Designing an Innovative E-Mail Client

Master Thesis

Project Period: April 1^{st} 2005 - October 7^{th} 2005
Student Name: Andrea Rezzonico
Student-No: 00-920-793
Supervising Professor: Prof. Bertrand Meyer
Supervisor Assistant: Joseph N. Ruskiewicz

Abstract

Most of today's e-mail tools do not address the main problem experienced by people who get a lot of e-mail messages: *information overload*.
This project aims to propose an applicable solution to this problem using database technology.
During this work we develop a set of deferred classes which defines the main components of our proposed e-mail client.
Since the proposed tool should deal with a huge amount of data we payed particular attention to performance issues. We develop a search algorithm which, according to our test, achieves a performance comparable with the fastest today's tools.
We believe that the object-oriented design and the algorithm described here are possible steps towards solving the information overload problem in the e-mail domain.

Contents

Chapter 1

Introduction

1.1 Overview

Electronic mail is an essential form of communication in today's society. The ability to have instant communication with friends, family, and colleagues has made it substantially important for common persons.

As more individuals are using e-mail, the mailboxes are getting fuller over time. As a user progresses from an average user to a power user, they are faced with the problem of organizing messages in such a way that they can find and recall them for later reference.

The current set of tools available for the power users are not able to keep up with this ever increasing load of e-mail on their machine. Even the most advanced tools for storing and retrieving e-mail are unable to filter and adjust e-mail according to the user's preferences. Thus the user is required to manage their own way of organizing e-mail into separate "folders" or repositories.

With advanced power users, which have to deal with a huge amount of e-mail data, this is a large problem. The ability of the user to keep up with the incoming e-mail and have them sorted has lead to an *information overload*.

This information overload is known in the e-mail tool domain (e. g. Gmail [13] and Lookout [18]), but these tools have only started to lead the way to solving this problem. The e-mail tool must allow dynamic relationships between e-mails to be created and managed (see [17]), automatically storing, and most importantly, quick and responsive searching across all the repositories.

This work proposes to provide a solution for the power users of e-mail technology. It will provide an object-oriented framework to build a powerful e-mail tool upon with emphasis on the storage and efficient retrieval of e-mails based on users preferences.

1.2 Scope of the Work

This project aims to provide an applicable solution to the information overloading that power users of e-mail are experiencing. To solve this problem the tool must use advanced software engineering techniques and have efficient algorithms for handling the storage and retrieval of e-mails.

The use of advanced object-oriented techniques will provide a foundation for further developers to create and deploy advanced versions of the tool. The foundation provided by the development of this project will be used as base for semester projects each of them focusing on one specialized part of the e-mail tool.

The tool will be responsive and adaptive to the user's needs, efficient algorithms in the domain of machine learning [27, 16] will need to be developed and deployed. The proposed solution will use modern techniques of machine learning to "learn" what the user is intending to do and be able to react to it in an automatic fashion.

Extending upon the work of the machine learning, it will also be quite important that the solution will provide an advanced query language [7] that will allow the user to rapidly find the important e-mails that they are searching for in an easy and efficient manner.

1.3 Intended Results

The goal of this project is to implement and deploy a foundation for future semester projects which should add functionality to the core system in order to develop a complete e-mail client. Possible semester projects are described in chapter 6.

To facilitate the future potential semester projects, the solution will need to provide a solid foundation for the students to work on. This foundation should use proven object-oriented techniques as given in object-oriented software construction [21] and should provide an extensible and reusable framework that will allow the students to continue extending and enhancing the e-mail tool.

The solution shall also provide the essential algorithms and data structures that will implement features required by power users. These algorithms and data structures will implement the indexing, searching, and storage features of the e-mail tool. The storage features will include message filtering that will adapt to the user's manual filtering criteria, eventually learn and automate the process.

Chapter 2

Analysis of Existing Technologies

In the first phase of the project we have reviewed seven tools which provide some way to deal with a huge amount of e-mail messages.

Test data

The test data was a set of 12350 e-mail messages for a total size of 458 MB. The messages were originally organized in a deep folder structure (more than 400 folders). Since some tools had some problems to deal with such folder structure we decide to test them using a "flat view" of the dataset (everything in one folder). Other tools were tested with an adapted folder structure, in these cases we will describe it in the reviews.

2.1 Mairix

2.1.1 Description

Mairix [20] is a command line tool for Linux. It creates an index on a given e-mail set and offers the possibility to search according to some simple but powerful criteria:

- Search in the `To`, `Cc` and `From` header fields (or all three together)
- Search in the `Subject` header field
- Search in the Body
- Search in a timespan
- Search according to the message size
- Search for a substring in the path to the file which store the message

Supported formats

Mairix supports as input format "Maildir", "mbox" and "MH".

In "Maildir" and "MH" formats each e-mail message is stored in a different file. In "mbox" format messages are concatenated and stored in a single file. A special line[1] is inserted between the messages as delimiter.

The output format is an e-mail collection stored in one of the three formats mentioned above. The e-mail collection generated by Mairix can be then browsed with an e-mail client.

If "Maildir" or "MH" is chosen as output format, Mairix will populate a folder with symbolic links to the original messages.

If "mbox" output format is chosen, Mairix has to copy every retrieved message into the output file. This is a slow operation which reduce its search performance.

2.1.2 Evaluation

Indexing process The indexing process is very fast: it took less than one minute to index the dataset[2]. The index is a table which indicates which words appear in which documents.

Searches Mairix offers no support for pattern matching or similar word search. Anyway it emulates similar word search allowing the user to specify how many differences between a search string and a string contained in a document should be accepted[3].

The few and simple search criteria listed in the previous section can be combined in a powerful way leading good search results.

Mairix can easily be configured to automatically populate folders with relevant messages but unfortunally it is not possible to perform searches directly from an e-mail client.

2.1.3 Example Queries

We have configured Mairix to use "Maildir" as input and output format, we have performed the following queries and we obtained the following results:

- All the messages with a size bigger than 100KB (search string: `mairix z:100k-`). This query matched 500 messages and took 0.374 seconds to be executed.
- All the messages from 8 months ago until now (search string: `mairix d:8m-`): this query matched 5288 messages and took 3.439 seconds to be executed.

[1] Usually this line starts with the string "From" and those it is called "From line".

[2] For this test we have used the flat view of the dataset (see paragraph "Test data" on page 3).

[3] It is possible to allow an error and retrieve the messages containing the string "foobar" with the search string "boobar".

- All the messages from John Doe to Jane Doe which are older than three months (search string `mairix f:john d:-3m t:jane@doe.com`): this query matched one message and took 0.025 seconds to be executed

2.2 Ifile

2.2.1 Description

Ifile [16] is an e-mail filter tool that uses machine learning to classify e-mail messages into folders. The algorithm that it uses is called naive bayes. Basically, naive bayes considers each document an unordered collection of words and classifies by matching the document distribution with the most closely matching folder distribution.

Once trained, Ifile hints for each message the folder where the user should archive it according to the user's archive habits.

In order to classify e-mail messages, Ifile considers the following issues:

Age of a word User's preferences are always changing: what is important or urgent today may become obsolete in a few weeks. To adapt its classification criteria to these changes Ifile introduces the notion of age of a word. The age of a word is defined as the number of documents which have been filtered by the system since the word was first encountered. The more a word is "old" the less it is relevant for the classification.

Header trimming Header trimming consists in mapping the header part of an e-mail message to English. Such a process is needed since naive bayes needs as input a stream of words. Ifile consider for trimming only the `From`, `To` and `Subject` header fields.

Feature selection In order to keep the number of features used for the classification reasonably small, Ifile ignores the most common words (e. g. "a", "the", ...) because they are not relevant for classification purposes.

Many lexing methods It is possible to choose the lexing method in order to define what sequences of characters Ifile should consider as a word. Ifile provides three lexing methods:

- **Alpha lexer**: every sequence of alphabetic characters is considered a word.
- **White lexer**: space and punctuation-separated sequences of character are considered a word.
- **Alpha only lexer**: only space or punctuation-separated sequences of alphabetic characters are considered a word.

Stemming Stemming is a process which consider different words with the same root as the same word. Using stemming it is possible abstract the semantics of a set of similar word which are equivalent for information retrieval purposes.

2.2.2 Test Description

We have tested Ifile with an adapted dataset divided into a training set and a test set. As training set we have used the archived e-mail of four specific folders plus the inbox folder. As test set we have used the messages of the current month which was not yet archived.

Content of the folders

- Folder A and folder B contain mail from two friends written in Italian
- Folder C contains messages from a mailing list written in English
- Folder D contains messages from a mailing list written in Italian
- The inbox folder contains mail from different correspondents written mostly in English, German and Italian

Results Very good results were achieved with folders A, B and C.

The bad classification result of the messages in the Inbox folder (see table 2.1) can be explained in a simple way: the messages contained in this folder are not strictly correlated (i. e. they have different correspondents, different languages, ...), it follows that the folder distribution has an high variance (i. e. the distribution does not provide information about the messages in the folder). For this reason the inbox messages tends to be assigned to folders with a more specific distribution.

The bad classification result of the messages in folder D can be explained in the following way: the not correct classified messages of folder D were mostly assigned to folder A, due to this results it seems that the two folders have a similar distribution and the one of folder A tends to match its own messages and the one of folder D.

Folder	**Training set**	**Test set**	**Correct**	**Percentage**
Inbox	255	65	29	44%
Folder A	3906	211	209	99%
Folder B	249	14	13	93%
Folder C	52	20	20	100%
Folder D	109	12	3	25%
Total	4571	322	274	85%

Table 2.1: Ifile test results

2.2.3 Evaluation

Ifile is a command line tool for Unix and it is used for research purposes. We think that, even if this tool does not directly try to solve the information overloading problem, it can still be very useful to hint the user in what (manual) folder the message should be archived.

Moving a message in to a folder is a common and tedious operation; with a classifier like Ifile an e-mail client can provide a "quick link" to move the message into the predicted folder which is, according to our tests, the right one in the 85% of the cases.

Even if the output of Ifile is not what expected the user can still move the incoming message into the correct folder and Ifile can be triggered to learn it in order to maximize its accuracy.

2.3 Zoe

2.3.1 Description

Zoe [28] is a tool written in Java. When launched it opens a port on localhost (port 10080) and with a browser it is possible to connect to this port using HTTP either locally or from a remote location[4].

The users should configure their e-mail accounts (both IMAP and POP are supported). Zoe connects to these accounts in order to download and index the messages.

The messages are sorted by date. A calendar is shown with a link for each day which can be used to retrieve all the messages received during that day (see Fig. 2.1).

2.3.2 Evaluation

Indexing process The indexing process is very slow (it took many hours to index the dataset) due to the fact that Zoe has to download each message from a remote location. The downloaded messages are then stored locally. Since Zoe indexes by default the whole tree of subscribed IMAP folders, we have used to test it the original folder structure of the test data.

Graphical User Interface The GUI is very powerful. The results of a query are listed in the left part. The right part of the interface shows a list of the contributors, of the links, of the mailing lists and of the attachments contained in the retrieved messages providing a context for the search results.

Search The search feature is not very powerful. It consist of a text field only, and no advanced search is provided. The retrieved messages can be sorted by:

- Relevance

[4]HTTP header authentication is used for access control.

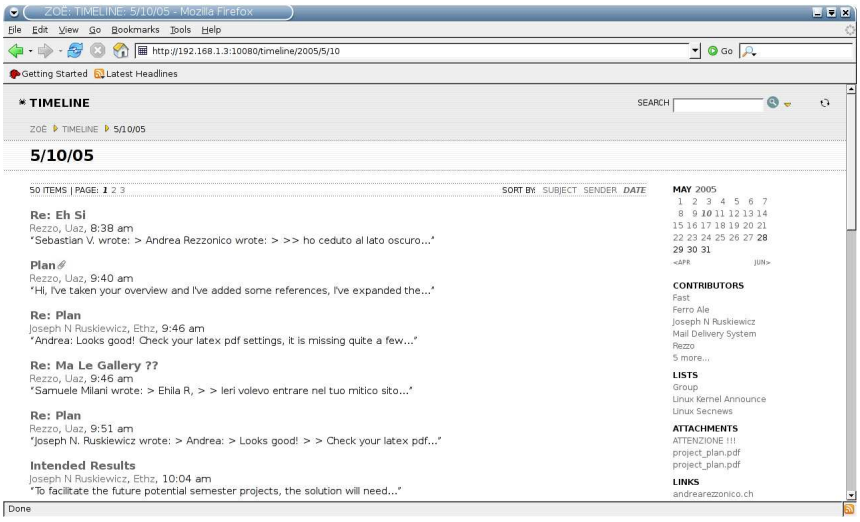

Figure 2.1: Screenshot of the Zoe graphical user interface

- Subject
- Sender
- Date

It is not possible to retrieve all the messages received in a user defined time span, it is only possible to use as time spans days, months and years (i. e. the time spans shown in the calendar). Even if it is not documented it is possible to restrict the search to the `subject`, `from` and `to` header fields using a query string like "`Subject:<search string>`".

2.4 Zoot

2.4.1 Description

Zoot [29] is an information processor used by people who need to manage a large amount of information. It tries to avoid information overload by processing the information as it comes. It offers an efficient system for collecting, reviewing and labeling raw information so that:

- It can be found quickly and easily
- It can be prioritized and classified
- It can be viewed in meaningful time frames and contexts.

Zoot is a general information processor and it is not specific for e-mail messages. It can receive and store in its databases information coming from every MS Windows application, for example a web browser can send the URLs that the user visits to Zoot in order to keep an history. When this information arrives to Zoot it can be manipulated in a lot of ways:

- Assign it to a smart folder either manually or automatically
- Add annotations to it
- Assign a due date or set reminders
- Prioritize it
- Create custom database fields and use them to describe it

In order to manage e-mail data, Zoot connects to the MS Outlook database, retrieves and indexes the messages stored there. For every MS Outlook folder, Zoot provides a database with many predefined views, like "New Entries", "Recent Entries" and "Hot List". These views are represented as smart folders in the GUI and users can access to the information that they contain in a few mouse clicks. It is possible to specify custom views of the data according to many criteria in order to automate searches.

2.4.2 Evaluation

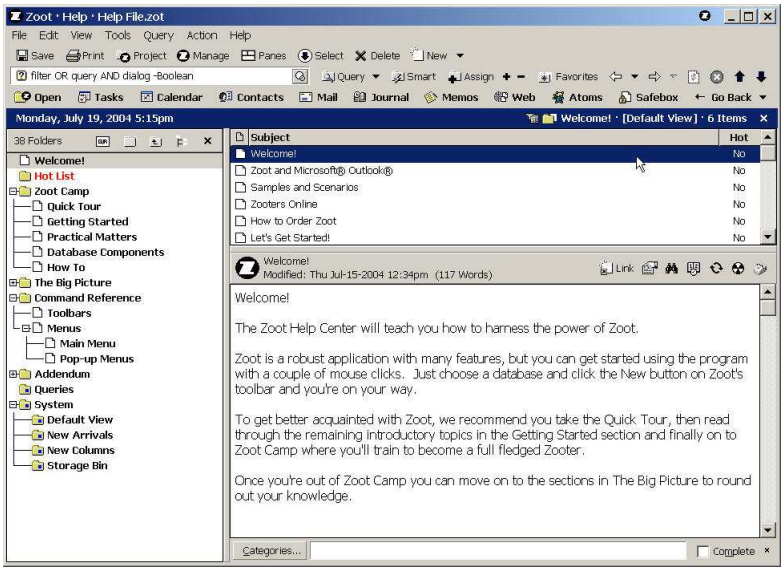

Figure 2.2: Screenshot of the Zoot graphical user interface

Graphical user interface The interface recalls the one of a classic e-mail client, it is divided into three main parts (see Fig. 2.2):

- Smart folder list on the left
- Items summaries on the upper right part
- Selected item view on the lower right part

In addition to these three parts a search bar and a set of buttons are provided for information retrieval purposes. Using the buttons under the search bar it is possible to select the type of items that should be shown (e. g. Tasks, Contacts, ...).

Indexing process The indexing process is very fast, in a couple of minutes Zoot has indexed the whole test data[5].

Search Searches are slower than with Mairix mostly due to the fact that Zoot has to display the results in its GUI. Query results are displayed when the user is typing the query; Zoot adapt the query results at each keystroke. In addition a smart folder is created for each query to avoid retyping the same query many times. Zoot offers an advanced search feature in order to let users combine different search criteria.

2.5 Google Desktop Search

2.5.1 Description

Google Desktop Search (GDS) [12] offers a google search engine for both desktops and web documents.

The following document types are indexed by GDS:

- E-mails stored in one of the following e-mail clients: Outlook, Outlook Express, Netscape Mail, Mozilla Thunderbird and Mozilla mail.
- Chat: AOL Instant Messenger
- MS Office files: Word, Excel and PowerPoint documents
- PDF documents
- Plain-text files

GDS opens a port on localhost (port 4664), there users can find the GDS homepage, which looks exactly like the Google main web page [15], see Fig. 2.3.

[5] For our test we have used the "flat view" of the test data as described in the paragraph "Test data" on page 3.

Figure 2.3: Screenshot of the GDS homepage

Queries are done either using the search bar provided in the GDS homepage, using the search bar in the system tray or using a floating search bar. Advanced searching can be done directly in these search bars using the following operators:

- Group of words ("`foo bar`"): this query will retrieve the documents which contain both the word "foo" and "bar"
- Group of words with exclusion ("`foo -bar`"): this query will retrieve the documents which contain the word "foo" but not the word "bar"
- Site search ("`site:www.ethz.ch`"): this operator will reduce the search domain to the documents which belong to the ETHZ website
- File type ("`filetype:pdf`"): this operator will reduce the search domain to PDF documents
- E-mail advanced search:
  - Subject search: "`Subject:`"

– Recipient search: "`To:`", "`Cc:`" and "`Bcc:`"
– Sender search: "`From:`"

2.5.2 Evaluation

Indexing process The first index creation took an acceptable time: about 30 minutes to index the filesystem and the test data[6] stored in MS Outlook. After that GDS updates it when the system is idle.

Search Search results are sorted by date per default but it is possible to sort them by relevance. As described in the previous section, advanced search features are provided but, from the point of view of e-mail searches, there are some limitation:

- It is not possible to search only between messages who have an attachment
- It is not possible to retrieve all messages received or sent in a given time span
- It is not possible to find the message bigger or smaller than a given size

GDS offers only search features. It does not offers the possibility to prioritize, to order and to label the information it manages.

2.6 Nelson Email Organizer

2.6.1 Description

The Nelson Email Organizer (NEO) [24] is a tool which works with MS Outlook to help users to manage their e-mail messages.

NEO organizes the incoming and outgoing messages using smart folders. The same message can be viewed in different contexts (correspondent, category, date, status, attachment) in order to make easy for the user to find it later.

NEO keeps a list of hot correspondent and a set of bulk mail folders. The messages from the hot correspondents have an high priority. Bulk mail folders can be used to separate subscription mail from the correspondent mails.

2.6.2 Evaluation

Indexing process The indexing process is very fast, it is possible to start retrieving information a few minutes after the setup. NEO do not copy the messages, which stay in the MS Outlook databases, but it keeps an index on them.

[6] To test GDS we have used the "flat view" of the dataset as described in paragraph "Test data" on page 3

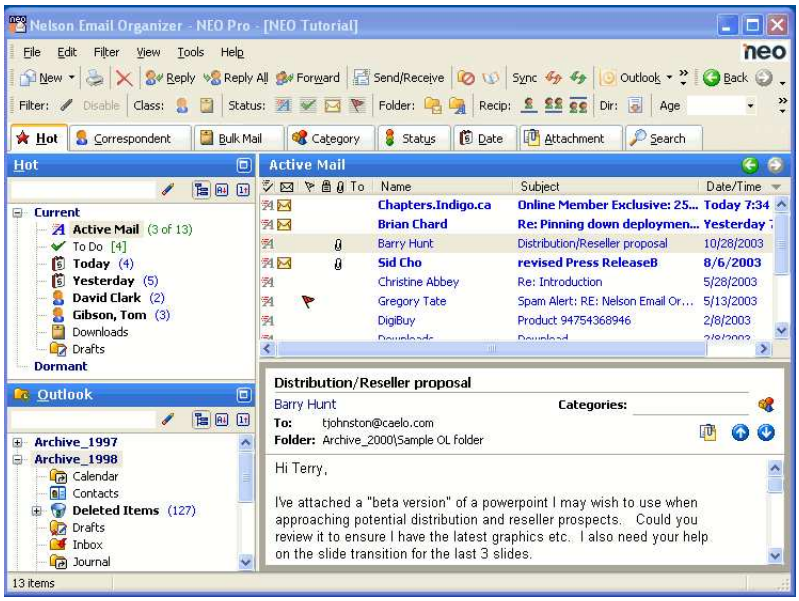

Figure 2.4: Screenshot of the Nelson Email Organizer GUI

Search Messages are sorted automatically by several build-in criteria (hot correspondent, correspondent, bulk mail, category, status and date and attachment). It is possible to define other search criteria. Smart folders can be pregrouped to reduce the search domain in a quick way. The NEO search algorithm supports the matching of similar words.

Since many filter features are already provided by the build-in views, the search dialog is very simple: three checkboxes let user choose what parts of the messages should be included in the search (subject, body and addresses), it is possible to reduce the search domain to the recent messages and there is an option to allow the match on similar words.

Graphical user interface As you can see in Fig. 2.4 the GUI of NEO looks like a common e-mail client (two folder views on the left and two message views on the right part). In addition NEO offers a set of tabs which provide different views of the e-mail messages set. A special bar offers many filters features which can be activated by a simple mouse click. Another bar let users perform the classical operation of an e-mail client (compose, reply, forward, . . .). Since these operations can be done directly from the NEO GUI, MS Outlook is only used as storage engine for the messages.

Correspondents Since a single correspondent may have more than one e-mail address it can be useful to treat a set of e-mail addresses a single correspondent.

With NEO it is possible to assign many e-mail addresses to a single correspondent.

2.7 Lookout

2.7.1 Description

Lookout [18] is a MS Outlook extension which add fast search features for mailboxes and filesystems.[7].

Lookout can index tasks, contacts and every MS Outlook item stored in any folder.

2.7.2 Evaluation

Indexing The indexing process is slow, to index the dataset in "flat view" it took more than four hours. After that the first index is created Lookout updates it automatically in the background when the system is idle. It is possible to trigger an index update using the "Indexer" button located in the lookout toolbar. Unfortunally messages are not indexed as they arrive and leave.

Search Lookout provides a simple search toolbar for quick search. Advanced search can be done using a special syntax directly in the toolbar or using a special dialog.

Lookout offers many criteria to filter the items. In addition to the most common (Sender, Recipients, Subject, ...) it is possible to filter items according to their category, company, file extension and many other criteria. Unfortunally date search criteria can not be used alone but only combined with other search conditions.

Search speed is comparable with the one of Zoot. Lookout displays the time needed to retrieve the query results but it does not count the time needed to display them: if many messages are found, the time experimented by the user and the one shown by Lookout can be very different.

2.8 Remarks

The tools described in the previous sections use many different approaches to help users to deal with their information, between their features we have found the following the most important for the purpose of designing an e-mail client.

Smart folders support The use of smart folders to filter messages, implemented in Zoot and NEO, seems to be a good solution. Using this folder users can organize their messages in many different ways without duplicating them.

[7] Lookout is now acquired by Microsoft, it is still available to use but not developed anymore. The MSN Toolbar Suite provides similar features.

Smart folders can be implemented using a database as described in [1] and information can be retrieved used a special query language [7].

Words aging This technique used in Ifile seems to be a good one in order to select relevant words from a given message. Relevant words can be used for indexing purposes or to retrieve messages which talk about the same topic.

Time sorting The sorting of the messages according to the message date is a useful. Zoe brings this to an extreme level offering only a calendar on its main page. Other tools, like Mairix, propose this in another ways allowing the reduction to the search domain to a given time span. Another way to deal with the age of a message is to divide them into "recent items" and "old items" and provide the option to search only in one of the two category as done in NEO and Zoot.

Search features In order to retrieve messages from a huge data set an e-mail tools should offer advanced features which let the user combine simple search criteria. Mairix shows how a few simple criteria can be powerful. Wildcards and inexact matching (e. g. using stemming) can be nice features to implement in the search engine. An even more powerful feature is the possibility to define a concept using a set of words which belong to the same semantic area (e. g. "DB", "Database", "DBMS"). In this way it is possible to retrieve all the messages about a particular concept (e. g. all messages about databases).

Chapter 3

Definition of the Object Model

3.1 Data Cluster

3.1.1 Foreword

The format of an Internet message was first defined in RFC822 [3] and then updated by RFC2822 [26]. These two RFCs specify the format of an Internet message defining the syntax and the semantic of the heading block of a message, the body of the message was defined as a flat us-ascii text. Using this format it is not possible to send messages which contain media different from plain text encoded in us-ascii.

To further define the structure of the messages' body the original specifications where extended by a set of RFCs [9, 10, 22, 11, 8] called "Multipurpose Internet Mail Extensions" or "MIME". This set of documents extends the format of a message in order to:

- allow textual message bodies in character sets other than us-ascii
- define an extensible set of different formats for non-textual message bodies
- allow message bodies to be composed by many parts (multi-part message bodies)
- allow textual header information in character sets other than us-ascii

3.1.2 Internet Message Format

RFC2822 defines a message as being composed by an heading and an optional body. The heading of a message is a set of (key,value) pairs called header fields.

According to RFC2822 the heading must contain at least two respectively three header fields:

- **Date field**: This field is a string which specify the time when the message was completed and ready to enter in the mail delivery system.
- **From field**: This field is a list of mailbox specifications which identifies the author(s) of the message. A mailbox specification contains an e-mail address and a display name (e. g. `"John Doe" <john@doe.com>`).
- **Sender field**: This field is a single mailbox specification that specifies who is responsible for the transmission of the message. This field must appear in the heading only if the `From` field contains more than one mailbox specification.

There are no other mandatory fields. Example 3.1 shows a minimal RFC2822 message.

```
From:  "John Doe" <john@doe.com>
Date:  Mon, 8 Aug 2005 18:54:58 +0200
```

Example 3.1: A minimal RFC2822 message.

In addition to the `Date` field and the originator fields (`From` and `Sender`) the following header fields should be present[1]:

- `Message-Id` field: this field contains a single unique identifier for the message. The uniqueness of this value is guaranteed by the host that generates it. Some algorithms to generate Message-Ids are proposed in [26]
- `In-Reply-To` field and `References` field: these fields should be present in the heading if the message is a reply to another message. They contain one or more Message-Id values. They can be useful to identify a thread of conversation.

This RFC does not specify a particular structure for the body of a message, this part is defined as a flat us-ascii text.

3.1.3 Multipurpose Internet Mail Extensions

The MIME define a structure for the messages' body which was considered before as being a flat us-ascii text. In order to do that; they defined the following set of header fields (none of them is mandatory):

[1] RFC2822 uses the words MUST and SHOULD with different meanings as described in RFC2119 [2]. MUST indicates an absolute requirement, SHOULD is used as synonym of recommended.

- `MIME-Version` field: until the moment of writing there exists only one version of MIME (i. e. this field is always "`MIME-Version: 1.0`"). It is used as an assertion that the message has been composed in compliance with the MIME specifications. This header, if present, should appear in the heading of the message.
- `Content-Type`: it is used to describe the data contained in the body. It is composed by a media type and subtype followed by a set of parameters. It is not mandatory, if not present the value "`text/plain charset=us-ascii;`" is assumed.
- `Content-Transfer-Encoding`: it specifies what sort of encoding transformation the body was subjected to.
- `Content-Id`: it is used to identify MIME entities in several contexts.
- `Content-Description`: it is used to associate some descriptive information to a given body entity.

These header fields (exception made for the `MIME-Version` header) can occur in the heading of a RFC2822 message or in a MIME body part within a multipart construct.

Media types

The media type is defined using the `Content-Type` header field. Its definition is structured in two parts: the top-level media type and the subtype. The top-level media type is used to declare the general type of data, while the subtype specifies the format for that data. The MIME define seven top-level media types (five "discrete" and two "composite").

Discrete media types These media types refers to discrete bodies which content should be processed by non-MIME mechanisms. The five discrete media types are:

- **text**: it is used to declare textual information. Its default subtype is "plain" which refers to plain text without formatting information.
- **image**: it is used to declare image data. The initial subtype is "jpeg" which refers to the JPEG format using JFIF encoding.
- **audio**: it is used to declare audio data. The initial subtype is "basic" which refers to a single audio channel encoded using 8 bit ISDN mu-law at a sample rate of 8000 Hz.
- **video**: it is used declare video data. The initial subtype is "mpeg" which refers to video coded according to the MPEG standard.

- **application**: it is used to declare discrete data which do not fit to any of the other categories. There are two initial subtypes: "octet-stream" which indicates a body containing arbitrary binary data and "postscript" which indicates a PostScript program.

Composite media types The two composite media types are:

- **multipart**: this type is used to combine in a single body entity one or more different sets of data. An entity of a multipart body is composed by a header area and a body area both optional.
- **message**: this type is used to encapsulate another e-mail message.

The entities of these types are handled by MIME mechanisms.

```
 1:  From:  "John Doe" <john@doe.com>
 2:  Subject:  A multipart message
 3:  Date:  Mon, 22 Aug 2005 14:28:12 +0200
 4:  MIME-Version:  1.0
 5:  Content-Type:  Multipart/Mixed;
 6:   boundary=‘‘Boundary-00=_cTcCDObLSYyHCdw’’
 7:  Message-Id:  <200508221428.12671.john@uaz.ch>
 8:
 9:  --Boundary-00=_cTcCDObLSYyHCdw
10:  Content-Type:  text/plain; charset="us-ascii"
11:  Content-Transfer-Encoding:  7bit
12:
13:  This is a textual part
14:
15:  --Boundary-00=_cTcCDObLSYyHCdw
16:  Content-Type:  image/png; name="an_image.png"
17:  Content-Transfer-Encoding:  base64
18:
19:  iVBORw0KGgoAAAANSUhEUgAACXBIWXMAAAuJAAALiQE3ycutAAAA
20:  [...]
21:  WM0LsRmDaml0N02ZCSb0HP8HAFolni5ND54AAAAASUVORK5CYII=
22:
23:  --Boundary-00=_cTcCDObLSYyHCdw--
```

Example 3.2: An example of a MIME message.

In example 3.2 we can see an example of a multipart MIME message. The heading of the message (from line 1 to 7) includes the definition of a boundary needed to divide the multipart body into its parts. In this example we have

two parts: a textual part (from line 10 to 14) and a image part (from line 16 to 22). The type of the part is defined using the `Content-Type` header in the part's heading (lines 10 and 16). The image part is encoded using the base64 algorithm as specified by the `Content-Transfer-Encoding` header at line 17.

3.1.4 Abstraction of an E-mail Message

In order to describe the structure of MIME messages we need to model the body of the message, which is now more than just a flat us-ascii text.

According to the MIME specification we can state that:

- an e-mail message consists of a heading part and an optional body part.
- the body of a message consists of an optional part of one of the seven types described in Sec. 3.1.3
- a part may have an heading
- a part may have a body which can be of different types according to the subtype of the part (an encoded us-ascii string for discrete media types, a set of parts for multipart media type and a e-mail message for the message media type).

It is useful to model the recursion of multipart and message media type using the composite design pattern described in [6].

Fig. 3.1 shows the model of an e-mail message used in our system.

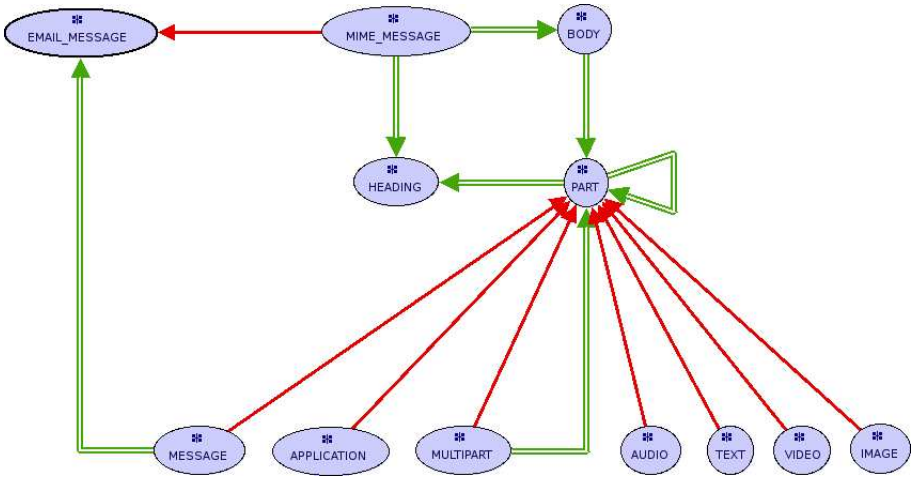

Figure 3.1: Structure of an e-mail message (simplified)

Class ADDRESS

This class is not shown in Fig. 3.1. It is used to represent e-mail addresses in the system. It stores three strings: the e-mail address (`address`), the display name (`display_name`) and the organization (`organization`). For each string a feature is provided in order to modify it. Only the e-mail address is required.

Class HEADING

This class offers an interface to store header fields which are either non-standard or not commonly used.

Feature `add_header` is used to add an header field to the heading of the message, `delete_header` and `delete_all` are used to remove one or more headers from it.

Class `HEADING` offers four access features:

- `encoded`: it is used to have a string representation of the heading
- `contains`: it is used to know if the heading contains a specific field
- `header`: it is used to retrieve the value of an header field
- `match`: it is used to know if the heading contains a specific field with a specific value

Class PART

This class represents a body entity[2]. It defines the following features which are common to all it subclasses:

- `accept`: this feature is used to implement the visitor design pattern. Each subclass should implement this feature by calling its specific feature in the visitor
- `store`: this feature is used to store the part into a file. In order to do that it should perform the decoding operations which are needed using an `ENCODER` object
- `encoded`: it is used to have an encoded version of the part which can be then introduced into the delivery mail system
- `parent`: it is a reference to the parent object. It should not be void if the part is not the root part of the message
- `id`: it is an integer value, unique in the scope of all the MIME entities of the message, which is used to identify a single entity

[2]With the term entity we identify a single MIME part which may appear in the body of a MIME message.

- `subtype`: is a string value which specify the subtype of the entity, for example it may have the value `plain` in a part of type `TEXT`
- `heading`: it is a reference to an `HEADING` object which is used to store the part's header fields

In addition to these services class `PART` defines a set of feature which should be used to set it fields.

As class invariant we require the `subtype` feature to be set.

Subclasses of `PART`

These classes implement the specific behavior of the entity type that they model. We have defined a subclass for each media type defined in Sec. 3.1.3 In the definition of class `MULTIPART` and class `MESSAGE` we implement the recursion defined in the RFCs using the composite design pattern.

Class `BODY`

Instances of this class are used to store the root part (feature `root_part`) of the body of a MIME message.

It provides features to add (feature `add_part`) and remove (`delete_part` and `delete_part_by_id`) parts from the message body. The creation of multipart messages is transparent: when the first part is added to the message body the `root_part` will be set and when a second part is added the `root_part` will become an instance of `MULTIPART` which contains both the old `root_part` and the new added part. In the same way a transition from a multipart to a single part body should be implemented when a multipart body contains only one entity.

Feature `encoded` is used to obtain an encoded version of the body of the message. Feature `part_by_id` is used to retrieve a specific part from a multipart body. Feature `next_id` is used to get a unique integer value which can then be assigned to a new entity.

Class `EMAIL_MESSAGE`

This class represent the highest abstraction of e-mail messages. It existence will be justified in Sec. 3.1.5 when we will explain how our system handles collections of messages.

It contains the standard header fields and the services needed to set and modify them. In addition it defines the following features:

- `finalize`: this feature is used to finalize a message. It performs all the steps needed to make a message ready to be introduce in the mail delivery system like for instance it set the `Date` header field
- `is_finalized`: this feature is true when the message is finalized

- `encoded`: this feature should return the whole message source encoded and ready to be introduced into the mail delivery system
- `short_out`: this feature should return a short string representation of the message, this string can for example contain the sender, the subject and the date of the message. It is ment to give a brief description of the message
- features used to wrap the services of class `HEADING` (`contains`, `header`, `add_header`, `delete_header` and `delete_header_all`)
- features used to wrap the services of class `BODY` (`accept`, `delete_part`, `delete_part_by_id` and `add_part`)
- features containing meta information about the message: `labels`, `priority`, `status` and their modifiers

This class provide a unified interface to build a message: everything can be done using the `EMAIL_MESSAGE` interface according to the "facade" design pattern.

Class `MIME_MESSAGE`

This class is used to represent MIME e-mail messages. It heirs from `EMAIL_MESSAGE` and it contains references to `HEADING` and `BODY` instances.

Class `RFC2822_MESSAGE`

This class (not shown in Fig. 3.1) aims to abstract a RFC2822 message. It hires from `EMAIL_MESSAGE` and it contains a reference to an `HEADING` object an to a string which represent the body of the message. This class can be used for messages which do not need MIME mechanisms.

Class `VISITOR`

This class is not shown in Fig. 3.1. It defines an interface to visit all the parts of a message and to perform tasks according to their type implementing the visitor design pattern.

Class `ENCODER`

This class is not shown in Fig. 3.1. It defines an interface to encode data in a format which can be embedded in an e-mail message (e. g. base46, 7bit, ...). Instances of this class should be used to encode the various media types and to set the `encoded` feature of `PART`'s instances.

3.1.5 Abstraction of an E-mail Message Collection

Class EMAIL_COLLECTION is used to model collections of e-mail messages.

As first attempt it is possible to declare in the class EMAIL_COLLECTION a feature of type ARRAY[MIME_MESSAGE] in order to store the message objects but since our e-mail client should deal with a huge amount of messages an e-mail collection may contain in worst case all the messages in the database and those it will need a lot of memory. In addition it will take a lot time to create all these objects.

The use of the proxy design pattern as described in [6] should provide a solution to these problems.

In order to apply this pattern we define a class EMAIL_MESSAGE and we let MIME_MESSAGE inherit from it. In addition we define EMAIL_PROXY as another subclass of EMAIL_MESSAGE. The EMAIL_PROXY should provide a way to retrieve the message it refers to. When a client ask for a service of an EMAIL_PROXY object the proxy retrieves the MIME_MESSAGE object and forwards the calls to it.

It is desirable that proxy objects store all the data needed to build the output of short_out in this way the GUI elements calling this feature to have a preview of the message do not force the proxy to fetch the whole message.

In chapter 2 we have seen that virtual folders defined by a query provide a good way to let users handle their e-mail messages. In order to support virtual folders we define two subclasses of class EMAIL_COLLECTION:

- MANUAL_EMAIL_COLLECTION: in manual e-mail collections the messages are explicitly added or removed using its features: add_email_message to add messages, remove_email_message and remove_email_message_by_id to remove them.

- VIRTUAL_EMAIL_COLLECTION: the content of virtual e-mail collections is defined by a query stored in feature query and set by set_query. The query is executed by calling the feature perform and the content is then retrieved by calling feature retrieve. Feature count contains the number of items contained in the collection. In this kind of collection it is not possible to add or remove messages explicitly.

Figure 3.2 shows the class structure described above.

3.1.6 Remarks

MIME support

Using this model it is possible to create and display e-mail messages in compliance to the MIME RFCs.

Body entities should store MIME headers in instances of class HEADING, MIME header which, like MIME-Version, should appear in the top heading of the message should be stored directly in the instances of class MIME_MESSAGE.

The various media types defined in the second part of MIME are implemented by the subclasses of class PART in an extensible way. If an update to

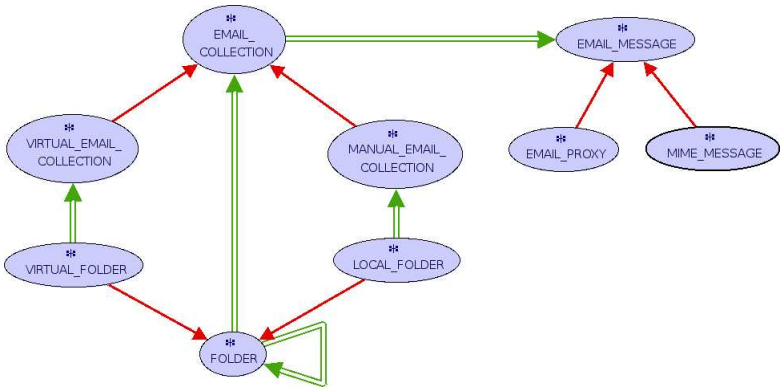

Figure 3.2: This figure shows how folders are implemented in the system. This structure provided support for both manual and virtual folders and implements the proxy design pattern in order to deal efficiently with a lot of messages.

these extensions will define another top-level media type it will be enough to add a subtype to `PART` in order to deal with this entity type. In order to deal with the multitude of media subtypes it is possible to define subclasses of the top-level media type and define a default subclass for media subtypes which are not recognized.

The support for non-ascii message headers as described in RFC2047 [22] can be implemented in the "`encoded`" feature in class `MIME_MESSAGE` and in the same feature in class `HEADING`.

Design patterns

The following design patterns are implemented in this model:

- **Composite**: the composite pattern is used in the classes `MESSAGE` and `MULTIPART` to model the recursion defined in the RFCs
- **Visitor**: this design pattern is used to call features in the polymorphic tree structure modeling the body of the MIME message according to the object type
- **Facade**: this design pattern is implemented by class `EMAIL_MESSAGE` in order to offer a uniform and simple interface to build `EMAIL_MESSAGE` objects
- **Proxy**: to guarantee efficiency we implement the proxy design pattern by creating the class `EMAIL_PROXY` which is a place holder for the other subclasses of `MIME_MESSAGE`. In this way instances of `EMAIL_COLLECTION` should contain in the beginning only proxy objects and create instances of `MIME_MESSAGE` only when it is needed

Programming principles

The set of deferred classes included in this cluster is designed according with the following programming principles described in [21]:

- **Uniform access principle**: services of the classes can be access through a uniform notation, which does not reveal whether they are implemented through storage or through computation.
- **Single choice principle**: the only module who contains an exhaustive list of all the subclasses of `PART` is `VISITOR`.
- **Self-Documentation principle**: Comments are provided to document each feature and each class in order to make modules self-documented.
- **Command-Query Separation principle**

3.2 Protocol Cluster

3.2.1 Foreword

This cluster contains all classes which deal with protocols ("protocol handlers"). In order to provide a simple and clean interface to the rest of the system we divide the protocols handlers into two categories:

- **Send protocols**: protocols of this type are used to introduce e-mail messages into the mail delivery system
- **Receive protocols**: protocols of this type are used to retrieve e-mail messages from the mail delivery system

To implement this division we provide two classes: `SEND_PROTOCOL` and `RECEIVE_PROTOCOL`. Classes implementing send protocols, like SMTP, should inherit from `SEND_PROTOCOL` and receive protocols, like IMAP, should inherit from `RECEIVE_PROTOCOL`. This design choice guarantees extensibility: if a new protocol appears it is possible to integrate it in the system by subtyping one of this two class and implement the state machine which should deal with it.

The next two sections explain the interface of these two classes.

3.2.2 Sending Protocols

The interface of `SEND_PROTOCOL` offers a few services to provide a simple interface to introduce instances of class `EMAIL_MESSAGE` into the mail delivery system.

These services are:

- `set_message` takes as parameter the instance of `EMAIL_MESSAGE` which should be sent.
- `send_mail` tries to send the message

- `message_sent` is a boolean value which indicates if the last call to `send_mail` was successful
- `last_error` contains the error message given by the server if the last call to `send_mail` was not successful

Using this interface it is possible to send a message with a few feature calls, and in a protocol independent way as shown in example 3.3.

```
1:  send_protocol :  SEND_PROTOCOL
2:  a_message :  EMAIL_MESSAGE
3:
4:  -- build and finalize the message
5:
6:  send_protocol.set_message(a_message)
7:  send_protocol.send_mail
8:
9:  if not send_protocol.message_sent then
10:     -- deal with the error
11: end
```

Example 3.3: Send an e-mail message with a few lines of code.

3.2.3 Receiving Protocols

The interface of `RECEIVE_PROTOCOL` offers a set of services which are used to check if new messages are arrived and to fetch them:

- `check_mail`: this service checks if new messages are arrived and stores the number of the new messages in `nof_new_messages`
- `fetch_headings`: this service fetches the headings of the new messages and stores them locally. Then it triggers an update in all the interested GUI elements. The service should store all the data needed to fetch the body of the message.
- `fetch_messages`: this service fetches all the new messages, indexes them, stores them locally and triggers an update in the interested GUI elements.
- `fetch_and_delete_messages`: this service performs the same tasks of `fetch_messages` and in addition it deletes the fetched messages on the server.
- `fetch_message_from_heading`: this service takes as parameter an instance of class `EMAIL_MESSAGE` which should at least contain the heading of a

message and tries to fetch the whole message. If the message is found the local copy and the interested GUI elements should be updated and the fetched message should be indexed; if the message is not found (i. e. it was deleted by an independent protocol session) the local copy of the message should be deleted and a GUI update should be triggered.

- `delete_message_from_heading`: this service takes as parameter an instance of class `EMAIL_MESSAGE` which should at least contain the heading of a message and it tries to delete the corresponding message on the server.

Remarks Note that in order to compute the correct number of new messages (i. e. the value of `nof_new_messages`) the protocol handler should be able to detect what messages it has already fetched but not removed from the server.

The protocol handler should notify the user if some error happens during the session.

Contracts vs. heading Since only the `Date` and `From` headers field are required from [26] it is only possible for us to require these field in the contracts of the `fetch_message_from_heading` and `delete_message_from_heading` services. For the same reason we can ensure in the `fetch_heading` service that only these fields are presents.

3.2.4 Receiving Poller

Instances of this class are used to poll a given account in order to find if new messages are arrived. They have a reference to a instance of `separate RECEIVE_PROTOCOL` and an integer value `timeout` which indicate the frequency of the check. Every timeout a call to the feature `check_mail` of `separate RECEIVE_PROTOCOL` is performed in order to check if new messages arrived. If new messages are detected one of the `fetch` features is called[3] in order to fetch the new messages, index them and store them locally.

3.3 Persistence Cluster

This cluster groups all the classes which are used by the system to deal with persistent storage.

The system need to keep a lot of information persistent, for example:

- Identity information
- Account information
- Virtual folder definition (i. e. a search strings)
- Manual folder information (i. e. the ids of the messages)

[3]The feature is chosen according to the user preferences.

- Index information
- Configuration information
- Additional information about the messages (e. g. labels, priority, ...)

Since the aim of the whole system is to deal with a huge amount of information it is reasonable to store all the data which should be persistent in a database management system (DBMS). This cluster provide an simple interface to store and retrieve information which is independent from the chosen DBMS backend.

The cluster is divided in two main classes: `DB_HANDLER` and `QUERY`.

3.3.1 Class `QUERY`

This class represents the abstraction of a query. It defines only a service named `sql` which provides a SQL query. Subclasses of `QUERY` are defined in order to provide an abstraction for insert, delete, select and update queries. Each of these classes offers an interface to build SQL queries of their specific type. Example 3.4 shows how to build a SELECT query using the API of class `SELECT_QUERY`.

3.3.2 Class `DB_HANDLER`

This class is used to perform the queries and to fetch their results. The connection to the DBMS is done in a transparent way, the services `connect`, `disconnect` and `is_connected` are exported to `NONE`.

The `DB_HANDLER` class offers the following set of services:

- `set_query` takes as parameter an instance of class `QUERY`. It is used to set the query which should then be executed.
- `query` is the query that will be executed
- `execute_query` performs the query and loads the results
- Iteration services (`start`, `after`, `item` and `forth`) are used to iterate over the query results.

3.3.3 Other Handlers

In addition to the query classes and to the `DB_HANDLER` class we define in this cluster five other classes which are used to deal with the persistence of the various components of the system. These classes are:

- `ACCOUNT_HANDLER`
- `CONFIGURATION_HANDLER`
- `IDENTITY_HANDLER`

- `MANUAL_EMAIL_COLLECTION_HANDLER`
- `VIRTUAL_EMAIL_COLLECTION_HANDLER`:

These handlers are the only components that have to know all the subtypes of the component that they should make persistent. For example class `ACCOUNT_HANDLER` which is used to store and retrieve accounting information in order to make `ACCOUNT` objects persistent is the only one which have to know all the subtypes of `ACCOUNT`.

These classes uses an instance of `DB_HANDLER` in order to do their tasks. We require that instances of these types have a not void reference to a `DB_HANDLER` object.

3.3.4 Remarks

The structure of this cluster is shown in Fig. 3.3.

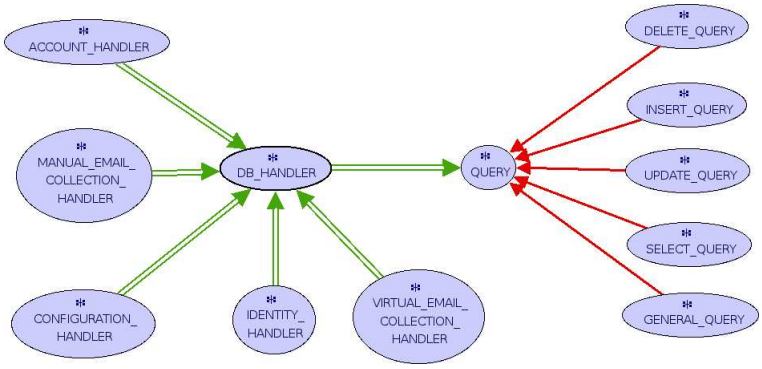

Figure 3.3: Scheme of the persistence cluster

Example 3.4 shows how create a query, execute it and then fetch its results. In the example we build the following SQL query: `SELECT ‘from‘,‘date‘ FROM ‘index_documents‘ WHERE ‘id‘="1981"`. In order to do that we need first to define the table from which we want to select using the `set_db_table` feature (line 4), then we add the select fields “from” and “date” using feature `add_select_field` (lines 5 and 6). At the end we add the condition that the selected rows have to satisfy using the `add_condition` feature (line 7).

3.4 Search Cluster

This cluster contains the definition of two classes which should provide indexing and search capabilities to the system. The next two sections describe this classes.

```
1:  query :  SELECT_QUERY
2:  db_handler :  DB_HANDLER
3:
4:  query.set_db_table("index_documents")
5:  query.add_select_field("from")
6:  query.add_select_field("date")
7:  query.add_condition("`id`=%"1981%"")
8:
9:  db_handler.set_query(query)
10: db_handler.execute_query
11:
12: from
13:     db_handler.start
14: until
15:     db_handler.after
16: loop
17:     -- consume db_handler.item
18:     db_handler.forth
19: end
```

Example 3.4: This code shows how to use the `QUERY` and `DB_HANDLER` API in order to build a query, execute it and fetch its results.

3.4.1 Class `INDEX`

The `INDEX` class is used to add documents to the index and to retrieve information about the documents stored into it.

Message identifiers

Message identifiers are integer values used to identify messages, for this reason they should be unique. The feature `free_id` returns the next free identifier value which should be used. Since the index is persistent and those survives a program execution the `free_id` should be initialized to a consistent value each time an `INDEX` object is created.

Document information

The feature `document_query` is used to retrieve the basic information about a message given an integer message identifier. This message id should be set using the `set_id` feature. The result of the query operation can be found in the `document` feature which is a tuple containing the following data:

- the `From` header field

- the `Subject` header field
- the `Date` header field
- the size of the message in bytes
- the filename in which the message is stored

This data can be used by an `EMAIL_PROXY` to implement the `short_out` feature or to fetch the whole message.

`add` features

The `INDEX` class provides a set of services which should be used to insert items into the various indexes. The structure of the index will be described in Sec. 4.1 for now it is enough to know that, in general, an index item associates a "key" value (either a string or a number) to a set of messages. For this reason the add features take as parameters a key value and the integer message id of the message that it has to associate with the key.

An exception is the `add_document_item` which need the same data fetched by `documents_query`.

Length constraints

Using the API of this class it is possible to set the minimal and maximal length of the indexed items (features `set_minimal_length` and `set_maximal_length`) in order to avoid the indexing of words which are either too short or too long.

3.4.2 Class `QUERY_MAKER`

The purpose of this class is to offer an interface to execute queries and fetch their results. The queries are strings which contain productions of the query language defined in Sec. 4.2.

Queries string are passed to the `QUERY_MAKER` using the `set_query` feature. The execution of the query is then triggered by a call to `perform`.

After performing a query it is possible to get the cardinality of the result set using feature `count`. The `fetch_proxies` feature is used to fetch the proxies which refer to the messages that satisfy the query criteria. These proxies can be then retrieved using the `proxies` feature.

Example 3.5 shows how a query can be performed using this API.

3.5 Configuration Cluster

The configuration cluster contains all the classes which deal with configuration issues.

```
query :  STRING
query_maker :  QUERY_MAKER
proxies :  ARRAY[EMAIL_MESSAGE]

query := "(-b eiffel)"
query_maker.set_query (query)
query_maker.perform

if query_maker.count < some_value then
    query_maker.fetch_proxies
    proxies := query_maker.proxies
end
```

Example 3.5: Execution of a query using the `QUERY_MAKER` API.

3.5.1 Class `ACCOUNT`

Class `ACCOUNT` is used to store the information needed to log into an account. This information is fetched from the database using an `ACCOUNT_HANDLER`. Since different account types may need different data to log into them and since one of the purpose of this system is to be extensible we define a subclass of `ACCOUNT` for each account type (i. e. `SMTP_ACCOUNT`, `IMAP_ACCOUNT` and `POP_ACCOUNT`).

Instances of class `ACCOUNT` are identified by an integer value stored in the `id` feature. In order to identify them in a more "user friendly" way we provide a field `name` which contain a display name for the class instance.

The hostname and the port number of the server which hosts the account are stored in features `hostname` and `port`.

The user credentials needed to log into the account are stored in the `username` and `password` fields.

For each of these features class `ACCOUNT` provides a modifier feature which should be used to set its content.

As class invariant we require to have a not empty `hostname`, a valid port number ($0 <$ `port` ≤ 65535) and a positive `id` value.

3.5.2 Class `IDENTITY`

Another interesting feature of modern e-mail clients is the possibility to have many different users with different identities.

In order to support it we provide the class `IDENTITY` which is ment to represent a user.

Instances of this class are identified by an integer value stored in the `id` feature and have a display name stored in `name`. In addition to that they have

a feature called `password` which is the password that a user should provide in order to use this identity.

User data are stored in an instance of class `ADDRESS`. More data can be added using inheritance mechanisms for example if we want to add PGP support to the client we can define a class `PGP_IDENTITY` which heirs from `IDENTITY` and add, for example, a `keyid` feature which represent the id of the PGP user key.

A set of features is provided to set the fields described above.

For this class we require that `address` is set, the `name` is set and not empty and that it has a positive identifier `id`.

Chapter 4

Definition of the Index Structure and of the Search Algorithm

4.1 Index Structure

In order to perform fast search across a huge amount of data we need to exploit in some way the structure of the data itself and reflect this structure in the index.

In this work the indexes are generally composed by a "key" field which is a string and a sequence of references (message ids) to all the messages which contains somewhere this string.

In the case of an e-mail message we can recognize many sources of interesting data: each header field, being standard or not, aims to provide in some way information about the message.

To build an index we need to select the most meaningful sources of information and find a way to map them in an indexable form[1]. For example let assume we want to index the following `From` header field: `From: "John Doe" <john@doe.com>, "Jane Doe" <jane@doe.com>`. If we consider as a word only character sequences which are composed of alphanumeric characters we will not consider the e-mail addresses as being a words and those they will not be indexed even if they are an important source of information.

We decide to index the following parts of the messages:

- Body parts which are either "`text/plain`" or "`text/html`"
- Senders of the message: `From` and `Return-Path` header fields
- Recipients of the message: `To`, `Cc`, `Bcc` and `Envelope-To` header fields

[1] This process was called "header trimming" in Sec. 2.2.

- Subject of the message
- Date of the message
- Size of the message

In general we consider as a word every sequence of alphanumeric characters: for example the string "abcd-efg" will be splitted into the words "abcd" and "efg". For the senders and recipients header fields we extend the notion of word in order to index e-mail addresses.

The index structure is composed by six subindexes each of them contains one of the six parts described above. The body, senders, recipients and subject indexes are composed by a set of "keys" and for each key they store the references to the messages which contains it.

The date index have the same structure of the four indexes described above but the key of this index is an integer value which specify a day[2]. For each key the index stores the references to the messages sent during that day.

The size of the messages is stored in a index with a different structure, this index is called `index_documents` and in addition to the size it contains the information needed to retrieve the whole message[3].

4.2 Query Language

4.2.1 Foreword

In the previous section we have described the structure of the index. Now we define a way to retrieve information from it.

In order to do that we define a query language which exploits the structure of the index. Productions of this language are executed by an instance of class `QUERY_MAKER`.

4.2.2 Syntax

The definition 4.1 states the syntax of the query language using the EBNF format.

A production of the language is called "query" and is composed by one or more subqueries enclosed by round brackets.

A subquery is composed by a list of query items ("qitem_list"). These query items are either numeric ("pnitem" and "nnitem") or string ("psitem" and "nsitem").

Numeric query items are defined by a string and two numbers separated by a dot. String query items are defined by one of the "qtype" strings and another alphanumeric string ("qstr").

[2]This integer value is build by concatenating the year, the month and the day of the date, e. g. 21. September 1981 becomes 19810921.

[3]A description of an implementation of this subindex is described in chapter 5.

An example of a production of this language can be the following string: "`(-d 20050401.20051007 -p foo)(-b bar -ns doh)`".

```
 1:  query ::= subquery {subquery}
 2:
 3:  subquery ::= '(' qitem_list ')'
 4:
 5:  qitem_list ::= pnitem {' ' qitem_list}
 6:      | nnitem {' ' qitem_list}
 7:      | psitem {' ' qitem_list}
 8:      | nsitem {' ' qitem_list}
 9:
10:  pnitem ::= pntype ' ' number '.'  number
11:  nnitem ::= nntype ' ' number '.'  number
12:
13:  psitem ::= pstype ' ' qstr
14:  nsitem ::= nstype ' ' qstr
15:
16:  pntype ::= '-d' | '-z'
17:  nntype ::= '-nd' | '-nz'
18:  pstype ::= '-p' | '-b' | '-s' | '-f' | '-r'
19:  nstype ::= '-np' | '-nb' | '-ns' | '-nf' | '-nr'
20:
21:  qstr ::= <a-zA-Z0-9@> {<a-zA-Z0-9@>}
22:  number ::= <0-9> {<0-9>}
```

Definition 4.1: EBNF syntax of the query language.

4.2.3 Semantic

Each production of the query language aims to define a set of messages matching some criteria. In this section we describe the semantic of each criteria offered by the query language. In order to do that we apply a top down approach.

Query Line 1 of Def. 4.1 states that a query is composed by one of more subqueries. Each subquery defines a set of messages. The union of these sets is the result of the query.

Subquery Line 3 of Def. 4.1 states that a subquery is list of query items. Items can be of four types:

- positive numeric item (`pnitem`)

- negative numeric item (`nnitem`)
- positive string item (`psitem`)
- negative string item (`nsitem`)

Each item defines a set of messages.

In order to evaluate a subquery we first evaluate each subquery item. Then we create two sets by intersecting all the positive items (`pnitem` and `psitem`) in one and by merging all the negative items (`nnitem` and `nsitem`) in the other.

The result of the subquery is computing by subtracting the negative item set from the positive item set.

Items evaluation Each item is defined by a type and either two `number`s or a `qstr`. We now define the set of messages retrieved by each item type:

- `-p`, `-np`: these types of items defines a prefix query. The set of messages that they retrieve is the union of all the sets in the body index identified by a key that have the `qstr` value as prefix. For example the item "`-p foo`" will match both the key "foo" and "foobar".
- `-b`, `-nb`: these types of items define an exact body query. They retrieve the set of messages identified by the `qstr` value in the body index.
- `-s`, `-ns`: these types of items define a subject query. They retrieve the set of messages identified by the `qstr` value in the subject index.
- `-f`, `-nf`: these types of items define a from query. They retrieve the set of messages identified by the `qstr` value in the from index.
- `-r`, `-nr`: these types of items define a recipient query. They retrieve the set of messages identified by the `qstr` value in the recipient index.
- `-d`, `-nd`: these types of items define a date query. They retrieve the set of messages from the date index which have a key between the two `number` values (included).
- `-z`, `-nz`: these types of items define a size query. They retrieve the set of messages from the documents index which have a size value between the two `number` values (included).

4.3 Search Algorithm

In this section we describe the search algorithm used by the prototype to evaluate the queries.

This algorithm is implemented in the `perform` feature of `QUERY_MAKER_IMPL`. Its purpose is to map productions of the query language to a set of SQL queries which are then performed by a `DB_HANDLER`.

4.3.1 Query Execution

In a normal query execution the QUERY_MAKER_IMPL receive a production of the query language, which has no relation both with the previous nor with the next production it has to execute.

In order to execute the query it has first to evaluate all the subqueries and then perform an union on their result sets.

The evaluation of a subquery requires the evaluation of each item and the creation of the two subquery result sets: the positive result set is composed by the intersection of all positive subquery items and the negative result set is composed by the union of all negative subquery items.

The subquery result is then built by subtracting the negative set from the positive set.

Definition 4.2 states the pseudocode of the algorithm.

```
 1:  Res := ∅
 2:  FOREACH subquery sq ∈ query DO
 3:      Pos := ∅
 4:      Neg := ∅
 5:      FOREACH item i ∈ sq DO
 6:          evaluate i
 7:          IF i IS positive THEN
 8:              IF Pos ≡ ∅ THEN
 9:                  Pos := result i
10:              ELSE
11:                  Pos := Pos ∩ result i
12:              END
13:          ELSE
14:              Neg := Neg ∪ result i
15:          END
16:      END
17:      Res := Res ∪ (Pos \ Neg)
18:  END
```

Definition 4.2: Evaluation of a query (pseudocode)

As we have seen in the beginning of this section the QUERY_MAKER has to map a production of the query language in a sequence of SQL queries. To implement the evaluation algorithm 4.2 we need to have a mapping of the following operations to SQL statements:

- the evaluation of a simple item (used at line 6 of the algorithm)
- union of two results (used at lines 14 and 17)

- intersection of two results (at line 11)
- subtraction between two sets (at line 17)

Evaluation of a simple item

The evaluation of a simple item can be easily mapped to a `SELECT` SQL statement of the form:
`SELECT DISTINCT <field> FROM <index table> WHERE ‘key‘="<value>"`
An exception is made for the prefix items. These items need pattern matching in order to find out if a key has some value as prefix. The mapping for them is:
`SELECT DISTINCT <field> FROM <table> WHERE ‘key‘ LIKE("<value>%")`
Queries of this type very efficient (execution time around 0.2 seconds).

Union of two results

The union operation is already part of SQL. If we want to merge tables `q1` and `q2` we can perform the following statement:
`(SELECT <field> FROM q1) UNION (SELECT <field> FROM q2)`
The result contains no duplicate entries: rows that appear both in `q1` and `q2` appear only once in the result. Query of these type are efficient, they usually have an execution time smaller than 0.5 seconds.

Intersection of two results

The intersection of two result sets can be mapped to another `SELECT` SQL statement. Let's assume the two sets are stored in tables `q1` and `q2` and these tables contains some message id field to identify the messages called "`mid`". The intersection of these two table can be done with the following statement:
`SELECT DISTINCT q1.mid FROM q1,q2 WHERE q1.mid = q2.mid`
The execution of this type of queries is not too efficient for huge sets, in order to achieve a good performance we need to avoid them if it is possible or to reduce the size of the two sets.

Subtraction of two results

The subtraction of two result sets is done by removing from the first set everything which belongs to the second set. This operation can be mapped to a `DELETE` statement. Let's assume that we want to subtract the content of table `q2` from table `q1` and these tables contains some message id field to identify the messages called "`mid`". The subtraction is then done whit the following statement:
`DELETE q1 FROM q1,q2 WHERE q1.mid = q2.mid`

4.3.2 Result Caching

An improvement to the performance of the query evaluation is the caching of the partial results of the queries.

The cache is used to map a part of a query to the database table which contains its result set without performing any database query.

Each time a query item is evaluated a database table which contains the set of messages identified by the item is created and an entry which map the item to the name of the table is inserted in the cache.

The next time that the `QUERY_MAKER` needs to evaluate this item it can directly use the result stored in the database table.

The evaluation of the query items is not a slow operation and the improvements provided by caching are of a small impact on the performance of the evaluation. But since the cache is never emptied it contains the results of the previous queries and this will provide a huge speed up when the queries are related. These improvements are described the next section.

4.3.3 As-You-Type Query Execution

Let's assume we want to provide to the users a text field in which they can simply type some word and get back the set of messages which contains these words without requiring to the users the knowledge of the query language.

In addition we want the result to be shown to the users as they type these words.

In order to do that we trigger a query each time the user types something, we call this process an "as-you-type query". Since looking for every message that contains a string of a tiny length is meaningless we start querying the index only if the query string have a length bigger than three characters.

Example

Let's assume the user want to retrieve the set of messages which contains the word "eiffel" and the word "studio". By typing these words it will generate the sequence of queries shown in example 4.1. We use the prefix query item since we assume that what the user has typed until now is not the final query.

```
1:  (-p eiff)
2:  (-p eiffe)
3:  (-p eiffel)
4:  (-p eiffel -p stud)
5:  (-p eiffel -p studi)
6:  (-p eiffel -p studio)
```

Example 4.1: Queries generated by the as-you-type query "eiffel studio"

The first three queries of the sequence shown in example 4.1 need only a `SELECT` SQL statement which is, as said in 4.3.1, an efficient operation.

To execute the fourth query we need first to fetch the result of the item `-p eiffel` and of `-p stud`, then perform the intersection of these results. The result of `-p eiffel` needs not to be computed since it is cached. The evaluation of the second item is fast too. But the intersection of the two huge result sets is a slow operation.

We have here to notice that the result of the fifth query is a subset of the result of the previous one. By adding the result of `(-p eiffel -p stud)` to the cache we can gain a lot of time in the execution of the next queries.

So in order to execute the fifth statement the engine recognize that the result is a subset of the previous one which is cached. We then fetch the result of `-p sudi` and then intersect it with the cached result of `(-p eiffel -p stud)`. The intersection is faster than the last one because the two sets have a smaller cardinality.

The same operations are done to compute the last query.

Query	Items
`(-p eiffel)`	13386
`(-p stud)`	6591
`(-p studi)`	3405
`(-p eiffel -p stud)`	572
`(-p eiffel -p studi)`	210
`(-p eiffel -p studio)`	145

Table 4.1: Cardinality of the retrieved sets

In order to quantify what said before let's look at table 4.1 which shows the cardinality of the retrieved sets of some queries. We see that `-p eiffel` retrieves 13386 items which should, in the fourth line of example 4.1, be intersected with the 6591 items of `-p stud`. If we do not consider the cached items, in the fifth line of example 4.1, we have to intersect a set with 13386 items with a set of 3405 items. The two set are huge and the operation is slow. But if we consider caching we need to intersect a set with 572 items with the one of 3405 and this operation is much faster than the one without caching.

Chapter 5

Prototype Description

5.1 Purpose of the Prototype

The purpose of this project is, as its title says, to design an innovative e-mail client. The design of this system is composed by a set of deferred classes divided into five clusters.

In order to provide a way to implement these deferred classes and to show that the search algorithm proposed in Sec. 4.3 is able satisfy the needs of a "power user" we implement a prototype.

This prototype does not aim to be either a complete e-mail client nor to be easy to use.

The next two sections describe the implementation choices and the features implemented in the prototype. Sec. 5.4 contains an evaluation of it and Sec. 5.5 explains how to install it on a Debian GNU/Linux system [4].

5.2 Implementation

This prototype was developed on a Debian GNU/Linux system and uses a MySQL DBMS backend [23].

Notice that this choices are not mandatories. The design itself is platform independent and do not require any specific DBMS backend (e. g. another implementation may use an Oracle DBMS backend and can be implemented to run on other operating systems).

5.2.1 Libraries

To implement the prototype we make use of the following set of libraries:

- EiffelBase: for all the structures of the Eiffel language
- EiffelVision 2: for the GUI

- EiffelStore: to connect to the DBMS backend
- e-Posix [5]: for the base64 encoder, for its e-mail message parser and for the networking
- Gobo [14]: because it was required by e-Posix

5.2.2 Database Scheme

In this section we describe the scheme of the database used to store the data that need to be persistent. We define a table for each subindex (see Sec. 4.1), a table to store the virtual collections and a table to store the SMTP accounts.

The SQL statements needed to create the database tables described here are stored in file `database-scheme.sql`.

Subindexes Tables

The body, from, recipients and subject subindexes are used to associate a string to a set of messages identifiers. Since MySQL support only simple types for the table's fields it was not possible to declare these tables as being composed by a string field and an array field. For this reason we declare them as being composed by string field called `key` for the key string and by an integer field for one single message id called `mid` as shown in table 5.1. During the indexing process we add to the table a pair (`key`,`mid`) for each message which should be associated with this key value. In this way a key value may reference many table rows and those we can not define it as a primary key. The only possible solution is to define a primary key on both fields.

Field	Type	Null	Default
key	varchar(50)	No	
mid	int(11)	No	0

Table 5.1: Structure of the common subindexes tables

The date index table have a similar structure as the other subindexes. The only difference is that the "key" field is an integer value. Table 5.2 shows the structure of this table.

Field	Type	Null	Default
key	int(11)	No	
mid	int(11)	No	0

Table 5.2: Structure of table index_date

The documents index should store much more information and those it has a different table structure (see table 5.3).

An integer field `id` is used to identify a single message and it is defined as being the primary key of the table. Since in Maildir format one file contains only

one message we declare the `filename` field as being unique. To make possible the size query we store in this index even the message size in bytes (field `size`).

This index stores the `From`, `Subject` and `Date` header fields too. We store them in this index because they are used to implement the `short_out` feature in the class `EMAIL_PROXY_IMPL`. In this way we can set this data in the proxies when we create them and a call to `short_out` does not force the proxy to perform any retrieve operation.

Field	Type	Null	Default
id	int(11)	No	0
from	varchar(255)	No	
subject	varchar(255)	No	
date	varchar(50)	No	
size	int(11)	No	0
filename	varchar(255)	No	

Table 5.3: Structure of table index_documents

Other Tables

In addition to the subindex tables we define two other tables in order to store the SMTP accounting information (see table 5.4) and the queries which defines virtual collections that the users decided to store (see table 5.5).

These two tables store the information needed to build instances of classes `SMTP_ACCOUNT_IMPL` and `VIRTUAL_EMAIL_COLLECTION_IMPL`.

Field	Type	Null	Default
id	int(11)	No	0
name	varchar(50)	No	
hostname	varchar(100)	No	
port	int(11)	No	0
username	varchar(100)	No	
password	varchar(100)	No	

Table 5.4: Structure of table smtp_account

Field	Type	Null	Default
id	int(11)	No	0
name	varchar(100)	No	
query	text	No	

Table 5.5: Structure of table virtual_collections

5.2.3 New Classes

The prototype was developed by creating new classes which hires and implement the deferred features of the classes described in the previous chapters.

Naming

We create for each cluster its correspondent implementation cluster. The names of these clusters are the one of the original cluster with a suffix "_IMPL". The same naming convention was used for the naming of the implementation classes.

Adapters

In order to avoid the dependency of the design from any library used by the prototype we have developed two classes which are used to make our system compatible with the libraries' object model. These classes are `MAIL_CONVERTER` and `INDEXER`.

Class `MAIL_CONVERTER` This class converts instances of class `EPX_MIME_PART`[1] to `EMAIL_MESSAGE` instances.

Class `INDEXER` This class is used to index messages either using our object model or using the e-Posix model. Two features (`index_eposix_message` and `index_email_message`) are provided in order to deal with these two different models. They extract the data that should be indexed (see Sec. 4.1) from either a `EPX_MIME_PART` or `EMAIL_MESSAGE` object and perform the needed calls to the `add`-features of an `INDEX_IMPL` instance in order to index it.

5.2.4 Remarks

During the development of the prototype we have found three errors in the used libraries. The next sections explain how we solve them.

Modification of `PAPI_UNISTD`

The original version, the one shipped with e-Posix version 2.2, of this class does not compile. We get the following compiler error:

```
Class: PAPI_UNISTD
Feature: posix_getgroups
Left-hand type: POINTER
Right-hand type: NONE
Line: 220
        valid_size: gidsetsize >= 0
->      valid_grouplist: gidsetsize > 0 implies grouplist /= Void
      external "C"
```

[1] This class is used by e-Posix to abstract e-mail messages.

```
------------------------------------------------------------------
Degree: 3 Processed: 1 To go: 11 Total: 12
```

In order to fix this error we have modified this line from
`valid_grouplist: gidsetsize > 0 implies grouplist /= Void`
to
`valid_grouplist: gidsetsize > 0 implies grouplist /= default_pointer`

File `papi_unistd.e.patch` provide a patch that corrects this error.

Modification of `EPX_MIME_SCANNER`

In the prototype we use `EPX_MIME_PARSER` in order to get `EPX_MIME_PART` objects from the messages source. The parser discards for some reason the `To` and `Bcc` header fields. Since we want to index these two fields we modify the scanner (class `EPX_MIME_SCANNER`) in a way that it will recognize the `Bcc` and `To` headers as general unstructured header fields. This kind of field are then considered by the parser.

In order to perform this modification we need to modify only four lines of code in the `EPX_MIME_SCANNER` class.

The file `epx_mime_scanner.e.patch` contains a patch for this class.

Modification of `odbc.c`

The original file `odbc.c` revision 1.27 shipped with EiffelStore does not define the constant `TRUE` but uses it, generating in this way compile time errors. We add to the code in `odbc.c` the definition of this constant.

The file `odbc.c.patch` contains a patch for this file.

5.3 Features

5.3.1 Indexing

The prototype implements an indexing feature. The set of messages that the user want to index needs to be stored into a given directory in Maildir format. During this process every file of this directory is parsed by an instance of `EPX_MIME_PARSER` and an integer value (message id) is associated to the message. The parser builds `EPX_MIME_PART` objects which are then passed to an `INDEXER` object which performs a set of calls to the `add`-features of an `INDEX` object in order to index the message. These features perform then, through a `DB_HANDLER_IMPL`, the insert SQL queries in oder to store the index data into the MySQL database tables. Figure 5.1 shows the information flow of this process.

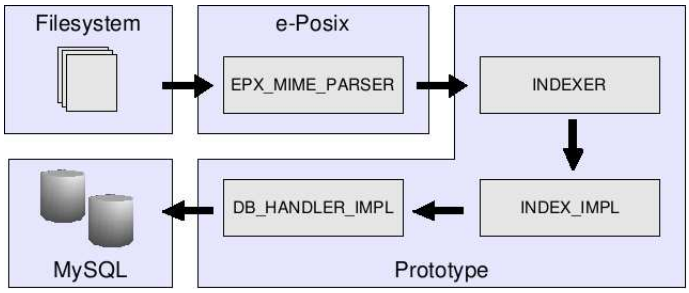

Figure 5.1: Information flow of the indexing process

5.3.2 Advanced Search

If the "query" option is chosen a prompt is shown. At this prompt users can type a production of the query language (see Sec. 4.2) to perform searches.

In order to deal with the queries and with their results the prototype uses virtual e-mail collections. When a query is submitted by the user a `VIRTUAL_EMAIL_COLLECTION_IMPL` object is created and its feature `perform` its executed. The execution time of this call is measured and then shown to the user together with the number of retrieved items.

If the number of the retrieved messages is not too big (i. e. smaller than 1000) the proxies objects for these messages are created.

After performing a query and after fetching the proxies users can see the source of a fetched message by typing `show` followed by the number of the message they want to see (e. g. `show 1` displays the first message of the virtual collection).

5.3.3 Graphical Query

The prototype provides a simple GUI in order to query the index without requiring the knowledge of the query language.

This GUI implements the "As-You-Type" query execution described in 4.3.3. Users can type some words in part 1 of the GUI (see Fig. 5.2) and each keystroke triggers a query execution. If the number of retrieved items is not too big (i. e. smaller than 500) the proxies object for these messages are created and they are displayed in part 2. We use the `short_out` feature to display the proxies, in this way we do not force them to retrieve the whole message.

If one of the items in part 2 is selected the `encoded` feature of the corresponding `EMAIL_PROXY_IMPL` object is called in order to get the whole message source. This operation force the proxy to retrieve the whole message. An adapted version of the source of the message is then shown in part 3.

A virtual e-mail collection is defined by a query. Using the "save" button in part 1 it is possible to create a persistent virtual e-mail collection defined by the current query string. The stored queries are then listed in part 4. If one of

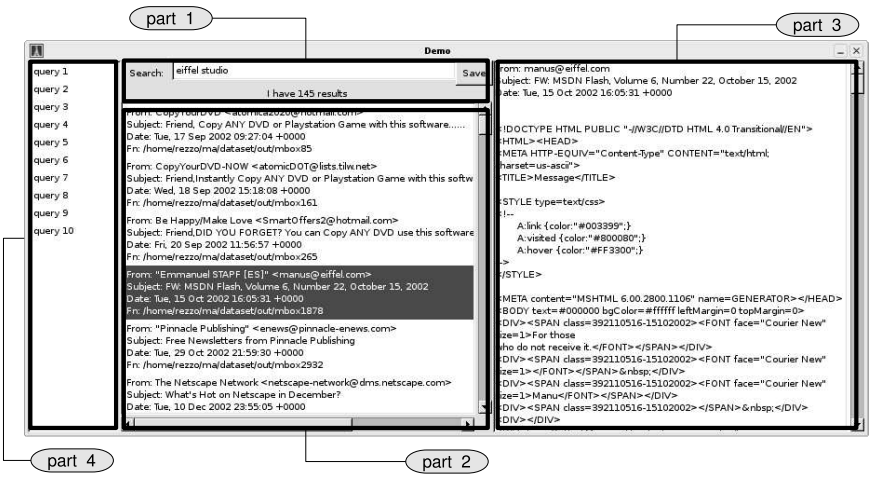

Figure 5.2: Screenshot of the prototype user interface

these items is selected the words composing the corresponding query are shown in part 1 and the content of the virtual collection is shown in part 2.

5.3.4 Composing and Sending Messages

If the "send" option is chosen the prototype ask to the user the data needed to compose the mail as shown below.

```
From address: john@doe.com
From display name: John Doe
Destination address: jane@doe.com
Subject: This is a test
Body (end with a line which contains only a dot '.')
This is a test

bye
John
.
```

After collecting this data the prototype builds a `MIME_MESSAGE_IMPL` object and displays a preview of the source of the message[2]. At the end the user should decide if the message should be sent or discarded as shown below:

```
PREVIEW:

Date: 07 Sep 2005 13:03:41 +0200 (CEST)
```

[2]The source is generated by calling the `encoded` feature of the `MIME_MESSAGE_IMPL` object.

```
From: "John Doe" <john@doe.com>
To: jane@doe.com
Message-Id: <2005SEP713341@myclient>
Subject: This is a test
MIME-Version: 1.0
Content-type: text/plain; charset=us-ascii;

This is a test

bye
John

Send? [y/n]: y
```

If the user chooses to send the message an SMTP session is started and the message is sent. The whole communication between the client and the server is dumped as shown below:

```
220 mail.doe.com ESMTP Postfix (Debian/GNU)
-> EHLO localhost
250-mail.doe.com
250-PIPELINING
250-SIZE 10240000
250-VRFY
250-ETRN
250-STARTTLS
250-AUTH LOGIN PLAIN DIGEST-MD5 CRAM-MD5
250-AUTH=LOGIN PLAIN DIGEST-MD5 CRAM-MD5
250 8BITMIME
-> AUTH LOGIN
334 VXNlcm5hbWU6
-> username base64 encoded
334 UGFzc3dvcmQ6
-> password base64 encoded
235 Authentication successful
-> mail from
250 Ok
-> rcpt to
250 Ok
-> DATA
354 End data with <CR><LF>.<CR><LF>
250 Ok: queued as D1DEAC3407E
-> QUIT
221 Bye
sent
```

If the message was successfully sent, it is stored locally and it is indexed.

5.4 Prototype Evaluation

5.4.1 Test Data

In order to test the prototype we use a different data set than the one described on page 3. This new data should stress the prototype much more than the other one.

The test data consists of 249883 messages. 235235 of them were correctly parsed and indexed. The total size of the indexed messages is 1117 MB and the mean size of a message is 4.9 KB.

The size of the messages has very small variance ($\sigma^2 = 3.4 \times 10^{-5}$), there are only 249 messages with a size bigger than 20 KB.

5.4.2 Indexing Performance

This process is very slow. We identify two bottlenecks which are discussed in the next paragraphs.

Parser performance

The parser is very slow, it takes much time to parse big messages. Even if non textual MIME parts, which are usually bigger than the textual ones, were not indexed they still are parsed. This problem can be solved by writing a more efficient parser which do not rely on the e-Posix library.

Insert operations in table "index_body"

The database table "index_body" tends have many rows. Since we define a primary key on this table the MySQL DBMS have to check at each insert that the inserted row has no duplicates. If we drop the primary key on this table the INSERT queries became very fast but we will loose the table indexes and the SELECT queries on this table will became very slow. This problem is not really a problem of our system, another DBMS may have a more efficient insert operation on big tables.

Remark

Even if the indexing of big dataset is a slow operation it should be done only once. After creating the first index the messages are indexed in an incremental way as they arrive and leave without having an impact on the system performance.

5.4.3 Index Size

Table 5.6 shows the sizes of the various subindex database tables.

We note here that a big contribution on the index size is done by table "index_body" which have a very high number of rows. The fact that this table

Subindex table	Items	Size
index_body	24'315'611	880.5 MB
index_date	234'805	5.3 MB
index_documents	235'235	42.0 MB
index_from	396'232	13.1 MB
index_recipients	45'587	1.6 MB
index_subject	987'387	35.0 MB
Total	**26'214'857**	**977.5 MB**

Table 5.6: Size of the subindex database tables

should have a primary key on both field `key` and `mid` make the MySQL index on that table bigger.

Anyway we assume that the size of the index is not a big issue for "power users" because they already need a lot of space in oder to store their messages.

5.4.4 Advanced Search Performance

In this section we want to give an idea of the performance of the search engine. In order to do that we have executed a set of example queries which show many aspects of the search algorithm.

The reported time is the one needed by the `QUERY_MAKER` to execute the query.

Simple query As first query we execute a simple subject query: we look for every message which has the string "ethz" in the subject header field (search string: `(-s ethz)`).

The search engine retrieves 284 messages and the query execution took 0.31 seconds.

Simple query with date constraint As second example we look for all the messages received between the 1^{st} January 2004 and the 1^{st} February 2004 which contains in the body the string "ethz" (search string: `(-b ethz -d 20040101.20040201)`).

The search engine retrieves 209 messages and the query execution took 0.49 seconds.

Here the `QUERY_MAKER` have to intersect the items retrieved by `-b ethz` (5399 items) with those retrieved by `-d 20040101.20040201` (16335 items).

Union of two subqueries The next query that we want to show needs an union operation between the sets retrieved by two subqueries. We look for all the messages which contain the word "eiffel" and "studio" or the words "eiffel" and "store" (search string: `(-b eiffel -b studio)(-b eiffel -b store)`).

The search engine retrieves 482 messages and the query execution took 1.51 seconds.

Complex query In order to show the flexibility of the query language we want to look for all the messages which either contain the word "eiffel" and the word "studio" but not the word "store" or the ones which contain the word "eiffel" and the word "store" but not the word "studio" (search string: `(-b eiffel -b studio -nb store) (-b eiffel -b store -nb studio)`).

The search engine retrieves 479 messages and the query execution took 0.74 seconds[3].

5.4.5 As-You-Type Query Performance

In order to show the performance of the "As-You-Type" queries we measure the time needed to perform each query triggered by a user keystroke. The time values reported here include the time needed to display the results of the query (i. e. it is the execution time of the agent `text_agent` in class `MAIN_WINDOW`) and those it should be very similar to the one experimented by the users.

"eiffel studio" Query

Search field content	Execution time (s)	Retrieved items
e	0	-
ei	0	-
eif	0	-
eiff	0.32	13432
eiffe	0.22	13389
eiffel	0.19	13386
eiffel s	0	13386
eiffel st	0	13386
eiffel stu	0	13386
eiffel stud	0.28	572
eiffel studi	0.22	210
eiffel studio	0.16	145

Table 5.7: "eiffel studio" query execution times

Table 5.7 shows the executions time of the queries triggered by the user's keystrokes by typing the words "eiffel studio".

The first three keystrokes do not trigger any query. As soon as the user type the fourth character the query `(-p eiff)` is performed. The same is done for the fifth and sixth keystroke. Then no query is triggered until the fourth character of the second word is typed, the query triggered here is the slowest one because an intersection operation between two big set is needed (`-p eiffel` contains 13386 items and `-p stud` 6591). The next query takes advantage of the cache. Its execution is faster than the last one even if the number of retrieved

[3] As "casting out nines" check the query `(-b eiffel -b studio -b store)` retrieves 3 messages.

messages is smaller than 500 and those the GUI has to display them. The same happens for the last keystroke.

Notice how, thanks to the use of caching, the execution time decreases in the last three queries.

5.5 Setup Instructions

Package Overview

The prototype is shipped in the file `ma-final.tar.gz`. The content of this archive has the following structure:

- `bin`: this directory contains the binary of the prototype
- `conf`: this directory contains the configuration file
- `patches`: this directory contains the patches described in Sec. 5.2.4
- `sql`: this directory contains the SQL script that creates the database and the iODBC configuration files
- `src`: this directory contains the source code of the prototype
- `store`: this directory is used by the prototype to store the messages

The next sections show the installation steps which are needed to install the system.

Install the System Packages

Some of the required software is already included in the OS distribution. Type the following command as root in order to install them:

```
# apt-get install libmyodbc libiodbc2 patch mysql-server-4.1 \
    libgtk2.0-dev unixodbc-dev libxtst-dev gcc make
```

Extract the Prototype Package

Copy the prototype package into the target directory (i. e. ∼/) and extract it using the following commands:

```
$ cd ~
$ tar xzf ma-final.tar.gz
```

Configure MySQL

Execute the SQL script contained in file ~/ma/sql/script.sql in order to create the database and to add a new user to the MySQL system.

```
$ mysql -u root -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 5 to server version: 4.1.11-Debian_4-log

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> source ma/sql/script.sql
mysql> exit
```

Configure iODBC

Files ~/ma/sql/odbc.ini and ~/ma/sql/odbcinst.ini contain the configuration for iODBC and should be copied to ~/.odbc.ini and ~/.odbcinst.ini.

```
$ cp odbc.ini ~/.odbc.ini
$ cp odbcinst.ini ~/.odbcinst.ini
```

Install EiffelStudio

Get EiffelStudio (i. e. file Eiffel56.tgz) and extract it in the target directory (i. e. ~/).

```
$ tar xzf Eiffel56.tgz
$ export ISE_EIFFEL=~/Eiffel56
$ export ISE_PLATFORM=linux-x86
$ export PATH=$PATH:$ISE_EIFFEL/studio/spec/$ISE_PLATFORM/bin
```

You may want to add the last three lines to your ~/.bashrc file in order to have them set the next time you log in.

Install Gobo

Download from [14] the Gobo library (file gobo34.tar.gz). Copy the downloaded file into the target location (i. e. ~/) and extract it.

```
$ tar xzf gobo34.tar.gz
$ export GOBO=~/gobo
$ export GOBO_OS=unix
$ export GOBO_EIFFEL=ise
$ export PATH=$PATH:$GOBO/bin
```

Install, Patch and Compile e-Posix

Download from [5] the e-Posix library (file `eposix-2.2.tar.gz`). Copy the downloaded file into the target location (i. e. ∼/) and extract it.

```
$ tar xzf eposix-2.2.tar.gz
$ cd eposix-2.2
$ export EPOSIX=~/eposix-2.2
$ patch src/papi/papi_unistd.e < ~/ma/patches/papi_unistd.e.patch
$ patch src/epxc/mime/epx_mime_scanner.e \
    < ~/ma/patches/epx_mime_scanner.e.patch
$ ./configure --prefix=$EPOSIX
$ make
$ make install
$ cd ..
```

Patch and Compile EiffelStore

EiffelStore is shipped with EiffelStudio. Use the following commands in order to patch and compile it:

```
$ cd ~/Eiffel56/library/store/dbms/rdbms/odbc/Clib/
$ patch odbc.c < ~/ma/patches/odbc.c.patch
$ cp ~/ma/config.sh .
$ sh Makefile.SH
$ make
```

Compile and Run the Prototype

The OS is now ready to compile the prototype. In order to do that type the following commands:

```
$ cd ~/ma
$ geant install
$ geant compile
```

In order to execute the system just type:

```
$ geant run
```

Chapter 6

Summary and Possible Extensions

6.1 Summary

The goal of this project has been to design an e-mail tool which will provide an applicable solution to the information overload experienced by power users of the e-mail technology.

In order to achieve this goal we developed a set of abstract classes which define main the components of the whole system. These classes offer an object-oriented foundation for implementing both the common features of an e-mail client and innovative features which should simplify the life of advanced users.

The most important innovative feature of the system is the advanced fast search: using the query language defined in Sec. 4.2 users can perform searches across the whole set of messages in a faster way with respect to the common e-mail clients.

The design of this e-mail client provides an innovative approach to the information overload in the e-mail domain taking advantage of database technology.

The foundation offers a base for many other projects which aims to build a complete and usable e-mail client. Some possible extensions which can be implemented using this framework are described in the next section.

6.2 Possible Extensions

In this section we describe what need to be done in order to implement a complete e-mail tool.

First of all the tool need a set of protocol handlers in order to send and receive messages. In the prototype we have implemented a very rough SMTP handler using the framework provided by the deferred classes of the protocol cluster.

The classes of this cluster should be used and extended in order to exploit the specific protocols features. In order to make the indexing process incremental messages should be indexed as they arrive or leave, this behavior can be easily implemented in the protocol handlers as done in class `SMTP_PROTOCOL_IMPL`.

We think that it is very important that tools which allow communication through the Internet are implemented in compliance with the standards. For this reason the e-mail client should implement a full MIME support. A way to implement it exploiting the design of the system is described in Sec. 3.1.6.

In order to make the system "user friendly" a GUI should be implemented for the whole system. The interface should provide a set of dialogs to compose and display messages in an easy way. In addition a dialog should be implemented in order to perform advanced and "as-you-type" queries without requiring the knowledge of the query language.

Another possible extension to the e-mail tool is the addition to the search engine of the capability to look for messages which belong to a given concept using synonyms search and stemming (see paragraph "stemming" on page 6).

A nice feature which can be added to the system is to put a given message into the context of the conversation that it belongs to. The paper "Threading Electronic Mail: A Preliminary Study" [17] suggests some way to do it.

Bibliography

[1] Karin Becker and Simone Nunes Ferreira. Virtual folders: Database support for electronic messages classification. In *CODAS*, pages 163–170, 1996.

[2] S. Bradner. Key words for use in RFCs to Indicate Requirement Levels. RFC 2119 (Best Current Practice), March 1997.

[3] D. Crocker. Standard for the format of ARPA Internet text messages. RFC 822 (Standard), August 1982. Obsoleted by RFC 2822, updated by RFCs 1123, 1138, 1148, 1327, 2156.

[4] Debian GNU/Linux. `http://www.debian.org`.

[5] Eiffel POSIX binding. `http://www.berenddeboer.net/eposix`.

[6] E. Gamma et al. *Design Patterns*. Addison Wesley, 1994.

[7] S. Ferreira. A query language for retrieving information in electronic mail environments, 1997.

[8] N. Freed and N. Borenstein. Multipurpose Internet Mail Extensions (MIME) Part Five: Conformance Criteria and Examples. RFC 2049 (Draft Standard), November 1996.

[9] N. Freed and N. Borenstein. Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies. RFC 2045 (Draft Standard), November 1996. Updated by RFCs 2184, 2231.

[10] N. Freed and N. Borenstein. Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types. RFC 2046 (Draft Standard), November 1996. Updated by RFCs 2646, 3798.

[11] N. Freed, J. Klensin, and J. Postel. Multipurpose Internet Mail Extensions (MIME) Part Four: Registration Procedures. RFC 2048 (Best Current Practice), November 1996. Updated by RFC 3023.

[12] Google Desktop Search. `http://desktop.google.com`.

[13] Gmail. `http://gmail.google.com`.

[14] Gobo Eiffel Project. http://www.gobosoft.com/eiffel/gobo.

[15] Google. http://www.google.com.

[16] R. Jason. ifile: An application of machine learning to e-mail filtering, 1998.

[17] David D. Lewis and K. A. Knowles. Threading electronic mail - a preliminary study. *Information Processing and Management*, 33(2):209–217, 1997.

[18] Lookout. http://www.lookoutsoft.com/Lookout.

[19] Maildrop. http://www.courier-mta.org/maildrop/.

[20] Mairix. http://www.rpcurnow.force9.co.uk/mairix.

[21] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2nd edition, 1997.

[22] K. Moore. MIME (Multipurpose Internet Mail Extensions) Part Three: Message Header Extensions for Non-ASCII Text . RFC 2047 (Draft Standard), November 1996. Updated by RFCs 2184, 2231.

[23] MySQL. http://www.mysql.com.

[24] Nelson Email Organizer. http://www.emailorganizer.com.

[25] Procmail. http://www.procmail.org.

[26] P. Resnick. Internet Message Format. RFC 2822 (Proposed Standard), April 2001.

[27] Richard B. Segal and Jeffrey O. Kephart. Dynamics of incremental learning in an e-mail classifier.

[28] Zoe. http://zoe.nu/.

[29] Zoot. http://zootsoftware.com.

Acknowledgements

I would like to thank my supervisor Joseph N. Ruskiewicz for his helpful feedback and for the reviews of my report.
I want to thank Prof. Dr. Bertrand Meyer for giving me the opportunity to do my master project in his group.
Last but not least I would like to thank my parents for giving me the possibility to study at the ETHZ.