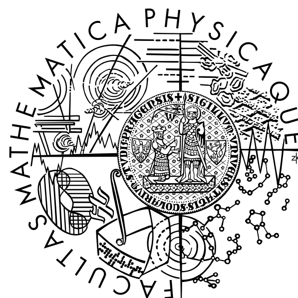
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE



Luboš Kulič

Automatické třídění pošty pro IMAP servery

Katedra softwarového inženýrství

Vedoucí bakalářské práce: RNDr. David Bednárek
Studijní program: Informatika

2006

Na tomto místě bych rád poděkoval vedoucímu práce panu RNDr. Davidu Bednárkovi za zajímavé téma a podnětné připomínky a návrhy při řešení ročníkového a posléze bakalářského projektu.

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

V Praze dne 8. 8. 2006 Luboš Kulič

Contents

Abstract **5**

1 Introduction **7**
1.1 E-mail classification 7
1.2 Related Work 8
1.3 Motivation 9
1.4 Project goals 9
1.5 Thesis organization 10

2 Program analysis and design **12**
2.1 Messages sorting and sorting rules 12
2.2 Data access 15
2.3 Event handling and logging 16
2.4 Logical division of the application 16

3 Program run **18**

4 Rules correcting **19**
4.1 Final and potential rules 20
4.2 Creating a new potential rule 21
4.3 Basic vs. Advanced rule-correcting algorithm 23
4.4 Condition relevance determining – Basic algorithm 24
4.5 Condition relevance determining – Advanced algorithm . . . 25

5 Implementation **28**
5.1 Used libraries 28
5.2 Hiding the libraries 29
5.3 Security 30

6 Conclusion **31**

A Supplied CD **34**

B Apofis quick start guide **35**

Bibliography **38**

Název práce: Automatické třídění pošty pro IMAP servery
Autor: Luboš Kulič
Katedra (ústav): Katedra softwarového inženýrství
Vedoucí bakalářské práce: RNDr. David Bednárek
e-mail vedoucího: david.bednarek@mff.cuni.cz

Abstrakt: Jelikož množství přijatých e-mailových zpráv rapidně stoupá, jsou uživatelé nuceni třídit je do několika kategorií. V předkládané práci představujeme nástroj, nazvaný Apofis, který pomáhá uživatelům automatizovat jejich rutinní každodenní práci s IMAP mailboxem tím, že jejich nové zprávy třídí a také postupně vytváří nová pravidla podle jejich chování, ovšem s tím, že stále ponechává uživateli možnost vytvářet, měnit nebo mazat pravidla a ovlivnit tak celý proces třídění a učení. V programu byl zaveden dvojúrovňový systém pravidel – finální pravidla, podle kterých se třídí, a potenciální pravidla, která byla vytvořena podle uživatelova chování a čekají na potvrzení dalšími úspěchy. Byly navrženy dva algoritmy na vytváření potenciálních pravidel – základní, rychlý nicméně v některých případech nedostatečný, a pokročilý algoritmus, který vylepšuje správnost vytváření pravidel tím, že při zjišťování relevance podmínek uvažuje uspořádání celého mailboxu. Tato práce obsahuje nejdůležitější informace a rozhodnutí o návrhu a implementaci aplikace a jejích algoritmů.

Klíčová slova: e-mail, třídění, automatické vytváření pravidel

Title: Automatic mail organizer for IMAP servers
Author: Luboš Kulič
Department: Department of Software Engineering
Supervisor: RNDr. David Bednárek
Supervisor's e-mail address: david.bednarek@mff.cuni.cz

Abstract: Number of received e-mail messages is growing explosively which forces users to classify them into several categories. In the presented work we introduce a tool called Apofis, which helps users to automate their routine every-day work with IMAP mailbox by sorting new messages for them and also by step-by-step creating of new rules based on their behaviour, while it still lets the user create, edit or delete the rules and thus affect the process of sorting and learning effectively. Two-level system of sorting rules is introduced – final rules, which the application uses to sort, and potential rules, which have been created according to user's behaviour and are waiting for more successes to prove their usability. Two potential rule learning algorithms were designed – the basic algorithm, fast but not accurate for some cases, and the advanced algorithm, which improves the accuracy by creating a new rule based on condition relevances determined from the organization of the whole mailbox. This thesis contains the most important facts and decisions about the design and implementation of the application and its algorithms.

Keywords: e-mail, classification, automatic rule-creating

Chapter 1

Introduction

1.1 E-mail classification

Since its birth, Internet has expanded unbelievably – once being only network for exchanging knowledge between scientists it is now used by millions of people to find information, communicate, entertain, make money, ... One of the most popular (and oldest as well) services is electronic mail, or shortly e-mail. It is faster than a telegram but can be as long as classic mail and it's for free. And one can read his/her mail from everywhere on the world. No wonder that a number of messages received is growing rapidly, even 'normal' person using e-mail only for private communication can receive tens of messages per day, while enterprises can easily reach hundreds or thousands.

This amount of e-mails enforces users to sort messages into some directories which usually contain messages, that are somehow similar - e.g. refer to the same topic or came from the same sender (or a group of defined senders). Of course most users want some tool to make this sorting for them automatically – that, as mentioned in [Pazzani (2000)], allow users to prioritize some mail and maintain his/her mailbox organized.

Another related task is getting more and more important – junk or spam

filtering. This problem has some specifics, that also imply the way to solve it: most of junk mails are (more or less) recognizable by it's style and usage of specific words. However being very annoying to most of users, this problem has been solved very well in many e-mail clients (such as Mozzila Thunderbird or Apple Mail) and also in most of free mailboxes, so this topic will not be discussed in the rest of the thesis.

1.2 Related Work

There has been a lot of research on the field of e-mail classification and there are many approaches to solve this task. Here we mention only a bit of them.

- Incremental rule growing and pruning – used as an improved Incremental Reduced Error Pruning algorithm called RIPPER in [Cohen (1995)] and then compared to TF-IDF style classifier in [Cohen (1996)]
- TF-IDF[1] – algorithm based on TF-IDF used in tool SwiftFile introduced in [Segal, Kephart (2000)], [Pazzani (2000)] introduces prototype method in many ways very similar to TF-IDF weighting
- Naive-Bayes – very often in junk filtering, employed for classifying in [Rennie (2000)] or software POPFile
- Associative classifier – used in [Itskevitch (2001)] to avoid the unrealistic independence presumption of Naive Bayes

Another text classification approaches like support vector machines, nearest neighbor (kNN) or even a combination of mentioned methods are also possible to use.

[1] Term Frequency - Inverse Document Frequency weighting

1.3 Motivation

There are many programs that allow user to define some rules and then sort e-mails according to them, e.g. almost every popular e-mail client (such as Microsoft Outlook, Mozzila Thunderbird, The Bat, ...) has this feature included. These applications however enforce users to make up many rules and quite surely edit them and add new ones frequently.

There also are many classifying systems, proposed or even implemented (as mentioned in section 1.2), that are really sophisticated and fully automated in rules learning. These can without a doubt work with high accuracy and effectiveness, but for many (even advanced) users are too complex for their every-day e-mail sorting and possibly not smart enough to catch the rare and special cases anyway.

But there is missing an application somewhere halfway – it should help user to sort mail according to simple rules, and it should be able to create new rules from user's behaviour. That means, when the user moves a message to another directory and thus changes its classification, the program should create corresponding rule, i.e. a rule which would sort this message to the new place. This rule learning should work successfully especially in the most obvious but also most frequent cases. It must also provide some easy way to delete or edit all rules as well as adding new ones. This behaviour can help user in his/her every day work with a mailbox but also don't try to be smarter than him/her.

1.4 Project goals

The main goal is to design and implement software that will help to automate some routine work with IMAP[2] mailbox. This tool should sort (especially non-junk) incoming messages, as well as outgoing ones, according to

[2]Internet Message Access Protocol, see RFC 3501

rules partly made by the user and partly created by the tool step-by-step in correspondence with user's actions. So as the user moves various messages to appropriate directories of the mailbox, the program should create corresponding rules, so that after some time most of the messages will be sorted automatically and the user will have to move himself only those ones which are rare, somehow specific or do not really correspond to the organization of the mailbox.

It should help to automate especially routine work, so it should be capable of creating rules in particular in the most frequent cases. And on the other hand, it should not try to interpret every little action of the user as a necessarily significant for his/her way of organizing the mailbox. Even if a small amount of new messages would be moved incorrectly, the tool would probably become annoying [Segal, Kephart (2000)]. The newly created rules should thus prove themselves right at least in couple of times, because it is better not to move the message anywhere than to move it to a place user would not really expect it.

The program is intended for experienced users to spare them some time, it should thus work alone without interaction with some e-mail client or user and there is no need for GUI. The settings and especially the rules have to be easily read and edited as well as portable, preferably by just copying some file(s), so that the user can adjust the behavior of the program quickly or just bring the settings from one computer to another.

1.5 Thesis organization

The purpose of this bachelor project was to design and implement a piece of software (according to goals as mentioned in section 1.4), the result of this is called Apofis (as an acronym of **A**utomatic e-**P**ost **O**rganiser **F**or **I**MAP **S**ervers) and is distributed with the thesis (see Appendix A).

This thesis discusses the most important decisions made in the process of

creating Apofis and accomplishment of the project goals. It is recommended to get familiar with the Apofis first. For that, User Guide, Program Documentation and Code reference are distributed with the program (for details see Appendix A) and there is also a Quick Start Guide included in this thesis (as Appendix B).

Here a is brief description of following chapters of the thesis:

Chapter 2 deals with program analysis and design, main structure of the application and data storing and accessing.

Chapters 3 and 4 present two most important algorithms – main program run and learning from user's behavior (and trying to correct the set of rules to correspond to that).

Chapter 5 contains some implementation comments and describes used libraries and the reasons for using them.

Finally, Chapter 6 summarizes the whole thesis and tries to make a conclusion of the program and of how it fulfill the goals.

Chapter 2

Program analysis and design

2.1 Messages sorting and sorting rules

The main purpose of the application is to sort e-mails according to rules and to add new ones, thus one of the most important parts of the design was to decide the way program will sort messages and define some form of its rules.

As described in section 1.2, there are many approaches to e-mail sorting and filtering. Before deciding what way should be chosen, some presumptions were made:

1. The sorting rules must be easily human-understandable as well as readable by program.

2. Sorted messages are not junk, but they may be both wanted and unwanted and both of them are usually in very similar style – nice example is given in [Pazzani (2000)]: it studies some message set from faculty member where unwanted e-mails included some talk announcements and grant opportunities, and are very hard to distinguish only by style or message bodies.

3. User of e-mail and this program does not really think statistically of

his/her mailbox. When he/she wants to sort some message to specific directory, it's not because it contains more of some words than other ones. However rules like this (such as prototypes) are quite suitable for automatic rule generation, they are really hard for user to make up.

Based on the first point, Naive-Bayes model was ruled out because it does not have real rules or any other mechanism for user to adjust behaviour of the sorter.

Then, because of the second point, it was decided to sort only according to message headers and don't consider bodies. Reasonable non-junk e-mail has the most important message of the body in its subject and this, together with sender and some other headers such as receiver (e.g. for e-mail conferences), should be sufficient to recognize the class of the message.

When we sum all that up, we are left by simple rules containing condition(s) about message headers. The exact form for the program was defined as a pair of *condition* and *place* with the meaning:

message satisfy *condition* $\Longrightarrow$ message belongs to directory *place*

Condition is build by predicates *is_equal* and *contains* which compare value of a specific key (message header) with given one, where these two values have to be perfectly equal for *is_equal* to be true, while for *contains* it is sufficient when the first value includes the second one as a substring. These predicates can be then combined with logical operators *and*, *or* and *not*.

These rules will be simply evaluatable – program will fetch values for headers of the sorted message and substitute them for appropriate header names in the predicates and then just evaluate the whole condition consisting of a combination of these predicates.

A new message can be thus classified if it satisfies any rule. In the first

version of the application, only the messages which satisfied just one rule were classified and moved. But this model turned out to have some serious disadvantages:

1. It is not really transparent for the user, especially for a hierarchical structure of the mailbox – messages belonging to the child directory must not satisfy rules of the parent folder, therefore these parent rules has to be almost as complex as the child ones. Whereas the logical way would be to have a general rule for the parent and more specific ones for the child directories.

2. When the program want to make a new rule according to some user's action, it should not produce rules which would be satisfied by messages satisfying other rules. And because it often makes these new rules according to messages formerly being in other directories, as explained in Chapter 4, this requirement is hard to fulfill. The algorithm has to edit some old rules and even that is not always enough.

Because of these major drawbacks, it was decided to change the semantics of the rule set so that if a message satisfy more than one rule, the program tries to find the most specific (complex) rule and assume the corresponding place has the most in common with the message. To improve the behaviour of this, user has the opportunity to define his/her priorities for various headers and the application will then take them into consideration about the most important satisfied rule.

Rules formed this way has logically hierarchical structure, so the best way to represent them in program runtime would be also hierarchical. And it can be used to make the evaluation more efficient by pruning – not evaluating branches not important for the final evaluation of the condition.

2.2 Data access

However the application is to be some kind of IMAP client and therefore (to correspond to the IMAP main idea) should be really 'thin' program, it needs to have some external data stored on a hard drive. At least, it will be the list of sorting rules, logging information of the mailbox(es) and settings.

One of the program requirements is to store these data human readable and also portable. Because of the rules, which probably will contain header values with non ASCII characters (especially for users out of English-speaking countries), it should support various codepages and should be easy readable on computers anywhere.

Designing some proprietary file type would also mean to provide some editor or reader to handle it and that is not really easy readable. So it was decided to use some standardized format. On the other hand, to achieve simple program readability of the data, the files should have some strict format of the content.

To fulfill all these needs, XML was chosen. It is (strictly speaking) a text file, so it is as easy readable as possible, but it also supports codepages and has a possibility of defining strict document schema (via DTD or XML Schema), which the document can be validated against. And it is also quite easy to handle by a program as well as there are many free (or even open source) libraries to do that.

Document Object Model (DOM) of XML permits storing and working with hierarchic data naturally – the node of the data can be easily represented as an element, which can have children or textual content (or both).

Logging information, or at least the passwords, should not be stored as plaintext, on the other hand, it would really bother the user to enter the password every while, so it should be stored somehow. Therefore some kind of secure storage should be used. The easiest way to do that is to have a special XML file which would contain the passwords in an encrypted form.

2.3 Event handling and logging

Because the application will run standalone without any interaction with user, it will be important to log all the important information about sorting and rule correcting – especially some important events, e.g. when the program checks a mailbox, sorts a message or invent some new rule. And in the debug phase of the development (or after that in the case of malfunction) it will be necessary to print out some debug messages for that.

To unify all the communication with the user and to centralize decisions about program events it was decided to include an event handler. If any kind of important event (both expected and unexpected) occurs in the application, it will be reported to the handler (along with some message, type of event and level of importance). The handler decides what to do according to its policies – if the event should be reported to user, if it should be logged or if it is some error, that should stop the run of current function (or the whole application).

This way it is the place of occurring event, where it is decided what kind of event it is and how it is important, but there is still only one place, where it is decided what to do with each kind of event. That of course allow changing this policy in the future or by program settings or command line options.

2.4 Logical division of the application

To separate different tasks the application is to be divided to several parts (as shown in Figure 2.1).

The most important is the *Manipulation kernel*, which is responsible for the main run of the application (as described in Chapter 3) and initiates all the important work of the application by calling member functions of owned classes.

The *Rules* are holding one set of rules and are able to sort given message according to that. It also provides some interface to allow other parts handling, editing and creating the rules without knowing their insides. *Rules* are edited by *Rule corrector* to follow user's behaviour.

Manipulation kernel holds and controls several *Abstract sortable boxes*, which is an abstraction for a box of items which could be classified. *Mailbox* is a concrete child of *Abstract sortable box* which represents one IMAP mailbox.

All the classes can find out their settings via *Options*.

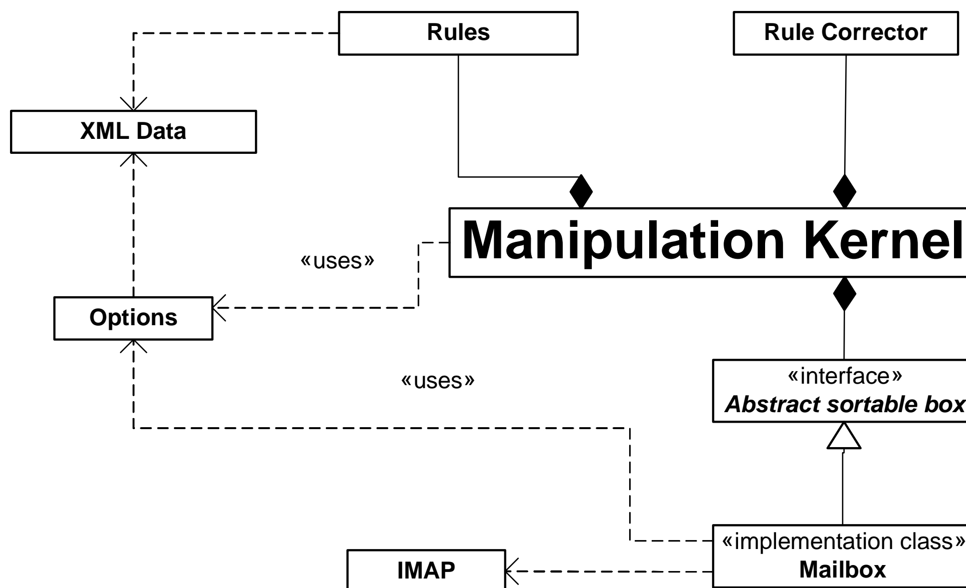


Figure 2.1: Basic division of the application

Chapter 3

Program run

The user of the application should barely know about it, so it should run without interaction with him/her and periodically check his/her mailbox(es), just like regular e-mail client behave.

So the main program run is quite straightforward – after initialization, it logs in all the controlled mailboxes (if a password is required, it asks the user for it and then save it via the secure storage). Then it checks all the boxes and wait until the next check time of some of the mailboxes. This is repeated until the user closes the program or some unexpected dangerous situation arise (especially one which could damage program data).

At one iteration of the loop the application checks mailbox for new messages (both in inbox and other directories). New messages in inbox and directory of sent messages (i.e. really new messages) are tried to classify according to rules corresponding to the box and moved to appropriate directory. New message in other directories is considered as moved by user and application supposes, that this message, as well as similar ones, should be sorted to this new directory. According to that the program tries to make up a new rule that would correspond with the situation (for the mechanism of that, see Chapter 4.

Chapter 4

Rules correcting

The most important feature of the application (and also the feature which distinguishes it from normal sorter included in almost every e-mail client) is the ability of guessing new rule only from user's behaviour. There are many possible ways of rule learning but almost none of them is successful at 100%, so the program should work on best-effort basis. According to the method of learning, they can be divided into two main groups:

- **Batch-building of rules**. This method needs some set of already classified messages (called training set) to learn itself and usually another set (test set) to decide whether the created rules are sufficiently accurate.
- **Iterative-building of rules** means the classifier learns itself iteratively with every moved, i.e. (re)classified, message.

Since e-mail is in its nature dynamic and new messages come almost every day, the iterative approach is more suitable and it was taken into consideration as a necessary need for the rule-building algorithm.

Another important presumptions about mailbox hierarchy and user's behaviour were made:

1. The application runs with (almost) no interaction with the user, so the only way for the user to classify a message differently than the program is to move it to another directory.
2. All the messages in a specific directory has something in common – same or similar values of some attributes. Some of these properties characterize the directory and make it somehow unique. If the user moves a message to this directory, it has some of this properties and they can be used to create a rule, which would sort this message (and possibly many other messages from the directory) correctly.
3. Not all moved messages mean wrong rule set – sometimes the user would like one specific message in a directory where other similar messages don't belong. This behaviour should not affect or even damage the rule set. On the other hand, if the user moves some number of similar messages to the same directory, it usually means, that he/she wants to sort them this way.

4.1 Final and potential rules

Clearly, there is a conflict between the second and the third presumption. The second one says all directories are unique and their messages correspond to their characteristics, so that when the user moves a message somewhere, it should be considered to belong there and (because of the iterative approach) there should be created a new corresponding rule. On the other hand, the third point implies the rule-creating process should be more careful and should not try to insert a new rule every time user moves something.

To solve this problem, the rule-learning process was structured into two phases and according to that, the rules was split into two categories:

- **Final rules** are the rules according to that the program actually sorts messages. They are either the ones inserted by the user or proposed

by program and proved themselves right.

- **Potential rules** are created by the program after a message is moved by the user. They should correspond to the moved item as well as the new directory. Each one of them has its hit count – number of moved messages which satisfied the rule (i.e. satisfied the condition and matched the place).

When user moves a message, the program tries to find a formerly created potential rule which would be satisfied by the message. If it is successful, it increases the hit count of the rule and when a boundary count (adjustable by user) is reached, the rule is inserted as a final and from now on, it is considered in new message sorting.

When no appropriate potential rule is found, the program tries to create a new corresponding rule (for details see following sections) and if successful, inserts it as a new potential rule.

This way the the user not only initiates the creating of new rules, but also has to confirm the guessed rules by similar behaviour in the future. That gives the necessary feedback without bothering the user much. And of course, he/she has always the possibility to delete all potential rules, which do not match his/her needs and could lead to unintended behaviour of the sorter. So he/she really has the full control over the rules creating.

4.2 Creating a new potential rule

Thanks to the system of the final/potential rules, creating of a new potential rule became a little less important – if the rule is guessed wrong, the user will probably not perform a similar action again. Thus the rule never became final and will not make any harm to the sorter.

On the other hand, the application must be able to make a rule that sorts the moved message to the right directory – the one it was moved to –

and should have some other characteristics:

1. Is satisfied by as many messages in the right directory as possible. That corresponds to the second presumption mentioned at the beginning of this chapter – the rule should not only sort the moved message here, but also messages that are similar to it and to other messages in the directory.

2. Should sort no (or as few as possible) messages currently being in another directory into this one. This supposes that if the user wanted other messages to be sorted in a specific directory, he/she would already moved them there. Therefore the application should not add rules which would match messages from other directories (and probably sort similar messages wrong).

3. Is as short and simple as possible – although some complex rule would likely be satisfied by more messages in the box or by less messages from elsewhere, it would be too complicated to make it and that would probably make the rule creating too long. Not to mention it would be less readable for the user. The program should rather then create the best possible rule try to make a rule that would be "good enough".

The making of a new rule is triggered by moving a message to specific directory and the new condition is to be satisfied by that message, so it must be built from some attributes of the moved message. The main task for the algorithm is therefore to pick some of these attributes and create the accurate condition (by joining all of them by logical and). To do that, it has to determine *relevance* of each condition – that means how much does the condition distinguish between the right directory and the other ones.

The only source for this decision is the mailbox and messages it contains, so to determine relevance of each condition, the program has to find out, which are characteristic for the directory (and not really characteristic for the rest of the mailbox). This method could however lead to giving too much

significance to not really important headers which have big count more by a coincidence, for example:

Some user can have two main categories depending on the senders: 'friends' and 'work' and let's say that his/her business uses MS Outlook by default while most of his/her friends have Mozzila Firefox. Although sorting the messages according to e-mail client could work quite well, it is probably not what this user wants – more likely, he/she would like to sort all e-mails from domain 'my_business.com' to the directory 'work' and others to 'friends', because not everyone in the work has Outlook and vice versa.

But there is no way the program could distinguish more and less important attributes (especially because it can vary from user to user), so it uses sorting keys priorities given by user in settings to distinguish really important condition from the ones with high count but no real relevance.

In the case of moving message from one specific directory to another (not from inbox or sent directory) the application usually has a formerly satisfied final rule available. The program supposes this means the moved message has the characteristics of the old directory (because it satisfied the rule) but it belongs to the other place, so the new rule must be more specific (to avoid sorting similar messages to the old directory in the future). Straightforward solution was chosen – to create a new rule on the basis of the old one just by adding some conditions. This method also allow to cover hierarchic mailbox organization.

4.3 Basic vs. Advanced rule-correcting algorithm

The first draft of the application contained quite simple rule correcting algorithm, partly because of simpler design and partly because of speed. In the testing phase it turned out that however this algorithm produces quite satisfactory results in most cases, there could be made some improvements

and the cost of less speed would not be as bad.

On the other hand, the algorithm is not all bad and has some advantages, so it was decided to keep it as an option (called *basic rule correcting algorithm*) and to add an improved algorithm (called *advanced*).

4.4 Condition relevance determining – Basic algorithm

This algorithm is based on a simple presumption – if user moves message from directory A to B, it should not affect messages in other directories. The moved message has some of the characteristics of the old directory and now the user is telling it also has some characteristics of the new directory, so an attribute of the message should be typical for either one of them. Therefore, the algorithm only determines relevance of a condition based on directories A and B.

Because the new created rule should distinguish messages belonging to B from the ones belonging to A, it should not match any messages in the old directory. In this algorithm it was chosen to discard from the condition all predicates that are satisfied by any message in A. This way it is not possible for any message that belongs to A to be sorted into B and that keeps the rule set consistent (messages sorted into some directory before should be sorted there with the new rule in the set as well). On the other hand, this approach is quite strict and sometimes can disqualify important and characterizing attributes with only a few appearances in old directory.

After that, the relevance of one simple condition with m appearances in directory of n messages could be simply expressed as $\frac{m}{n}$. But we want to decide, what condition is the most relevant according to messages in one specific directory B, so it is sufficient to compare them only according to m (adjusted according to user priorities).

4.5 Condition relevance determining – Advanced algorithm

Testing of the (basic) algorithm showed some drawbacks and this version tries to improve them. They are especially:

1. Determining relevance only based on two directories (old and new). This behaviour works quite well for messages moved from one non-inbox (and not sent) directory to another one – the algorithm can presume the message has something in common with other messages in the old place and it was enough to sort it there, i.e. distinguishes it sufficiently from another places. And it also knows the new rule would be probably based on one which originally sorted the message into the old directory, therefore there is almost no possibility that the new rule would be satisfied by wrong messages. However, in the case of message moved from inbox or sent directory (so that not satisfying any final rule) to some specific new directory, this presumption is not right and the new rule does not really distinguish the new directory from others.

2. Discarding all conditions which are satisfied by any message in the old directory – as the first point, this is also more a problem in the case of message moved from inbox or sent directory.

The first problem has quite simple solution – to determine relevance of a condition not only from old and new directories but from all the directories present in the mailbox.

To solve the second problem, the simplest way would be to just don't throw away all conditions satisfied in some wrong directory, but set some boundary count of satisfying messages and only if a condition overcome this number, it should be discarded. However, this method would probably not make it any better. It is quite different when some condition is satisfied by 1 message in directory of 3 than in directory of 400.

And from the other point of view, if two conditions are satisfied by the same number of messages in wrong directories, the one with more successes in the right place is the one to pick to the new rule.

This leads to redefining relevance of a simple condition to some formula of counts of appearances in both right and wrong directories and of overall sums of messages in these places. As shown in [Quinlan (1987)], the ground for the relevance can be expressed in a contingency table:

	in right directory	in other directories
condition satisfied	a	b
condition not satisfied	c	d

It is intuitively clear that the higher is the ratio of a and c while the ratio of b and d is still low, the higher the relevance is. The first way of determining this significance tried was just to count a ratio of positive cases (i.e. either satisfied condition in the right directory or not satisfied elsewhere) and all messages:

$$\frac{a+d}{a+b+c+d}$$

But this do not distinguish cases with the same number of positive cases ($a+d$) but with different count of matches in the right directory (a). Therefore some modifications were tried, which would give more importance to that. But however it brought some improvement, it still don't really express the idea of condition relevance – how confidently the condition distinguishes a directory from others.

To find a better method, a statistic view of the problem was considered. In the speech of statistics, the relevance of the condition is a degree (statistical significance) of contingency (or dependence) between the two attributes of a message:

- whether a message belongs to the right directory
- whether it satisfies the condition

As mentioned in [Wikipedia: Contingency table], several tests should be used to count the significance - χ^2-test, G-test or Fisher's exact test. For the purpose of the relevance computing in a program, the χ^2-test is the most suitable one because it has the least complexity. On the other hand, it is not suitable for cases with very small values of the counts a, b, c or d (small means here approximately less than 5 according to [Connor-Linton, Ball]). In this application however, the goal is not to compute the most exact significance, but to compare relevances of several conditions, so this little drawback will not make much harm.

The χ^2 of a contingency table above can be expressed as:

$$\sum_{c \in \{\text{all cells}\}} \frac{(O_c - E_c)^2}{E_c}$$

where O_c means *observed* (i.e. really present) number of cases in the cell c and E_c *expected* theoretical number appropriate for this cell, i.e. the number corresponding to *null hypothesis* (independence of belonging to the directory and satisfying the condition). This *expected* number of cases for a cell can be computed as the product of sums of the row and the column divided by a total number of cases. For example for the cell in the upper left corner of the table it is:

$$\frac{(a+c)(a+b)}{a+b+c+d}$$

This number is then adjusted by user preferences.

Chapter 5

Implementation

Implementation of the functional kernel of the application just followed design and algorithms explained in Chapters 2, 3 and 4.

5.1 Used libraries

Some parts of the program was decided not to implement from scratch, but to use available libraries. Concretely, they are used to provide:

- **Communication with IMAP mailbox** – c-client library[1](part of the University of Washington IMAP toolkit), API to make (not only) IMAP clients which supports many features and authentication methods.
- **Parsing of XML files and working with the data** – Xerces-C++ Parser[2], portable library faithful to the XML 1.0 recommendation.
- **Secure storage** used for remembering passwords to mailboxes – Crypto++ library[3], class library of many cryptographic schemes with FIPS 140-2

[1]For details see http://www.washington.edu/imap
[2]For details see http://xml.apache.org/xerces-c
[3]For details see http://www.eskimo.com/∼weidai/cryptlib.html

validation.

The reason for this decision was mainly the fact that the most important purpose of this application was to design and implement a tool to sort messages and create rules according to user's behaviour because there was no similar tool known. Unlike that, parsing of XML files or RFC compliant IMAP client were implemented many times and there are many available libraries free to use in the application. And it is also better to use well tested and widely used library than to try to invent some own way and likely make the same mistakes as its authors again.

5.2 Hiding the libraries

However useful the libraries are and however perfect they might seem to be, it is not a good idea to use them directly in the whole program. Sometimes in the future they could stop being good enough or better ones could be explored etc. Therefore all of them were wrapped into program's own class or classes, which have interfaces general enough to presume all the functionality could be reimplement to use another library and the rest of the application don't have to know. This presumption is supported by these observations:

1. **Data** – program works with DOM, so any other library following W3C recommendation will probably have almost the same interfaces and surely the same functionality, so porting will be quite easy.
2. **IMAP communication** – functionality of any similar library should be based on IMAP RFC, so it should offer similar functions (based on IMAP commands).

Some kind of wrapping or hiding was especially necessary in the case of IMAP library c-client because it is not an object library. So the other reason was to allow other parts of the program to work with some easy interface

which would give them needed complex functionality. Special class working as a facade of all the functions of the library was created to do this job.

5.3 Security

In the case of security storage, the security itself was the main reason. However all the main cryptographic algorithms are open, so the only task is to implement some, most of the successful attacks even to theoretically safe schemes was caused by mistakes in implementation. So the usage of a certified library is most likely a better way. For real safety, one of the most secure symmetrical ciphers at the time was chosen – algorithm AES (Rijnadel).

And because many users use the same password on every mailbox (however dangerous it is), before ciphering the passwords are *salted*[4] by string unique for every mailbox, so that the same password ciphered as a password for different box looks different.

[4] as explained for example in [Wikipedia: Password salting]

Chapter 6

Conclusion

The main goals of the project was to design and implement software tool to automate routine, every-day work with an IMAP mailbox which would follow these requirements:

1. Sorts new incoming and outgoing messages according to some kind of rules given by the user.
2. Also creates these rules step-by-step as user moves messages in the mailbox.
3. The newly created rules have to prove themselves right to be inserted in the set to avoid unintended sorting based on rare action.
4. Works (almost) without interaction with the user.
5. All the rules and settings are easily readable and editable and yet portable.

To fulfill these needs, program Apofis was created. In the following text there is an summary of if and how it satisfies each of these goals.

A model of quite simple rules was introduced, which can yet cover almost every message category by giving a condition made of predicates of message

header values with the possibility to make really complex conditions as well (by connecting number of these predicates with logical operators). The message is classified and moved to the directory according to a most important satisfied rule – by default the most complex, but the meaning of the most important can be adjusted by giving priorities to various headers.

The application runs standalone, the only interaction with the user is asking for and retrieving a password to a mailbox, which is then safely stored though to don't bother the user again. On the other hand, to make it possible to trace the behaviour of the application, e.g. find out what rule was satisfied by a message, what one was newly created etc., unified event and logging system was introduced and all the events in the program are logged together with their time, place of occurrence and level of importance.

The rules and all the settings are stored in XML files, which thanks to the textual nature of XML can be read on every computer, but also supports storing the rules (and especially the conditions) in natural hierarchic way. And it allows to define various encoding pages of the document, so the users can build rules with their native language words without any trouble.

To fulfill the points 2 and 3, a two-level system was introduced – final and potential rules. The program sorts new messages only with the set of final rules, the potential ones are used to determine which of the rules created by the program are really useful – this way also follows the requirement of the point 4. After a message is (re)classified by the user, program first tries to find some suitable potential rule and if successful, the rule's count of successes is increased and if it reaches the boundary count, the rule is made final.

If no satisfied potential rule is found, a new one is tried to create. To do that, two algorithms are present – the basic and advanced one. Both share the same ground – they try to make the new rule from some of the headers of the moved message. The headers with values most characterizing and most distinguishing the directory are chosen. The level of this relevance of particular header is determined according to other messages in the mailbox

– the basic algorithm works only with the directories from and where the message was moved, the advanced one use all the directories present.

Both of these algorithms fulfill quite well the needs for the application – they are capable of creating suitable rules for most frequent and not very difficult cases. The weakness of the basic algorithm – creating rules for messages moved from inbox (or sent directory) – is solved in the advanced version by processing all the directories and thus determining the significance of distinguishing of the condition more properly. The basic version on the other hand works still very well in the case of messages moved from other directories and has the advantage of higher speed.

Due to the possibility to switch between these two versions, the user can enjoy the advantages of both of them by using the advanced one at the beginning to create rules corresponding to his/her mailbox organization and then switch to the basic version for everyday work (when changing the classification of a message to another category is not expected often).

However these algorithms are not perfect, designing and implementing of a really sophisticated rule-learning system would go beyond the scope of a bachelor project. And it would probably have to give the program full control over the rules, which would likely disallow the user to keep survey of the rule set and maybe even to create his/her own rules as well as edit the ones created by the learning algorithm. Introduced algorithms on the other hand follow the goals for the program – they are capable of automating user's every-day work and the way they work tries to bother the user as little as possible.

Appendix A

Supplied CD

The CD distributed with this thesis contains all of the results of this work – this thesis in its electronic form in various formats, including source files, and the application Apofis described in this thesis. The program is provided in two forms – packed in a zip file, for easier copying and distribution and as a plain directory structure, which allows to install it without zip software and also to browse it (e.g. documentation or source files) without copying to the hard disc. Both forms contain all the files necessary to run it and also bianco versions of the settings and rule files with example comments. The whole program documentation is also included, it consists of:

- **User Guide** – explains in detail how to set up and use the application and what is the precise format of all the settings and rules files.
- **Program Documentation** – contains detailed object design together with description of the structure and algorithms of the application.
- **Code Reference** – consists of detailed description of all program's functions, classes and interfaces between them.

The distribution also contains all the source files and libraries necessary to build the application.

Appendix B

Apofis quick start guide

To use the application, it is necessary to unpack it to some place in the hard drive. The directory hierarchy will then look like this:

— **boxes_and_rules** – in this folder, boxes settings, logging informations and sorting rules are stored.

— **data** – this is used internally by the application to store its data, so it is not recommended for user to edit any content of it, it is initially empty.

— **docs** – contains whole program documentation.

— **libs** – all library files necessary to build the application are stored here.

— **logs** – a place to store log files of the application. Every log has a name with a syntax 'logYYYYMMDD@hhmmss', where the time signature means the time of the start of the application instance.

— **options** – contains a file with program settings.

— **src** – all source files of the application are stored here.

- **program main directory** – contains the program win32-executable file (Apofix.exe), libraries necessary for the run (dlls) and a configuration file 'paths.xml', which contains paths to directories where the application should find list of boxes (with appropriate rule sets) and options.

The distribution itself contains all the necessary option files already filled with default values. Thus to run the application user only have to make list of all mailboxes he/she would like to be controlled by the program, the right file to do that is 'boxes_and_rules/boxes.xml'. For the full syntax of that file (and also all the other settings files) see User Guide, the required values for one mailbox are:

- User's name for the box
- Name of the file with appropriate rules
- The mailserver address
- Login name to this mailbox

And these values should be filled in as follows:

```
<box id="name for the box" rules="rules file" type="mailbox">
    <host>server address</host>
    <login>login name</login>
</box>
```

There are two main possibilities for the run of the application (especially important for the first run) depending of what messages should it try to sort:

- **New ones** – when started for the first time, the application finds out what messages are there in the box and in the following runs it will only sort (and create rules according to) new messages.

- **All messages** – when started with appropriate command line option (`--sort_all_items`), the application will consider all messages in the whole mailbox as new. This option is especially suitable for making a set of rules from already sorted mailbox.

At the first run, the user will also be asked to enter the password to the mailbox (which is then safely stored, so this would probably happen only once).

For more details about the settings and other possible modes of running the application, see User Guide.

Bibliography

[Cohen (1995)] W. W. Cohen: *Fast Effective Rule Induction*, Proc. of the 12th International Conference on Machine Learning, 115–123, Morgan Kaufmann, Tahoe City, CA, 1995. http://citeseer.ist.psu.edu/cohen95fast.html

[Cohen (1996)] W. W. Cohen: *Learning Rules that Classify E-Mail*, Proc. of the AAAI Spring Simposium on Machine Learning in Information Access, 18–25, 1996. http://citeseer.ist.psu.edu/cohen96learning.html

[Connor-Linton, Ball] J. Connor-Linton, C. N. Ball: *Chi square tutorial*, http://www.georgetown.edu/faculty/ballc/webtools/web_chi_tut.html

[Itskevitch (2001)] J. Itskevitch: *Automatic Hierarchical E-Mail Classification Using Association Rules*, 2001. citeseer.ist.psu.edu/itskevitch01automatic.html

[Pazzani (2000)] M. Pazzani and D. Billsus: *Representation of Electronic Mail Filtering Profiles: A User Study*, Proc. ACM Conf. Intelligent User Interfaces, ACM Press, NewYork, 2000. http://citeseer.ist.psu.edu/pazzani00representation.html

[Quinlan (1987)] J. R. Quinlan: *Simplifying decision trees*, International Journal of Man-machine Studies, 27:221–234, 1987. http://citeseer.ist.psu.edu/article/quinlan87simplifying.html

[Rennie (2000)] J. D. Rennie: *ifile: An Application of Machine Learning to E-Mail Filtering*, Proc. of the KDD-2000 Workshop on Text Mining, Boston, MA, 2000. http://citeseer.ist.psu.edu/rennie00ifile.html

[Segal, Kephart (2000)] R. B. Segal, J. O. Kephart: *Swiftfile: An intelligent assistant for organizing e-mail*, AAAI 2000 Spring Symposium on Adaptive User Interfaces, Stanford, CA, 2000. http://citeseer.ist.psu.edu/segal00swiftfile.html

[Wikipedia: Contingency table] *Contingency table*, Wikipedia, the free encyclopedia, http://en.wikipedia.org/wiki/Contingency_table

[Wikipedia: Password salting] *Password salting*, Wikipedia, the free encyclopedia, http://en.wikipedia.org/wiki/Password_salting