

University of Göttingen, Center for Informatics, Prof. X. Fu, Prof. D. Hogrefe

Email client application supporting SMTP and POP3

Telematics Practicum, SS 2007

Salke Hartung, Tim Waage, David Koll, Christian Menke
9/1/2007

1) General Project Organization

1.a) Goals

The primary goal was to write a “useable” email client. In particular, it should be capable of

- retrieving mails from a POP3 server as common on the internet today
- storing retrieved mails and allowing the user to read mails
- composing new mails and a “reply” function
- sending mails
- keeping a “persistent state”, i.e. mails should not be lost when the client is closed/restarted
- providing a visually appealing and easy-to-use graphical user interface

These goals led to a number of sub-projects which were then assigned to individual team members.

1.b) Sub-tasks

We decided to use the Model-View-Controller design pattern which we had learned about in the “Ausgewählte Aspekte der Softwaretechnik” lecture for this project. This meant that graphical user interface, data model and controller would be split up into separate classes.

It was obvious that in addition to an SMTP implementation, a POP3 implementation would be required as well. So we needed a GUI, written using Java's Swing, a Controller, one or more classes to store data in, as well as POP3 and SMTP handlers. We decided to assign these separate tasks to the team members as follows:

- GUI and Controller – Tim
- Data model – Christian
- POP3 implementation – David
- SMTP implementation – Salke

The next step was to design a rough time schedule for the implementation of these sub-parts.

1.c) Time schedule / Milestones

Initially we set the following milestones:

1. **May 13th**: Finish modelling, talk about how separate program parts are going to communicate with each other, create UML class diagram and sequence diagram(s).
2. **June 2nd**: Implement basic functionality: Fetching, reading, composing and sending mails. This should have been tested coarsely as well.
3. **June 17th**: All planned features should be implemented by now (without intensive testing). One week of testing and bug fixing after this milestone.
4. **June 24th**: Finish testing and bug fixing. Everything should be implemented and work as intended. No more work needs to be done on the program code.

However during the implementation phase we fell one week behind and missed the June 17th and June 24th milestones by one week. This was due to a number of problems related to mail servers not complying to standards arising in the SMTP and POP3 handler sub-tasks (see POP3 and SMTP section of this report for details.)

2 Data containers

2.a) Requirements

While some parts of the program work with collections of emails (for example the Inbox, Sent, Trash, ... folders), other parts work directly on or with single emails (such as the SMTP handler that sends emails). The logical solution was to model a single email as an object and then have "mail folders" as collections of these objects.

To make working with emails as easy as possible for the GUI and POP3/SMTP handlers, it should be possible to pass a complete email as plaintext to the email representation class and then get single components (such as the To- or From-header) using simple get-methods. For this feature the email representation had to be able to parse plaintext emails into body and individual headers.

Another important requirement was the „persistent state“, meaning that emails stored in any of the folders must not be lost if the client is closed and re-opened. This made saving mail boxes and restoring them from hard disk a requirement. The Mailboxes and EMail in them are saved in a custom, simple XML representation.

2.b) *E*Mail class

The *E*Mail class represents a single email, as described in the requirements. It can receive a plaintext email as an argument to its constructor and will then parse that text into individual mail parts, providing easy access to headers of interest through get-methods.

An *E*Mail object has the following private fields that contain the mail data:

```
private String HeaderFrom;  
private Vector<String> HeaderTo;  
private String HeaderSubject;  
private String HeaderDate;  
private String HeaderMsgId; // Message-Id  
private String HeaderCType; // Content-Type  
private String Body;
```

The `HeaderTo` field is a `Vector` because an email may have multiple recipients. To maintain compatibility with components using the `getHeaderTo()` method, this method returns the header as a single `String`. If there are multiple recipients, they are concatenated and newline characters are inserted inbetween.

There is rudimentary support for character encodings in the *E*Mail class. For all headers affected by characters encodings, there is a `getDecodedHeader*()` method that tries to decode any quoted-printable or base64 encoded sections in the data before returning it. A special character encoding section may look like the following:

```
Wer hat die Mails =?ISO-8859-15?Q?gel=F6scht=3F?=  
The encoded section starts after the '=?' token. This is followed by the character set specifications (green color).
```

The encoded section starts after the '=?' token. This is followed by the character set specifications (green color).

After a question mark '?', the encoding method is specified. It can be either 'Q' for quoted-printable or 'B' for base64 encoding. Another question mark is followed by the actual data. For quoted-printable encoding, characters that can be represented in 7 bit ASCII, remain unchanged. Characters that require encoding are exchanged with an equal sign '=' followed by the hex code for this character in the character set specified. The encoded section ends with a '?=' token.

For base64 encoding, the whole text in the encoded section, in the specified character set, is transformed to it's base64 representation. This allows for easy decoding by reversing the base64 transformation and then translating all characters from the specified character set to the applications character set.

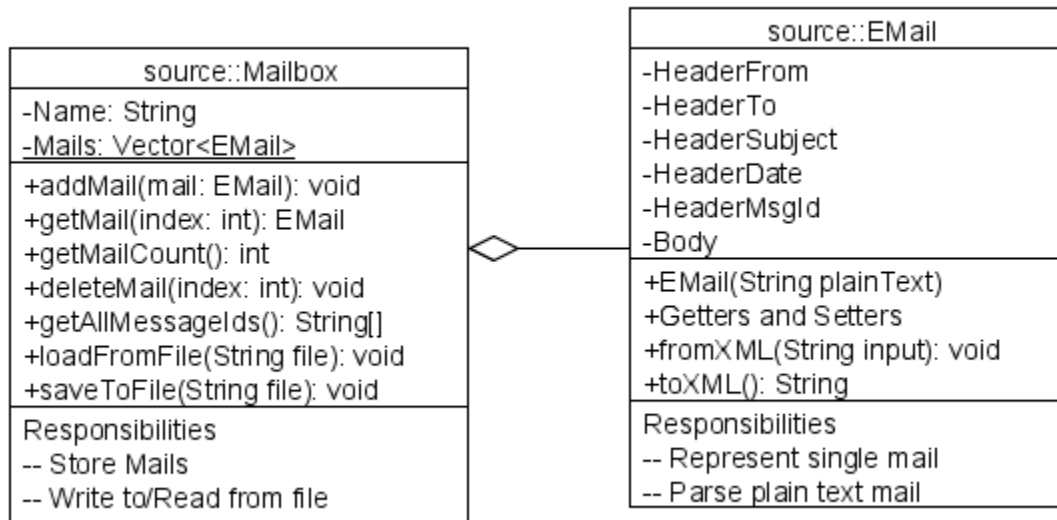
The *E*Mail class also provides a method `toXML()` that returns a `String` containing the XML representation of the email data contained in the object. This allows for easy saving of Mailboxes to files. A method `fromXML()` takes a `String` containing a field name and a `String` containing

the field's value to facilitate parsing and XML data record. This way, a parser for the XML format can pass any sub-nodes of mail nodes down to the EMail object without any knowledge of what a particular sub-node contains.

2.c) Mailbox class

A Mailbox is a collection of email objects belonging to a common folder, for example the Inbox. The internal storage method is a Java Vector, but this is hidden to the outside by using getters and setters. Access to individual EMail objects is gained using a getMail() method that takes an index (int) as a parameter. The number of EMail objects stored can be obtained using a getMailCount() method.

The Mailbox class also provides a public saveToFile() and a public loadFromFile() method. They are responsible for saving the current Mailbox state to a file on the hard disk drive and loading a Mailbox state from the HDD. Both take one parameter, the file to load from/save to, as a String. They make use of the fromXML() and toXML() methods in the EMail class. Mailboxes are saved in XML format, described in detail in the next section, and are loaded using the Simple API for XML (SAX). A class MailboxParser which extends the DefaultHandler class and overwrites the startDocument(), endDocument(), startElement(), endElement() and



characters() methods was created for this purpose.

2.d) XML file format

There are a number of existing, open specifications for representing emails in XML. However, most of them are very “feature rich” and therefore also complicated. A custom XML mail format was developed for ih-mehl that contains elements for all fields contained in EMail objects, but no overhead. This leads to a simple implementation of both XML output and parser. A sample snippet of a single EMail object in XML representation might look like this, with blue color denoting markup and green color denoting actual mail data:

```
<mail>
  <hdr_message-id>&lt;465C1C22.8070000@davion.de></hdr_message-id>
  <hdr_from>Christian Menke &lt;cmenke@davion.de></hdr_from>
  <hdr_to>ih-mehl@davion.de</hdr_to>
  <hdr_subject>Wer hat die Mails =?ISO-8859-15?Q?gel=F6scht=3F?=</hdr_subject>
  <hdr_date>Tue, 29 May 2007 14:28:58 +0200</hdr_date>
  <body>

Ist der Mailaccount leer, oder was?!
Na egal, jetzt nicht mehr!</body>
</mail>
```

These `<mail>` nodes are the only sub-nodes allowed in the `<mailbox>` node, which is the XML files' root node. The `<mailbox>` element may have a `name` attribute, which contains the name of the mailbox.

The `saveToFile()` method in `Mailbox` generates a `<mailbox>` element (including a `name` attribute if the name is set), then calls `toXML()` on each object contained in the internal EMail Vector and outputs the result, following by a closing `</mailbox>` tag.

2.e) Classes using EMail and/or Mailbox

The SMTP handler uses a single EMail object to represent the email it is sending. The POP3 handler fetches mails into EMail objects and adds them to a Mailbox. The GUI has Mailbox instances for each of the „folders“ and the From-/To-headers, Date headers and Subject headers visible in the tables are read from the EMail objects in those Mailboxes. When composing a new mail, anything entered by the user is saved in an EMail object which is then passed to the SMTP handler.

3 POP3 implementation

3.a) Overview

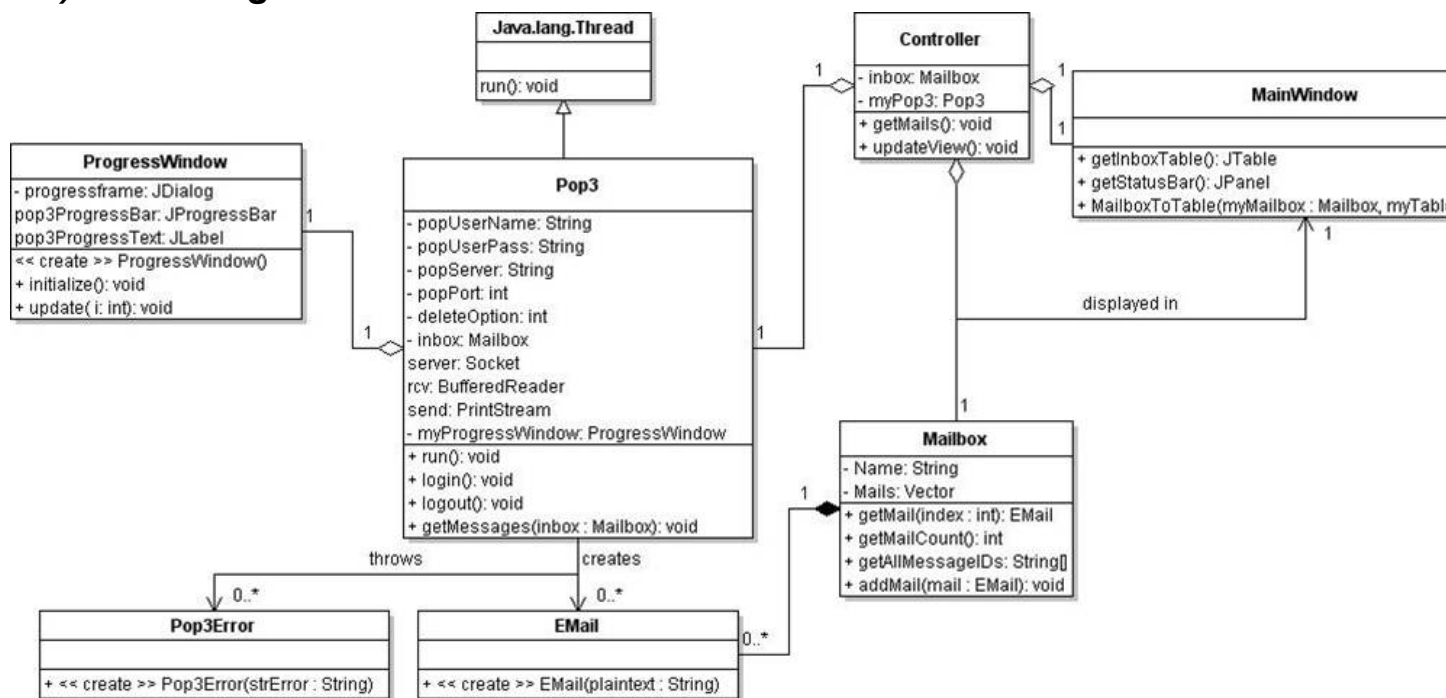
To implement the POP3 functionality we had to stick on RFC 1939. The client needed to be able to login at a POP3 server and to get mails from that server. Furthermore these mails had to be stored in an EMail object, whereas each EMail object had to be stored in a Mailbox object. Finally, the client had to be able to delete messages on the server.

All of these features were mandatory and implemented successfully. Moreover the client now supports threading, deals with attachments (in a declaredly dirty way) and shows its progress via the GUI.

Basically there were no problems while implementing the POP3 process, apart from even big providers ignorance of RFC conformity (like web.de replying to a RETR command in a way not consistent with the RFC 1939).

You will find a UML diagram under section b) and implementation details under section c)

3.b) UML diagram



3.c) Implementation details

Pop3.login();

This function tries to login on a server. The following settings are needed: user name, user password, server address, server port. Those settings are stored in the variables

```
private String popUserName;  
private String popUserPass;  
private String popServer;  
private int popPort;
```

and set by the controller via the given setters.

When logging in a socket is created with input and output streams at the beginning. This is done by

```
server = new Socket(popServer, popPort);  
rcv = new BufferedReader(new InputStreamReader(server.getInputStream()));  
send = new PrintStream(server.getOutputStream());
```

Afterwards the client is able to interact with the server. For logging in the USER and PASS commands as given in RFC 1939 are used:

```
send.print("USER " + popUserName);  
send.print("PASS " + popUserPass);
```

Each response sent by the server is checked for whether it is +OK, signalling a positive response where everything went fine, or not. This is done for each step in the whole POP3 implementation.

```
if (!response.startsWith("+OK")){  
    throw new Pop3Error("Password wrong: " + response);  
}
```

If everything went fine, this is followed by a switch to transaction state, as written in the RFC.

Pop3.getMessages();

This function is the main method of the client's POP3 part. First of all, it checks for the given state (transaction state has to be given). Afterwards it uses the STAT command of RFC 1939 to get information about the status of the locked mailbox (how many mails are there, ...)

```
send.print("STAT");
```

STAT returns a server response, where the number of mails begins at the 5th digit (index 4). Because the number of mails may be longer than one digit, there has to be a loop over this (with 32 representing whitespace in ASCII):

```
while (response.charAt(maxmails) != 32) {
    mailcount = Integer.parseInt(new String(" " + mailcount +
        response.charAt(maxmails)));
    maxmails++;
}
```

After the number of mails is known, the progress window is initialized and the current inbox is sorted with a binary search supported by the Arrays class of JAVA. This is important later on, when the inbox is checked for duplicates.

```
String[] checkIds = inbox.getAllMessageIDs();
Arrays.sort(checkIds);
```

Loop through the mailbox on the server subsequently in order to get each mail.

```
for(int i=1; i<=mailcount; i++){
    RETR each mail;
}
```

The loop in detail: The client sends the server the RETR command given in the RFC with the number of the actual mail.

```
send.print("RETR " + i);
```

Afterwards there is another loop to go through, when fetching the mail, which contains several lines, that have to be read out of the BufferedReader. If the end of the mail has not been reached yet, the line that is read via `rcv.readLine()` is added to the `plainmail` variable. As soon as the end of the mail is reached (identified by a line consisting of

a single "." or in cases of an attachment by the line "Content-Transfer-Encoding: base64", a new mail is created and added to the `inbox` mailbox via using:

```
EMail mailToAdd = new EMail(plainmail);

if (Arrays.binarySearch(checkIds, mailToAdd.getHeaderMessageId() ) < 0) {

    inbox.addMail(mailToAdd);
    Arrays.sort(checkIds);
}
```

`Arrays.binarySearch` looks for the `HeaderMessageID` of the `EMail-Object` that's about to be added to the `inbox`. If there is a match, the mail won't be added to the `inbox` because it is a duplicate.

After that there is a check, if the `deleteOption` was set by the user. If that is the case, the mail is deleted from the server:

```
if (deleteOption == 1) {
    send.print("DELE " + i);
    send.print(EOL);
}
```

Finally the GUI is refreshed to show the updated inbox in its `MainWindow` via calling the `updateView()` method of the `Controller`.

Pop3.logout();

This method is a very simple one, which terminates the connection with the server. This is done via:

```
send.print("QUIT");
send.print(EOL);
send.close();
```

Afterwards, the POP3 state is set back to authentication state.

Pop3.run();

This method implements the `run()` method of the JAVA class `Thread` and is used to achieve the threading of the POP3 code. Therefore it simply calls the methods `login()`, `getMessages()` and `logout()`. This method is also called by the `Controller` to start the execution of the POP3 code.

4 SMTP implementation

4.a) Overview

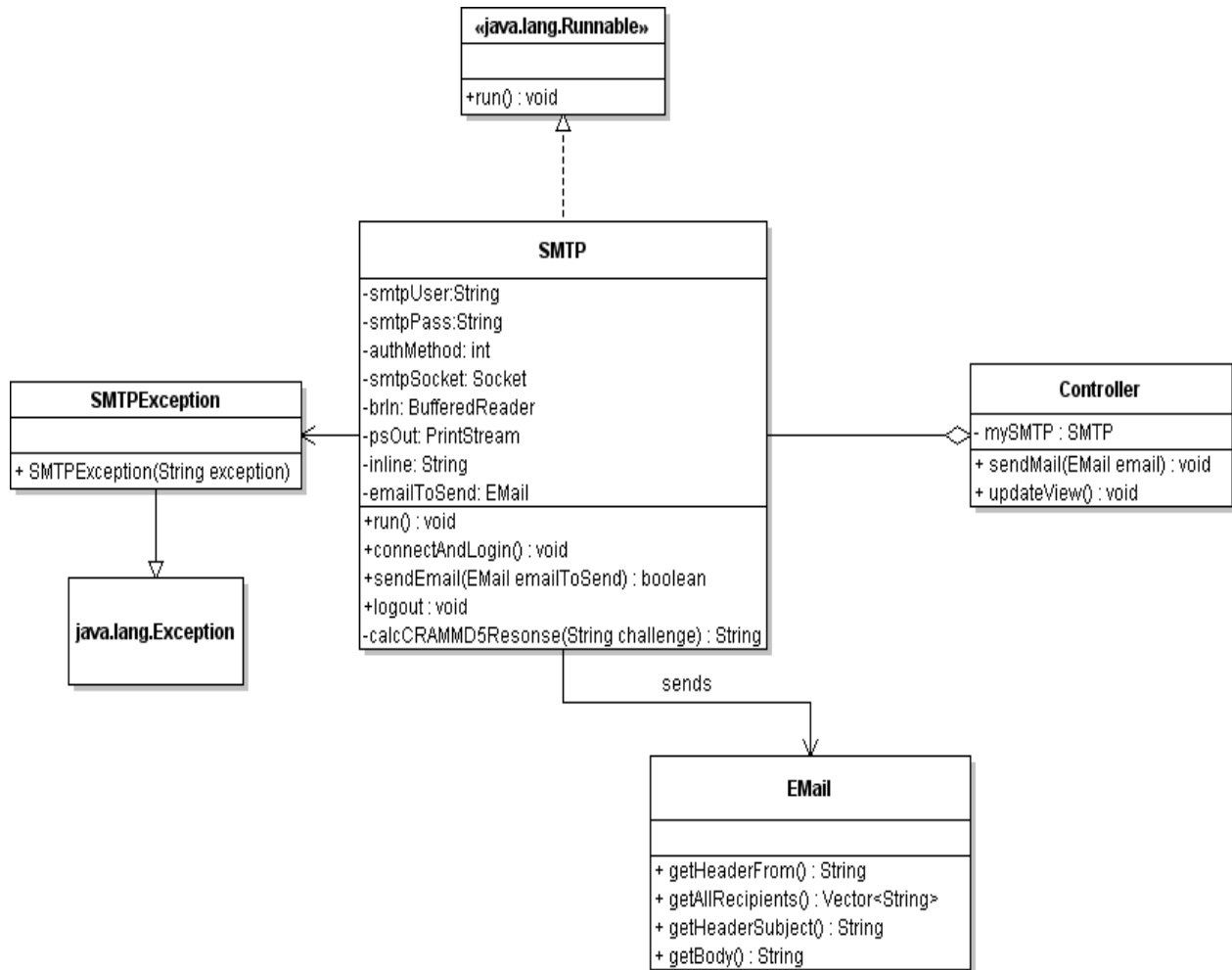
For sending emails, SMTP has been implemented. Ih-Mehl supports all mandatory commands described in RFC 821 and additional commands from the SMTP extensions.

Mandatory features for the client were sending plaintext messages, messages to multiple recipients and basic authentication methods (Plain/Login). In addition to that Ih-Mehl can handle the CRAM-MD5 authentication method, which provides more security and threading support while sending emails to give the user the opportunity to do other tasks meanwhile.

To implement additional SMTP features we had to refer to RFC 1869 for SMTP extensions and RFC 2554/2104 for a description of CRAM-MD5 and the included HMAC algorithm.

All mandatory features have been implemented successfully and were shown during the final presentation in the “live-demonstration”.

4.b) UML-Diagram



4.c) Implementation Details

connectAndLogin()

This method is responsible for establishing a connection to the SMTP-Service and logging in using the in the GUI selected authentication mechanism. It's still possible to connect to open-relays which do not need any authentication by disabling the whole authentication settings.

All important server settings, like host, port and authentication details are stored in the SMTP-class and can be changed using public Getter/Setter-methods.

```
private String smtpServer;
private int smtpPort;
```

```
private String smtpUser;
```

```
private String smtpPass;  
private int authMethod;
```

After successfully creating a socket and opening the input/output-streams the method can start sending SMTP-commands and retrieving SMTP-responses from the server.

```
smtpSocket = new Socket(smtpServer, smtpPort);  
brIn = new BufferedReader(new  
InputStreamReader(smtpSocket.getInputStream()));  
psOut = new PrintStream(smtpSocket.getOutputStream());
```

Ih-Mehl first uses the HELO-command to say hello to the server and after that checks for the selected authentication method. AUTH-PLAIN and AUTH-LOGIN are actually very simple, basically all Ih-Mehl has to do is sending username and password encoded in Base64 via plaintext message. As an example, consider these two lines from the AUTH-LOGIN code:

```
sBase64Output = Base64Coder.encodeString(smtpUser);  
psOut.println(sBase64Output);
```

The Base64Coder-class has been taken from a public code exchange forum on the web.

More interesting is the CRAM-MD5 method since it provides more security because username and password are never sent using plaintext messages. CRAM-MD5 is a challenge/response mechanism, meaning the server is sending a challenge the client has to respond to. All messages between server and client are Base64 encoded for CRAM-MD5, so the first thing the client has to do is to decode the challenge.

```
byte challengeDecoded[] = Base64Coder.decode(challenge);
```

After that, the HMAC algorithm is performed as described in RFC 2104 with one restriction: The implementation of HMAC in Ih-Mehl has a limit of 64 Bytes for all hashing inputs. That's sufficient for most applications but does not completely fulfill the standard. First step is to calculate the inner and outer padding: (just inner padding shown here)

```
char key[] = smtpPass.toCharArray();  
byte keyHashInputInnerPadding[] = new byte[64];  
for(int i=0; i<64; i++)  
{  
    if(i<key.length)
```

```

        keyHashInputInnerPadding[i] = (byte)key[i];
    else
        keyHashInputInnerPadding[i] = 0;

    // xor keyHashInput with iPad

    keyHashInputInnerPadding[i] ^= 0x36;
}

```

When the paddings have been calculated the MD5 hashing function is used to calculate a hash of the paddings and the server challenge:

```

MessageDigest md5Inner = MessageDigest.getInstance("MD5");
md5Inner.update(keyHashInputInnerPadding);
md5Inner.update((byte[])challengeDecoded);
byte md5OutInner[] = md5Inner.digest();

MessageDigest md5Outer = MessageDigest.getInstance("MD5");
md5Outer.update(keyHashInputOuterPadding);
md5Outer.update(md5OutInner);
finalHash = md5Outer.digest();

```

Username and finalHash are encoded in Base64 and sent back to the server.

sendEmail(EMail emailToSend)

If authentication has been successful lh-Mehl can start to send the email. As an example, the "MAIL FROM" cmd is given here:

```

psOut.println("MAIL FROM: " + emailToSend.getHeaderFrom());
inline = brIn.readLine();
printSMTPOutput(inline);
if(Integer.parseInt(inline.substring(0,3)) != SMTP_OK)
{
    throw new SMTPException("Server did not accept sender: " +
inline);
}

```

lh-Mehl checks every response code from the SMTP service. If the operation hasn't been successful a SMTPException is thrown back to the invoker. Since the email-objects don't include a header for the DATA-part of the email to be sent, the SMTP code has to build it itself:

```

// from:
psOut.println("From: " + emailToSend.getHeaderFrom());

// to:

```

```

String toLine = "To: ";
for(int i=0; i<recipientsCount; i++)
{
    recipient = emailToSend.getAllRecipients().get(i);
    toLine += recipient;
    toLine += ", ";
}
toLine.substring(0, toLine.length()-3);
psOut.println(toLine);

// X-Mailer:
psOut.println("X-Mailer: " + Controller.getAppTitle());

// subject:
psOut.println("Subject: " + emailToSend.getHeaderSubject());
psOut.println("");
psOut.println(emailToSend.getBody());

```

Without the header information, the emails are sent but look like spam because of the missing header-fields for sender and recipient. All webinterfaces of public email providers like gmail.com, web.de, gmx.net marked emails without header information as spam.

At the end lh-Mehl can disconnect from the server and close all streams and the socket:

logout()

```

psOut.println("QUIT");
inline = brIn.readLine();
printSMTPOutput(inline);
if(Integer.parseInt(inline.substring(0,3)) != SMTP_SERVICECLOSING)
{
    throw new SMTPException("Server did not send quit message: " +
inline);
}
psOut.close();
brIn.close();
smtpSocket.close();

```

To implement threading functionality the SMTP code implements the Runnable interface and defines the run()-method:

```

public void run() {
    try
    {
        connectAndLogin();
        if(sendEmail(emailToSend))
        {
            Controller.getSent().addMail(emailToSend);
            Controller.updateView();
        }
    }
}

```

```
    }  
    else  
    {  
        Controller.getDrafts().addMail(emailToSend);  
        Controller.updateView();  
    }  
    logout();  
}  
...
```

5 Controller & GUI Implementation

5.a) Controller UML Diagram

Controller
<u>settingsFile : String</u> <u>inbox : Mailbox</u> <u>sent : Mailbox</u> <u>drafts : Mailbox</u> <u>trash : Mailbox</u> <u>myPop3 : Pop3</u> <u>mySMTP : SMTP</u> <u>myMainWindow : MainWindow</u> <u>myMainWindowListener : MainWindowListener</u> <u>myMainWindowComponentListener : MainWindowComponentListener</u> <u>mySettingsWindow : SettingsWindow</u>
getPop3() : Pop3 getSMTP() : SMTP getMainWindow() : MainWindow getSettingsWindow() : SettingsWindow getSettingsFile() : String getInbox() : Mailbox getSent() : Mailbox getDrafts() : Mailbox getTrash() : Mailbox main(args : String[]) : void start() : void createDefaultSettingsFile() : void createMail() : void replyMail(m : Email) : void forwardMail(m : Email) : void viewMail(m : Email) : void editSettings() : void getMails() : void sendMail(email : Email) : void loadSettingsFromFile() : void loadMailboxFromFile(myMailbox : Mailbox,path : String) : void getAppTitle() : String stop() : void updateView() : void

5.b) Implementation details

A few words about the interesting fields and methods in the Controller class:

First of all the class has the four mailboxes in order to store the mail objects.

```
private static Mailbox inbox;
private static Mailbox sent;
private static Mailbox drafts;
private static Mailbox trash;
```

Incoming mails were saved to the “inbox”. The sent-mailbox contains every mail, which was sent by the user. Already written messages, which are not supposed to be sent now, can be saved to the drafts folder. And every mail deleted from the three other boxes is going to be moved into the trash-box.

The Controller class also contains references to all windows, the GUI consists of:

```
private static MainWindow myMainWindow;  
private static MainWindowListener myMainWindowListener;  
private static MainWindowComponentListener  
myMainWindowComponentListener;  
private static SettingsWindow mySettingsWindow;
```

Very important are the references to SMTP/POP3 classes and the settingsfile as well.

```
private static Pop3 myPop3;  
private static SMTP mySMTP;  
private static String settingsFile = "settings";
```

start()

When the application is about to start, the start()-method creates the objects according to all of these fields. It also adds listeners to some of the window-classes. Then it fills the mailboxes with mails, loaded out of the XML-files, like this:

```
drafts = new Mailbox();  
drafts.setName("Drafts");  
loadMailboxFromFile(drafts, "drafts.xml");
```

When that is done, the GUI is ready to show everything by invoking the updateView()-method. Last but not least, the start()-method has to load the SMTP/POP3-settings from the settingsfile. At the first program-start, no settingsfile exists. So the start()-method creates a default one by invoking its createDefaultSettingsFile()-method.

```
File f = new File(settingsFile);  
if(!f.exists()) createDefaultSettingsFile();  
loadSettingsFromFile();
```

This first default settingsfile looks like this:

```
pop.lund1.de|110|ihmehl@davion.de|salkeistdoof|0|smtp.lund1.de|25|0|ihmehl@davion.de|salkeistdoof
```

That is the place, where all the SMTP/POP3 settings are saved. Every time the settingsWindow is opened, these settings are going to be loaded. And every time the settingsWindow is about to be closed, the input from the GUI will be saved in there. So loading and saving happens automatically.

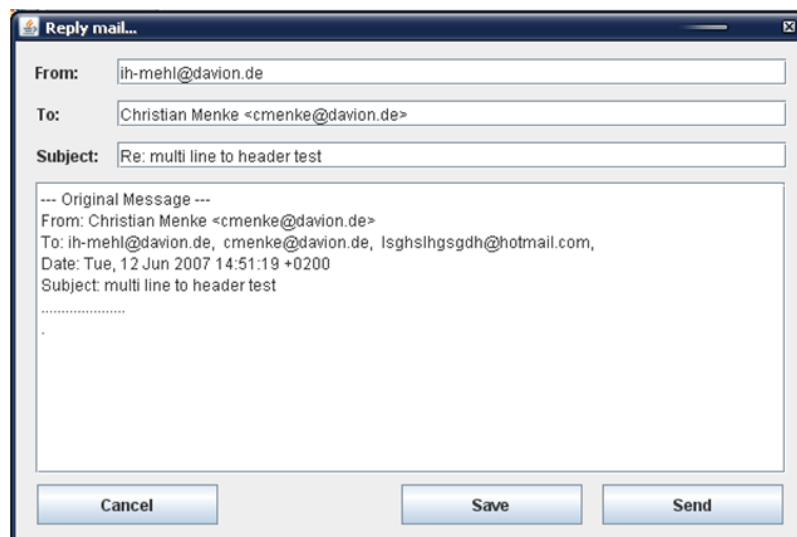
`createMail()`, `replyMail(EMail m)`, `forwardMail(EMail m)`,

All of this methods in the Controller class can be used in order to compose new mails. They all look pretty similar, here the `replyMail(EMail m)` for example:

```
public static void replyMail(EMail m) {
    ComposingMailWindow cWindow = new ComposingMailWindow((byte)1,
m);

    cWindow.setModal(true);
    cWindow.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
    cWindow.setVisible(true);
}
```

It's easy to see, that all possibilities of doing a mailcomposing are using the `ComposingMailWindowClass`. That means writing a mail always make use of the `ComposingMailWindow`:



`getMails()`

This method realizes a very essential function of an email-client. It uses the POP3-object to fetch mails from a server, using the actual POP3-settings:

```
public static void getMails() {
    try {
        myPop3 = new Pop3();
        loadSettingsFromFile();
        myPop3.setInbox(inbox);
        myPop3.start();
    }
    catch(Exception e){
        myMainWindow.setStatusbarText(e.toString());
        e.printStackTrace();
    }
}
```

sendMail(EMail email)

The equivalent to the getMails()-method. This one sends a single Email to its specified destination. A separate thread is used.

```
public static void sendMail(EMail email)
{
    myMainWindow.setStatusbarText("Sending email...");

    mySMTP.setEmailToSend(email);
    Thread t = new Thread(mySMTP);
    try {
        t.start();
        myMainWindow.setStatusbarText("Mail sent successfully!");
    }
    catch(Exception e){
        myMainWindow.setStatusbarText(e.toString());
        e.printStackTrace();
    }
}
```

stop()

This method is invoked, when the application is about to be closed. Then it is necessary to save the actual mailboxes back into the XML-files. Therefore the saveToFile()-method of the Mailbox class is needed:

```
public static void stop() {
    inbox.saveToFile("inbox.xml");
    sent.saveToFile("sent.xml");
    drafts.saveToFile("drafts.xml");
    trash.saveToFile("trash.xml");
}
```

So far the interesting and basic controller methods. There are a lot more, but less exciting.

5.c) GUI classes

lh-mehl provides an easy to handle graphical user interface. Therefore it comes with four window-classes. All of them were designed with the Visual Editor Plugin for Eclipse. So there is lots of auto-generated code, which would be completely uninteresting to show here.

The most important window is the MainWindow. It contains tables representing the four Mailbox-classes. From here the user can initiate fetching mails or open the other windows.

The ComposingMailWindow is used in order to write mails, as already mentioned above.

If the user wants to manipulate the POP3/SMTP settings, the settingsWindow becomes visible and the account data can be edited. All changes were loaded and saved here automatically.

Last of all there is the MailViewWindow. That comes in handy, when the user wants to see a message from a mailbox in detail. So it is supposed for displaying a single mail with its content.

6 Conclusion

While writing the application and in particular the GUI in Java proved to be a good choice because Swing reduces the work of developing a GUI considerably, especially in conjunction with Eclipse JDT and the Visual Editor, there were some problems with regard to the protocols implemented and the email format.

The absence of authentication for SMTP and the authentication methods added to the SMTP protocol in additional RFCs were one of these problems.

For the POP3 protocol, some servers did not comply with the RFC. This became a problem when dealing with multimedia attachments to emails: As this was not part of our basic tasks we decided not to realize support for such attachments. In order to avoid the attachments being converted into plain text, which would result in mails being extremely long and the important information being overwhelmed by the plain text representation of the mail it was planned to use the RETR command mentioned in RFC 1939. Using this command should return a number of octets of the mails size, which then could be skipped in the implementation to shorten the mail and cut off attachments leaving a notice informing the user about this action. Unfortunately, most of the servers did not implement RFC 1939 in this way, but used their own messages, where the number of octets of a single mail could not be obtained.

In the actual emails, the 7 bit-structure of the headers and possibly mail body in connection with special characters caused problems. Overall, it was very noticeable that the Electronic Mail service had not been developed in one step but had evolved during its usage. It might be worth considering introducing a new, cleaner design of Electronic Mail in the future.