zda uloží novou zprávu či jí nahradí starou, krerá se má zeditovat.

Jak graficky znázornit zprávu

Grafické znázornění hlavičky a jejího obsahu slouží třída **MailForm**. MailForm byla nejdříve napsána Martinem, ale důležité části nebyly funkční (parsování, kreslení textu, způsob při práci se serverem, databází, reakce na nestandartní situace), tak jsem je musel přepsat.

MailForm funguje tak, že se metodě *void viewMessage(MessageHeader, TheBox)* předá hlavička zprávy, která spustí zobrazování v novém threadu. Činnost tohoto threadu provádí metoda *void run()*, která načte obsah zprávy pomocí metody *MessageHeader loadBody(MessageHeader)*. loadBody(MessageHeader) nejdříve zjistí, jestli hlavička obsahuje nějaké BodyPart ve vektoru MessageHeader.bodyparts. Pokud hlavička nějaké BodyPart obsahuje, loadBody() vrátí zpět tuto hlavičku metodě run(). Pokud hlavička neobsahuje žádné BodyPart, metoda loadBody() se pokusí zavolat metodu InBox.getBody(MessageHeader), aby stáhla tělo zprávy. Po těla zprávy metoda run() zjistí, která BodyPart je zobrazitelný jako první a nastaví, aby atribut **byte currAttachmentID** ukazoval na tuto BodyPart a spustí vlastní vykreslování v metodě *void paint(Graphics)*. Paint() musí nejdříve rozpoznat, jakýho formátu je aktivní BodyPart. Jestli je BodyPart obrázek, tak zavoláním MessageHeader.getBodyPartContent(currAttachmentID) se přístoupí k datám a vykreslí se obrázek knihovní funkcí Graphics.drawImage(). Jestli je BodyPart text, musíme nejdříve po získání obsahu parsovat text, abychom věděli na kolik řádků vyjde a jak budou vypadat tyto jednotlivé řádky.

O parsování textu se stará metoda *Vector parseTextToDisplay(String body, Graphics g)*. Každý řádek je reprezentován třídou **TextDisplayLine**, který obsahuje počáteční index **int TextDisplayLine.beginLn**, a ukončovací index **int TextDisplayLine.endLn**. Každá stránka je reprezentována vektorem řádků. A celý text představuje vektor stránek. Indexy třídy TextDisplayLine nám říkají odkud a kam sáhá konkrétní řádek v textu - ve *Stringu body*.

Metoda parseTextToDisplay() prochází text (String body), načte celé slovo, zjistí, jestli slovo se vejde do šířky displaye či ne. Pokud se vejde vše je v pořádku, posune koncový index řádky o počet znaků slova. Pokud se nevejde, zkusí jistit, jestli by se nové slovo vešlo do nového řádku. Pokud se vyjde do nového řádku, vytvoříme novýho řádek s tímto slovem. Jestli se ani na nový řádek nevejde, musíme ho zlámat na několik částí, aby se tyto částí vešly na display. Podobně, bílé znaky se přidají do řádky, pokud se vejdou; pokud ne tak se přidají na nový řádek.

Následující *kód[11]* demonstruje tento postup:

```
bow = cursor; //beginning of the word is this position
wordWidth = font.charWidth(c);
//read the whole word
while ( (cursor+1) < bodyLen && (c = body.charAt(cursor+1)) != ' ' && c != '\r' && c !=
        '\t' && c != '\n') {
        wordWidth += font.charWidth(c);
        ++cursor;
}

if (x+repliesCounter*2+wordWidth <= dspWidth) {//the word does fit in display width
        //dont forget +1 because of char at cursor position is the last char of the word
        line.endLn = cursor+1; //and String.substring's endIndex is the last char‘s position+1
        x += wordWidth;
```

```
} else {//doesnt fit in
        if (wordWidth+repliesCounter*2 <= dspWidth) { //does fit in next new line
                currentPage = addLineToPage(line, currentPage, pages, maxLinesPerPage);
                //create a new line with the word including the last word's char - cursor+1
                line = new TextDisplayLine(line.endLn, cursor+1, repliesCounter);
        } else { //too long word that doesn't even fit in new line
                int i = 0;
                wordWidth = 0;
                while (bow+i <= cursor) { //let's try to break it to smaller parts that fit in
                        c = body.charAt(bow+i);
                        if (x+wordWidth+repliesCounter*2+font.charWidth(c) <= dspWidth) {
                                ++line.endLn;
                                wordWidth += font.charWidth(c);
                                ++i;
                        }
                        else { //this part of the word doesn't fit in display width
                                //break the word
                                currentPage = addLineToPage(line, currentPage, pages,
                                                                maxLinesPerPage);
                                line = new TextDisplayLine(line.endLn, line.endLn,
                                                                repliesCounter);
                                //and reread the char at position (bow+i) -- don't increase i
                                wordWidth = 0;
                                x = 0;
                        }
                }
        }
        x = wordWidth;
}
```

Metoda parseTextToDisplay() nám tedy vrátí vektor stránek, kde každá stránka je vektorem řádků – objektů TextDisplayLine. S tímto výsledkem, již vypíšem požadovanou stránku snadno. Tedy projdeme každým řádkem vybrané stránky a vypíšem na display podřetězec textu začínající indexem TextDisplayLine.beginLn a končící na TextDisplayLine.endln.

MailForm má 3 módy zobrazení:

- MODE_LIST – když se zobrazuje jenom seznam části těla
- MODE_BASIC – když zobrazujeme první zobrazitelnou část těla
- MODE_BROWSE – když si prohlížíme jiné části těla (přílohy)

V módu MODE_LIST mailForm nabízí možnost smazat vybranou část těla metodou *void deleteBodyPart(byte index)*, která jednoduše využije metodu MailDB.deleteBodyPart(int index). Druhá zajímavější možnost je znovu stáhnutí vybrané části těla metodou void *void regetAndList(MessageHeader header, byte index)*, která vytvoří nový thread a němž se zavolají služby metody *void inBox.regetBody(MessageHeader msgHeader, byte bodyPartToRedown)*.

Jak spravovat kontakty

AddressBook

MujMail má vytvořenou třídu AddressBook, která má za úkol umožnit uživateli spravovat, zobrazovat a vyhledávat kontakty. AddressBook má vnitřní třídu **Contact**, která představuje jeden kontakt. Tyto kontakty se jednak uchovává ve **vektoru addresses**, jednak do databáze, kde každý záznam představuje jeden kontakt.

AddressBook umožňuje formuláři SendMail připojit emailové adresy kontaktů. Nejdříve se tento požadavek uskutečňuje metodou *void addEmails(Form form)*. Vybrané kontakty se uloží do hashtabulky, aby se rychle zjistilo, jestli byl kontakt vybraný, a odebíralo se z této tabulky. Po potvrzení výběru se metodou ***void pasteEmails()*** uloží vybrané adresy do formuláře pro posílání emailů.

Nový kontakt se vytváří formulářem a tento nový kontakt se uloží do databáze metodou *void saveContactForm()* nebo jiné moduly můžou přidávat nové kontakty přes metodu *void saveContact(Contact)* – to se používá, chceme-li automaticky přidat nový kontakt, kterému píšeme email. Pro mazání z kontaktů z databáze se používá metoda *void delete(int index)*, která najde kontakt ležící ve vektoru addresses na tomto indexu a zavolá privátní metodu *void delFromDB(Contact contact)*. Metoda delFromDB(Contact) smaže záznam mající index v databázi uloženou v atributu **Contact.DBIndex**. Kontakty se načítají automaticky z databáze při volání kontruktoru třídy AddressBook, kdy se jednoduše z jednotlivých záznamů vytváří jeden kontakt.

Zajímavější možností AddressBooku je vyhledávání kontaktů na způsob a la T-9. Nejdříve pro rychlejší vyhledávání vytvoříme ‘záložky‘ ve vektoru addresses. Každá záložka představuje písmenko a index; index je místo kde leží první kontakt, který má jméno začínající na toto písmenko. Je to analogie reálného telefonní seznamu. Proto musíme nejdříve setřídit kontakty podle jména. Chceme-li najít nějaký kontakt, zjistíme, jestli pro jeho jméno existuje záložka. Existuje-li záložka, pokusíme se najít kontakt, jehož jméno začíná s nejvíce stejnými písmeny tím, že jména, která lexikálně mají ležet před vyhledávaným jménem, přeskočíme. A nakonec, zkontrolujeme, jestli jsme přeskočili toho moc a dostali jsme se až k jménu, který lexikálně leží až za hledaným jménem. Záložky jsou uloženy v hash tabulce **nameHash**, kde klíčem je první písmeno a hodnota index ve vektoru adresses.

Kód[12] ilustruje tento postup:

```
public int search(String name) {
        if (name == null || name.length() == 0)
                return -1;
        name.toLowerCase();
        //get the closest
        Integer i = (Integer)nameHash.get( new Character(name.charAt(0)) ); index
        if (i == null) //if its first letter was never indexed
                return -1;

        int size = addresses.size(), index = i.intValue();
        String contactName = null;
        //lets find its correct position (index)
        while ( index < size) {
                contactName = ((Contact)addresses.elementAt(index)).name.toLowerCase();
                if ( contactName.charAt(0) != name.charAt(0) ) //has different first letter
```

```
                return -1; //not found
            //name should be after the contactName
            if ( contactName.compareTo(name) < 0 )
                index++;
            else
                break; //found or name lies before contactName
        }
        if (contactName.startsWith(name)) //now check if it really matches
            return index;
        else return -1;
}
```

Systém hlášení výjímek, lokalizace a nastavení

Jelikož způsob zobrazování oznamovacích oken (alerts) v J2ME neumožňuje zobrazovat několik varovných oken najednou ani je nějak zobrazit postupně. Musel jsem vytvořit třídu **MyAlert.** Když si jiné moduly chtějí zobrazit nějaké oznámení uživateli, zavolají metodu *void setAlert(Object callObject, Displayable nextDisplay, String text, byte mode, AlertType type).* Tato metoda vytvoří z tohoto oznámení úlohu – instance třídy **AlertJob** a tuto úlohu přidá do **fronty jobQueue**. Spustí se časovač (Timer), který pravidelně kontroluje, jestli je ve frontě nějaká úloha, dokud je fronta prázdná. Vyvolávání oznámení z fronty probíhá přes metodu *void invokeAlert(AlertJob job)*.

MujMail má definovanou třídu **MyException** pro správu výjimek. Většina operací vyvolávající výjimku jsou operace pracující s databází nebo s připojením. Proto pokud během práce nastane nějaká systémová výjimka, se tato výjimka převede na MyException se specifickou hodnotou (tyto konstanty jsou definovany ve třídě MyException), která se nastaví do atributu MyException.errorCode, aby se mohlo zjistit, kde tato chyba nastala a abychom jí mohli vypsat uživateli metodou *String MyException.getDetails()*.

Všechny textové řetězce – hlášky, popisky, které se mají zobrazit uživateli jsou uloženy ve třídě **Lang.** Kde je každý řetězec specifikován číselnou konstantou uloženou ve třídě Lang. Pokud chceme získat nějaký textový řetězec, musíme zavolat metodu *String Lang.get(Lang.ciselna konstanta řetězce)* , který vrátí odpovídající řetězec napevno určený ve zdrojovém kódu. Tento způsob lokalizace je méně praktičtější, ale na druhou stranu nezabírá tolik paměti RAM, než kdybychom si řetězce ukládali do nějakého kontaineru.

O správu nastavení se stará třída **Settings**. Settings byla vytvořena Martinem (kromě části při práci s bity u theBox.sortMode). MujMail má mnoho nastavení, která jsou statickými proměnnými této třídy (jsou statické, aby se k ním přistoupilo snáze z jakéhokoli modulu aplikace). Settings jednak představuje formulář, kde může uživatel změnit hodnoty nastavení, jednak Settings třída pracující s databází, kam se tyto hodnoty ukládají (ukládá se do prvního záznamu databáze).

Jak propojit graficky moduly

O tuto práci se stará třída **Menu**. Menu napsat Martin jen s minimálním zásahem ode mě(posouvání dlouhého textu). Takže ji popíšu stručně.

Menu se skládá z jednotlivých **tabů**. Takže menu je vlastně polem tabů **MenuTabs[]** tabs. Každý tab má kromě jiného vektor položek a ikonku. Každá položka je reprezentována třídou **MenuItem**, která má atribut pro jméno, hodnotu položky a ikonku.

Takže hlavním úkolem vykreslení Menu, je vykreslit jeden konkrétní tab. Obsah tabů se inicializuje na začátku aplikace metodou *void Menu.init()*.

Jak propojit všechny moduly, inicializace a ukončení aplikace

Tuto práci má na starost hlavní **MIDlet MujMail**, který na začátku vytváří všechny potřebné objekty, nainicializuje potřebné zdroje jako je načítání hlaviček z databáze, načítání nastavení a účtů v metodě *void startApp()*. A propojuje moduly tím, že implemetuje rozhraní CommandListener pro všechny moduly, takže zmáčkneli uživatel nějaké tlačítko, potom je to na metodě *void commandAction(Command c, Displayable d)* , jakýmu modulu předá příkaz. Při ukončení aplikace *void destroyApp(boolean unconditional)* se musí MIDlet odpojit od všech přípojených serverů metodou void discServers(boolean forcedClose).

Kapitola 5

Závěr

Cílem projektu bylo vytvořit plnohotného emailovýho klienta pro J2ME zařízení. Proto největší překážkou bylo, jak navrhnout program, který by byl méně náročný na RAM a uložný prostor a vypořádat se s těmito problémy.

Avšak všechny body specifikace byly splněny.

A jelikož jsem pracoval na programu s vědomím, že i já a můj vedoucí jej budeme používat, tak program byl vyvinut s velmi osobním přístupem. Jelikož program je open-source, tak doufám, že se ostatní dobrovolníci připojí, abychom mohli program stále vylepšovat.

Literatura

[1] Qusay H. Mahmoud: Naučte se Java 2 Micro Edition. GRADA Publishing
[2] Bruce Eckel: Thinking in Java, 3rd Edition. Prentice Hall
[3] Technologie J2ME: www.Sun.com