Department of Electronic and Computer Engineering

Master Thesis

MSc in Distributed Computing Systems Engineering

Conception and implementation of a tool for automated backup status controlling with rule-based behaviour in a distributed backup server system

Sebastian Pritschow

Supervisor: Mr. Peter van Santen

February 2001

A Dissertation submitted in partial fulfilment of the requirements for the degree of Master of Science

1 Summary

2 About this dissertation

To distinct between normal and special parts of the text, the text is written in different font style:

- Source code, names of classes, objects, methods and variables in the text are formatted in the font `Courier New`.
- Epitomes of the source code or configuration file will be drawn in a transparent box in the font `Courier New` (size 10).
- To show that the content of a variable has to be used and not only the variable's name, the variable will be written in squared brackets (e.g. `[variable]`).
- Important items in the text are printed **bold** to highlight them.
- Citations are formatted *cursive in Times New Roman*.

3 Introduction

The fear of losing data is not only at large companies an important topic. Hence, methods have to be used to prevent the loss of data. A simple solution is a regular backup. With small or rarely modified volume of data, it is usually sufficient to mirror the hard disk or to make a weekly backup on a removable data medium. However, with the increasing amount of data backup servers are used, which backup 24h daily 7 days a week on record carrier like tapes.

Backups unescorted from controlling whether backups were correct or not do not make sense. Hence, modern backup programmes offer a good console, in order to control the state of the data protection. Indeed respectively in large companies backup systems with distributed backup servers at different places will be used to prevent the loss of all backups for example after a fire. Therewith a central analysis of the statuses of the backups is no longer possible. In such a distributed backup servers system backup software will be used, which offers a mechanism (for example via email) to forward the backup status information about the current backup to a central place for evaluation. This is information like the status of the backup, the saved files or the tape label, where the backup has been saved.

So far, this backup status information was analysed manually. In the beginning of the system, the handling time for the evaluation was just some minutes each day. With the increasing number of backup servers and hence resulting flow of backup information the evaluation time ascended increasingly up to many hours a day. Thus, the idea was born to automate the analysis.

The aim of this thesis is to develop a programme to analyse the emails from the backup system in a rule-based manner and store the data into a database. Configuration data and analysis rules should be modifiable at any time in a plain way without a programme modification. As the backup statuses are very important, data security, constancy and failure-tolerance will be discussed as well as scalability and parallelism to regard an increasing amount of emails in the future. Of course, the rule-based behaviour of the project will be a main topic.

A prototype of the analysis programme called **EmailParser** will be implemented in Java. The database is an Oracle database connected via JDBC (Java Database Connection). To receive and forward emails the Java Mail API from Sun will be used. The dissertation will show that the rules can be defined in a programming language independent way and that EmailParser can connect to any kind of database. The implementation is only a reference implementation to show the mechanisms in form of a prototype.

4 Background to the project

The new analysis programme EmailParser has to take over a consisting evaluation service of the backup emails. This had been done by human beings by now. It will be shown that the programme will have to evaluate the emails by rules in the same way as its human pedants. To understand the necessary discussions in the analysis, the current system will be described in the next chapter.

4.1 The current system

In the distributed backup server system, up to 35 backup servers save data each day. The currently used backup software Legato NetWorker sends status emails about the backups to an administrator email address.

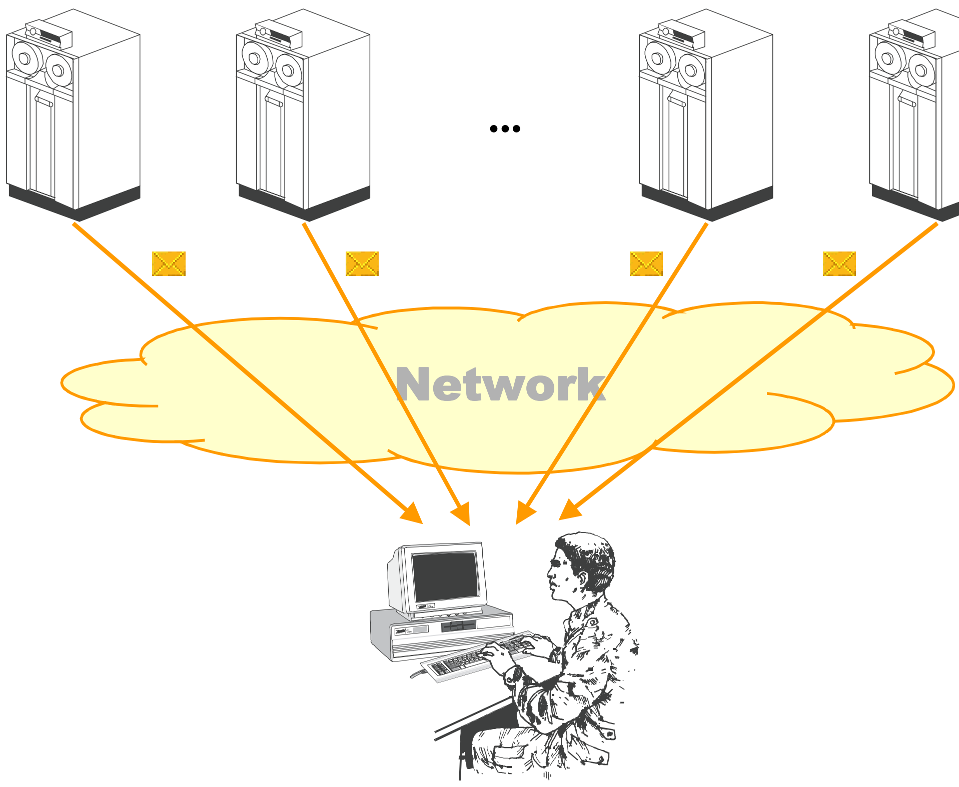

Figure 1 – The old backup status system

Each morning a backup administrator has to evaluate the emails for backup status information. This is information like the backup server, the status of the backup, the tape label where the data has been saved to or an error during the backup. All data will be written into a paper sheet for later usage like recovering backups or to give account of correct backups.

As a matter of course, this procedure is not very comfortable. To locate a backup from a half year ago it took some time to find the sheets in the different archives. In addition, the evaluation time of the emails has been increased from a few minutes up to four hours each day.

As the evaluation of the emails is a routine work, the new analysis programme EmailParser will be developed to automate the daily evaluation and to reduce the evaluation time as well. The backup information can be stored electronically. Thus, they will be usable in a very comfortable way. The new system will be described in the next chapter.

4.2 The new system

The new system differs from the old one in the evaluation of the emails. EmailParser will receive the emails and evaluate them automatically. The backup information will be saved into a database and the email itself will be saved for persistence. Furthermore, emails recording backup errors and warnings or any other problems the system cannot detect, have to be forwarded to an administrator address to be reviewed manually.

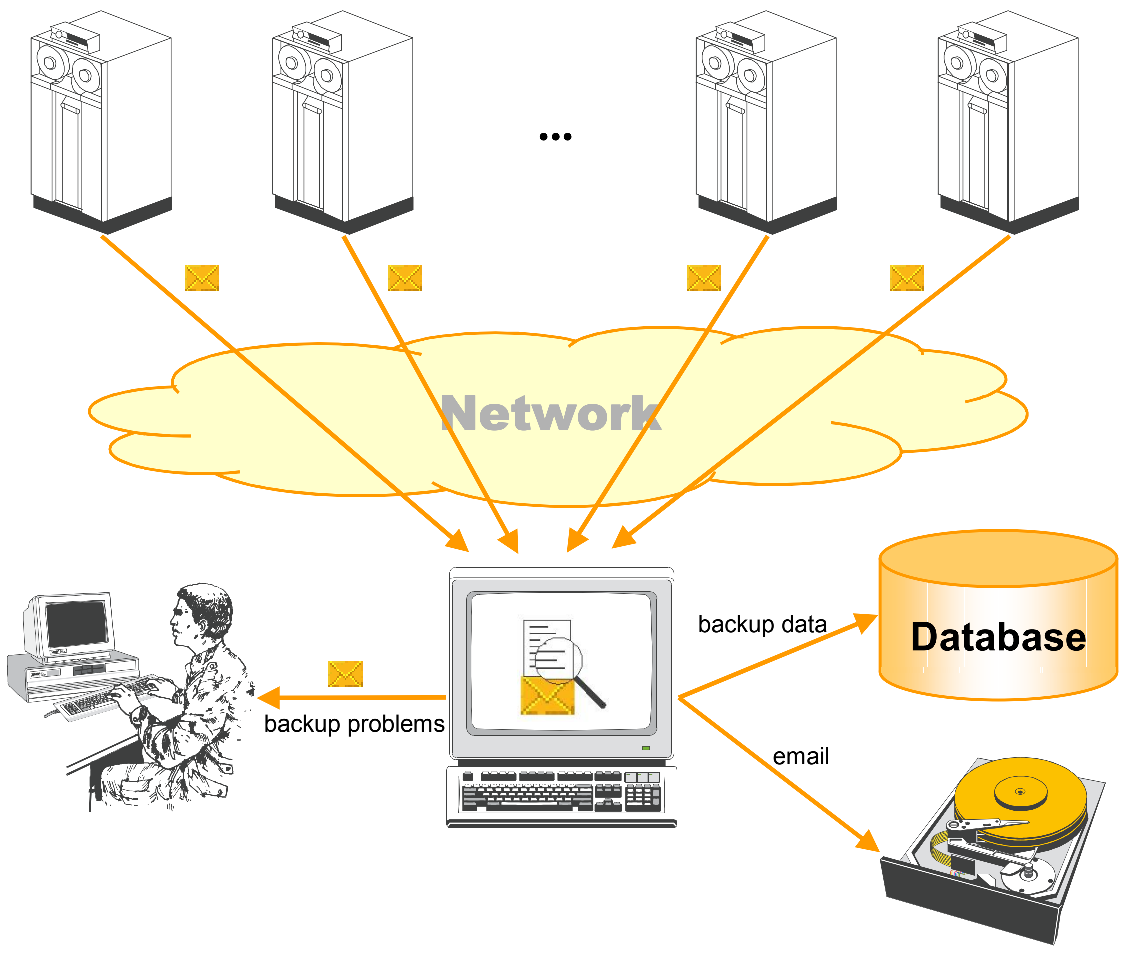

Figure 2 – The automated backup status system

Of course, it is not very difficult to develop a simple parser for emails. However, there is an important problem, which has to be solved by this project. The scenario in the next chapter will explain this point.

4.3 Motivation

If the analysis programme receives the emails from the backup servers containing the backup information, it can parse them and save them to a database. Of course, this will reduce the time to manually parse the emails, but the evaluation of the data has to be done as well as before.

Actually almost all backups complete successfully and at least 95 % of the resulting emails only contain information about the backup server, where to find the backup and which files have been saved. Indeed the other 5 % contain the important information like the occurred error or warning or the request to exchange a full backup tape.

It is obvious to make the programme to evaluate standard backup information and only to filter the anomalous mails with the problems, which have to be handled manually by a human being. Hence, the difficult topic of the project is to make the analysis programme to decide between whether a backup was okay or not. The problem is to give the analysis programme a **rule-based behaviour**.

5 Approach

This chapter will describe the process, which is used to develop EmailParser. The process is abutted to the object-oriented software engineering with UML (Unified Modelling Language) [Scha99].

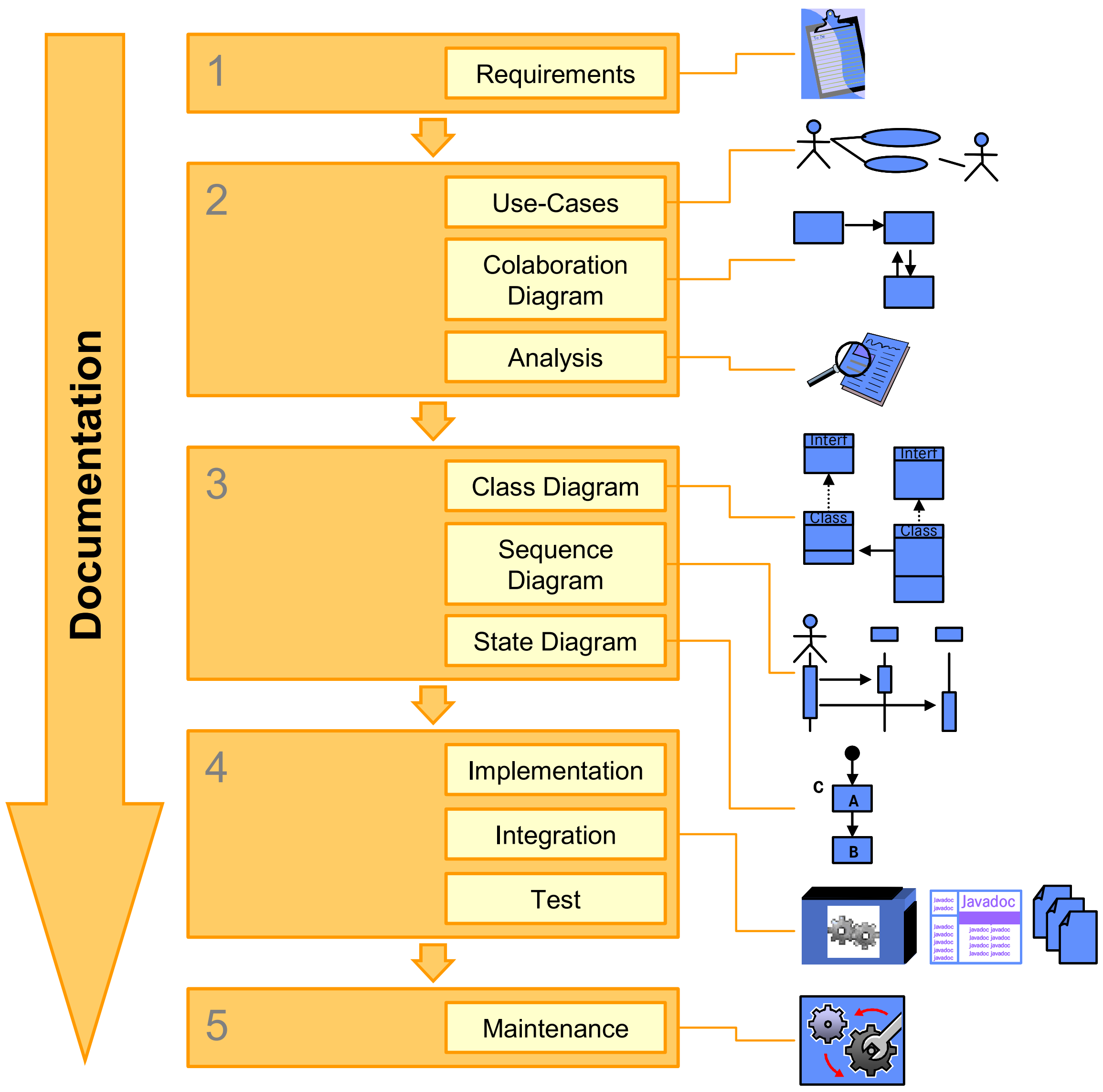

Figure 3 – Development process of the project

The development process of object-oriented software can be divided into five essential steps. The first step is the specification of the **requirements**, which describes the real needs to the software. It is very important to define the requirements very precisely to prevent later expensive corrections of the programme.

The next step is the **analysis**, also called object-oriented analysis (OOA). The analysis of the requirements leads to the use-cases. Collaboration diagrams are useful to find the borders of the system and the interconnections to extern services on distributed servers. Additional deliberations to the future capability of the system and hence resulting additional requirements conduct to the architecture of the programme. The analysis is the most complex and time intensive part of the development, because it can prearrange substantially the possibility of enhancement and therewith the future ability.

The third development step is the **design**, also called object-oriented design (OOD). This is an iterative process where UML diagrams like class, sequence and state diagrams are used to build the core of the system. Therewith the classes and functionality of the programme will be found.

In the fourth step the **implementation, integration** and **test** of the programme in the system will be realized. EmailParser has to be integrated in an existing backup status analysing system. It has to be ensured that the existing system will be replaced by the new one without problems like loosing data. Hence, the new system will be run in parallel to the existing system to replace the old one step by step. This strategy allows a seamless intervention if a problem of the programme makes this step necessary. The testing can be divided into the white-box and black-box tests. The first one will be normally performed during the implementation regarding details of the implementation. The black-box tests can be done by using dummy-emails and configurations to test the correct over-all functionality of the programme.

The last step is the **maintenance**. Of course, it is not an immediately following step in the development process. However, normally it is very important for companies to offer support and maintenance to the customer for continuing customers' satisfaction and of course to earn additional money. In this project, maintenance will be not considered, as it is an internal completed project. Money will be not invested for further maintenance.

The **documentation** of the project will be written during the whole development process to keep it actual.

5.1 Phases of the development

This chapter describes shortly the phases of the development and their aims.

5.1.1 Requirements

The requirements will be determined to show the real needs of the software.

5.1.2 Analysis

The analysis phase will start with the use-cases deduced from the requirements. Therewith the borders of the system can be found and hence, the architecture can be build. The programme will have a rule-based behaviour to react on cases specified by rules. Therefore, mechanisms have to be considered to fulfil this feature. It has to be thought about an increasing email load in the future as well.

Which kind of technologies should be used? How can rules be defined in an efficient way? And so on.

5.1.3 Design

In the design phase the fine-grained structure of the programme will be determined by classes, interfaces and their methods.

5.1.4 Implementation

During the implementation phase the analysis programme will be implemented.

5.1.5 Integration and Test

The programme has to be integrated into an existing backup evaluation process. Thus, it will be running in parallel to the normal evaluation during the test phase. White-box tests will be made during the implementation. Black-box tests will show the behaviour of the programme during normal usage.

5.1.6 Maintenance

Normally maintenance is a very important topic of any kind of software, because companies often earn the most of their profits by selling maintenance to their products. As this programme is only a development for internal use, maintenance will be not considered in this project.

5.1.7 Documentation

The documentation will be written during the whole development process.

6 EmailParser

The following chapter describes the development of EmailParser regarding to the last chapter. It will show the development steps from the requirement, analysis, design, implementation and integration to the test.

6.1 Requirements

The analysis programme EmailParser has to meet the following requirements.

1. A programme has to be developed, which has to be executable on workstations running Sun Solaris or Windows NT, as these are the operating systems used in the department.
2. The programme has to receive emails sent by the backup servers to a central email address containing the backup status information.
3. The received emails have to be saved persistent for later usage or control of specific backups.
4. The programme has to be able to forward pre-defined emails immediately to the administrator without any further steps except requirement three.
5. The programme has to evaluate the emails whether the backups were correct or not. The base of the decision must be configurable in a comfortable way.
6. The programme should evaluate the emails for backup information. Errors and warnings are backup information as well as all other essential information like backup server, date and tape label.
7. The backup information has to be saved to an Oracle database.
8. Emails containing information about backup errors, warnings or other not usual backup information has to be forwarded to an email address.
9. The following settings for the programme have to be configurable without modifying the source code:
   - POP3 server information like host, account and password
   - Forward email address and SMTP server host
   - Database information like server host, server port, profile name, login name and login password to the database
   - The evaluation rules
10. Data security and failure tolerance has to be considered. The backup information is very important to recover and confirm correct backups. If the programme fails during the evaluation of the emails, the complete backup information must not be lost.
11. The kind of database server and email server should be exchangeable without changing the whole programme.

12. An increasing amount of emails in the future should be considered. On the same subject, parallelism and scalability should be considered as well.

After defining the requirements to the programme, they will be analysed. This will be done in the next chapter.

6.2 Analysis

This chapter will discuss the analysis. During the analysis, the requirements will be reviewed to show how they can be met. The discussion will show that different solutions can solve specified problems. The result of the analysis is the architecture of EmailParser, which describes the structure.

6.2.1 Use-cases

The following chart shows the use-cases of EmailParser. Each use-case identifies a main task of the programme, which is external influenced – in this case by the user.

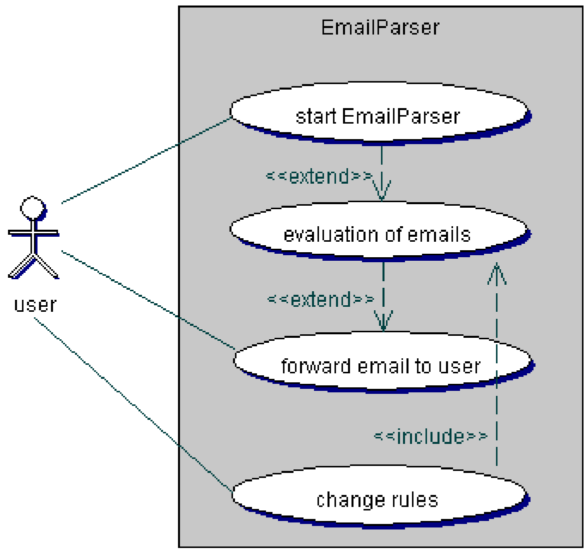

Figure 4 – Use-Cases of EmailParser

<u>1. Use-Case:</u> Start EmailParser

In fact, this is the main use-case of the EmailParser. The user has to start the EmailParser to activate the other use-cases.

<u>2. Use-Case:</u> Evaluation of emails

After starting the EmailParser, the emails will be evaluated. This use-case is dependent on the first use-case.

<u>3. Use-Case:</u> Forward emails to user

The emails can be forwarded to the user by specific rules. This use-case is dependent on the second use-case, as the emails have already been evaluated before they will be forwarded to the user.

<u>4. Use-Case:</u> Change rules
Changing the evaluation rules by the user affects the results of the evaluation of the emails and therewith the second use-case.

6.2.2 Sequence of the evaluation of the emails

Regarding to the existing system and some of the unequivocally requirements, a flow chart of EmailParser will be created (see next figure). This flow chart is just a first assumption of the possible sequence of steps, email parser will have to do. It does not regard to all requirements or the results of the analysis, but will be helpful during the analysis.

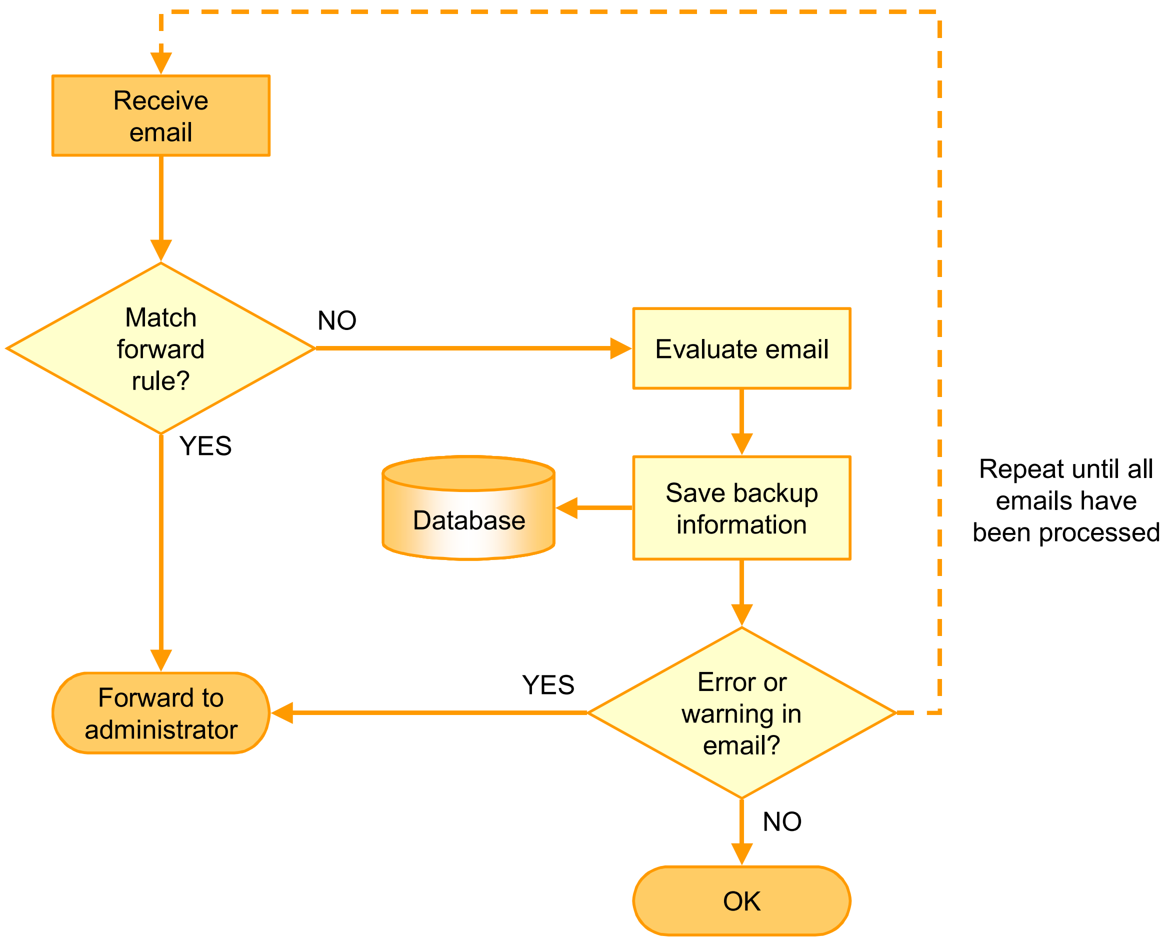

Figure 5 – Assumed flowchart of EmailParser

In the first step, EmailParser will receive the emails from the email server. Thereafter, it will check, whether emails match the pre-defined emails and hence have to be forwarded or not. These pre-defined emails will be forwarded to a defined email address. The other emails will be processed one after another by evaluating them by defined evaluation rules whether problems during the backup have occurred or not. All backup information will be saved to a database in the next step. If a problem has been found in the email, it will be forwarded as well.

6.2.3 Rule-based behaviour

EmailParser will have to decide on specific cases. Hence, rules have to be specified. In other words, EmailParser will have a **rule-based behaviour**. Rule-based behaviour means that a programme will change its behaviour on specific rules. This feature is a main aspect of the project.

As defined in the requirements the rules should be changeable from extern without editing the source code of a programme. In fact this technique saves a lot of money and resources and the programme becomes much more flexible. In this chapter, rule-based behaviour in principle will be described as well as techniques to define the rules in an extern configuration file.

6.2.3.1 Behaviour and intelligence

Talking about behaviour of a programme presupposes a kind of intelligence. Hence, we have to distinguish between the ability to make a decision coursed by a **pre-defined rule** and the ability to make a decision coursed by **implicit knowledge**. The first ability is just a hard-wired attitude and so a very restricted intelligence. The second ability indeed means that the programme has to have knowledge about **what** it is doing and has to be able to **learn** from its mistakes, i.e. the programme is able to define and change its own behaviour rules. In other words, the programme contains **artificial intelligence** (AI).

In fact, artificial intelligence is a very complex science and not the real aim of EmailParser. EmailParser should be able to react in a specific way on complex conditions, but needs not to learn. Thus, its behaviour can be hard-wired in the programme. Indeed the decision-changing statements and the reaction have to be transferred from the programme into the rules to make them configurable from extern. This will be shown in the next paragraphs. First, we have to define how programmes can handle rules.

6.2.3.2 Rules

Generally, each programme has a kind of rule-based behaviour, because each IF – ELSE command is nothing else but a decision based on a rule (also called **forward chaining** or **data-driven**):

```
IF < condition > THEN <action>
```

The opposite is the **backward chaining** or **goal-driven** rule, i.e. the circumstance will be proved after the consequence has been already defined. For example, a consequence will be repeated until a condition is true:

```
< consequence > IF < circumstance >
```

A characteristic feature of a good rule-based system is not only the decision between YES and NO, but the **perhaps**. The system should be able to weight among different rules to make a decision if two or more rules clash with each other. [Cimo00] describes a weighted rule as:

```
IF < antecedent > THEN < consequence > [weight]
```

If a specific antecedent has occurred then the programme should perform a predefined consequence. Optional the rules can be weighted so the programme can additionally decide among coincident rules because of their weighting factor. An example will be shown in the next chapter.

Another presupposition to get a complex handling rule-based system is the ability to interlace different conditions in the antecedence block and actions in the consequence block of a rule. Of course, rules have to be grouped as well. Interlacement of rules, conditions and actions can be realized with Boolean operations. DeMorgan's laws [Haeb90] allow easily recovering Boolean expressions. Hence, this technique will be used.

As it is a requirement to make the rules configurable without changing the programme, they cannot be defined within EmailParser. Thus, the rules must be stored in an easy accessible way like in an extern file with XML. The realization of rules with XML in an extern file will be shown in the next chapter on the basis of an example.

6.2.3.3 Rule definition with XML

XML (Extended Markup Language) [Behm00] along with DTD (Document Type Definition) is a possible way to declare rules. The DTD is the declaration how the rules have to be written in XML. The main advantages are the facility to declare own tags and the platform independency. Furthermore, XML and DTD are standards and therewith many tools are available to handle and parse XML documents.

The following DTD defines that rules, conditions and actions can be concatenated each for themselves with the Boolean operators AND, OR and NOT and each rule is composed of an antecedence and its consequence. The Boolean combined conditions have to become true to actuate a consequence. Each rule, Boolean operator and condition can contain a weighting factor to be weighted against another one. Actions cannot be weighted, as they have to be performed or not.

```
<!ELEMENT Rules (AND+ | OR+ | NOT+ | Rule)>
<!ELEMENT Rule (Antecedence, Consequence)>
<!ATTLIST Rule
     Name CDATA #REQUIRED
     Weighting CDATA #IMPLIED
>
<!ELEMENT Antecedence (AND+ | OR+ | NOT+)>
<!ELEMENT Consequence (AND+ | NOT+)>
<!ELEMENT Condition EMPTY>
<!ATTLIST Condition
     EmailPart ( SUBJECT | CONTENTS | SUBJECTCONTENTS | SENDER |
                 SENDDATE | ADDRESSEE) #REQUIRED
     Term ( CONTAINS | STARTSWITH | ENDSWITH | EQUAL | NOTEQUAL |
            LESSTHAN | GREATERTHAN | LESSOREQUAL |
            GREATEROREQUAL) #REQUIRED
     CompareValue CDATA #REQUIRED
     Weighting CDATA #IMPLIED
>
<!ELEMENT Action EMPTY>
<!ATTLIST Action
     Do (FORWARD | SAVETODATABASE | SAVETOFILE | DONOTHING) #REQUIRED
     Destination CDATA #IMPLIED
>
<!ELEMENT AND (AND+ | OR+ | NOT+ | Rule+ | Condition+ | Action+)>
<!ATTLIST AND
     Weighting CDATA #IMPLIED
>
<!ELEMENT OR (AND+ | OR+ | NOT+ | Rule+ | Condition+ | Action+)>
<!ATTLIST OR
     Weighting CDATA #IMPLIED
>
<!ELEMENT NOT (AND+ | OR+ | NOT | Rule | Condition | Action)>
```

Table 1 – Example DTD for a rule definition

The following XML example based on the DTD above illustrates a rule declaration and how to use the weighting factor.

```
<Rules>
  <Rule Name="Error">
    <Antecedence>
      <AND>
        <OR Weigthing="100">
          <Condition  EmailPart="SUBJECTCONTENTS"
                      Term="CONTAINS"
                      CompareValue="error"/>
          <Condition  EmailPart="SUBJECTCONTENTS"
                      Term="CONTAINS"
                      CompareValue="warning"/>
        </OR>
        <NOT>
          <Condition  EmailPart="SUBJECT"
                      Term="CONTAINS"
                      CompareValue="information"
                      Weigthing="100"/>
        </NOT>
        <Condition  EmailPart="ADDRESSEE"
                    Term="EQUAL"
                    CompareValue="server1"
                    Weigthing="50"/>
      </AND>
    </Antecedence>
    <Consequence>
      <AND>
        <Action Do="FORWARD"/>
        <Action Do="SAVETODATABASE"/>
        <Action Do="SAVETOFILE"/>
      </AND>
    </Consequence>
```

```
   </Rule>
   <Rule Name="Second Rule">
     …
   </Rule>
</Rules>
```

Table 2 – XML example for a rule with weight

In the first step the weighting factors will not be used. The consequence of the rule is to forward the email, save it to the database and as a file. Thus to actuate the consequence, the following conditions have to be met within the rule named “Error”: The email’s **subject or contents** has to **contain** either the word **error or warning and** the email’s **subject** must **not contain** the word **information** (i.e. the mail is not a backup status mail) **and** the email’s **addressee** must be **server1**.

Now the weighting factors will be taken into consideration. Hence, the global definition has been made that the antecedence will be only met if the sum of all weighting factors of true conditions is greater than or equal to the sum of all weighting factors of false conditions:

$$\sum weighting_{true} \geq \sum weighting_{false} \Rightarrow Condition\, true$$

That means for this example, if the email contains the words error or warning and the email’s subject doesn’t contain the word information it doesn’t matter whether the email’s addressee is server1 or not. The antecedence will be fulfilled because the sum of the weighting factors of the true conditions is already 200 and therewith in any case greater than the sum of the false condition (0 or 50).

On the other hand, if the email contains the words error or warning and the email’s subject contains the word information the addressee condition weighs down the balance whether the antecedence will be fulfilled or not.

In other words, this rule means: Mails containing errors or warnings have to be forwarded, saved to database and to disk, except if they are information mails. Error or warning mails from server server1 have to be forwarded in any case.

Without weighting this simple example, we would need 18 instructions in contrast to 10 instructions (compare Table 2 and Table 3). This is already a saving of about 45%. Converting the Boolean expression of Table 2 by DeMorgan’s laws reduces the amount of instructions to 12, which means a saving of about 17%.

```
<Antecedence>
  <OR>
    <AND>
      <OR>
        <Condition "error"/>
        <Condition "warning"/>
      </OR>
      <NOT>
        <Condition "information mail"/>
      </NOT>
    </AND>
    <AND>
      <OR>
        <Condition "error"/>
        <Condition "warning"/>
      </OR>
      <Condition “server1”/>
    </AND>
  </OR>
</Antecedence>
```

Table 3 – Simplified XML code for the example rule without weighting

The disadvantage of the method with weighting is the complex calculation of the weighting factors. The cost of the calculation is only warrantable if the savings are big enough. Simply changing the rules is not that easy as well, because the weighting factors have to be recalculated each time. As EmailParser has only to check the mails for keywords with some exceptions, the supposed reduction of instructions for the rule definition with weighting will be not big enough to vindicate the enormous cost of their making. Hence, the weighting will be not used in the prototype. Nevertheless, the possible later usage of XML defined rules with weighting will be taken into account in the design.

6.2.4 Failure tolerance and reliability of data

Backups are one of the safest ways to prevent lost of important data. At least equally important is the carefully handling of the necessary information to restore the backups. These information contain for example the tape label of the backup tape, if necessary the master tape label or other tape labels, if the data has been saved on different tapes. The time of the different backups, the backup files, the directory tree and other information are important as well. All these information are called sensitive information.

Working with sensitive data as this, mechanisms have to be taken into consideration to prevent lost of them. On the same subject, failure tolerance has to be taken into account for the system itself. Hence, a main topic of the project EmailParser is the failure tolerance and reliability as well as the persistence of the backup status information as demanded in the requirements.

This chapter will show the requirements security and integrity of sensitive data. It will discuss the analysis of critical points and their prevention, which have to be taken into account in the design of the failure tolerance of the programme. The requirement persistence will be discussed in chapter 6.2.5.

6.2.4.1 Data integrity and data security

In [NISS92] data security is defined as:

> *Data Security: "[The] protection of data from unauthorized (accidental or intentional) modification, destruction, or disclosure."*

This chapter will give a short overview about the integrity and security requirement on working with sensitive data like the backup status information.

In general, if we work with security on data we have to decide between three kinds of security:

1. Access protection
2. Data integrity
3. Protection from loss

Access protection is not a requirement to the project, so it will not be taken into consideration in the analysis, but will be shortly dipped in the next chapter just for completion. However, the prevention of change of data and the protection from loss of data are very important topics to the project. These points will be discussed in the next paragraphs as well.

Access protection

Access protection prevents unauthorized access to data. To allow a user to access data, he has to be authenticated. **Authentication** takes place by **identification** – i.e. the name or description of the user, whereby he is known in the system – and **verification** – i.e. the validation that the user is indeed the person he declared to be. The identification of the user will be done during the **security auditing**. After the authentication of the user the system has to make the decision about whether the user has the rights to access the data or not and to control the access by control mechanisms – also called **access-control** by **authorization**. In other words, the user is trustworthy. Therewith the **confidentiality** is given, i.e. data will be only imparted to users, who are allowed to access the data. Access control can be differenced between **rule-based** and **identity-based access**. In a system with an identity-based access concept the access will be granted because of the identity of a user. In a rule-based access system, however the access will be granted because of a certain rule of the user. The administration of the access rights in the last concept is easier, because changes do not have to be done for each user but only in the rule. In fact, security concepts like the access-control must not prevent the **availability** of data. To reconstruct possible malfunctions in the access protection all data access has to be **logged**.

In contrast to the next chapters, this topic is not important to the project, as it is not a requirement. Thus, it will not be taken into consideration of the analysis.

Data integrity

Data integrity means the prevention of illegal change of data like during the transfer. Hence, a few points like reliability and integrity have to be fulfilled. **Reliability** means that the data is exactly the one, which has been stored or sent. With **integrity**, the completeness of the data is assuredly. The user is accountable for his act and has to give account for it if necessary (**accountability**).

In the project, it is not required to prevent the change of the data by a third party, but the unaware change during the transfer from the email server or to the database server. Hence, the integrity of the data can be obtained by building a CRC (Cyclic Redundancy Check). This means that a checksum will be calculated over the data to check whether the data are the same before the transfer or not.

Protection from loss of data

As the backups cannot be restored without the necessary backup status information, it is very sensitive. Thus, the **protection from loss** of this data is one of the most important topics of the project.

Indeed this topic affects the requirement of persistence of the backup information as well, because one solution to protect data from loss is to hold the data persistent. Hence, this analysis will be done in the next chapter.

6.2.5 Persistence

The backup information has to be available at all events. As the data will be stored into a database, the persistence of the data is fulfilled if the programme ends as usual. Nevertheless, the backup data evaluated by the EmailParser has to be stored in a way that it is still available even after a malfunction of the programme before the data could be stored into the database. Therefore, EmailParser has to be analysed for critical sections at first, where the data may go lost.

6.2.5.1 The critical sections of EmailParser

To find the critical sections in the project, the flow chart described in chapter 6.2.2 has to be analysed. There are three critical sections in the principle flowchart of EmailParser where the loss of emails and with it the loss of the backup information can occur: During the reception of the emails, the storage into the database and of course during the evaluation of the emails (as shown in the next figure).

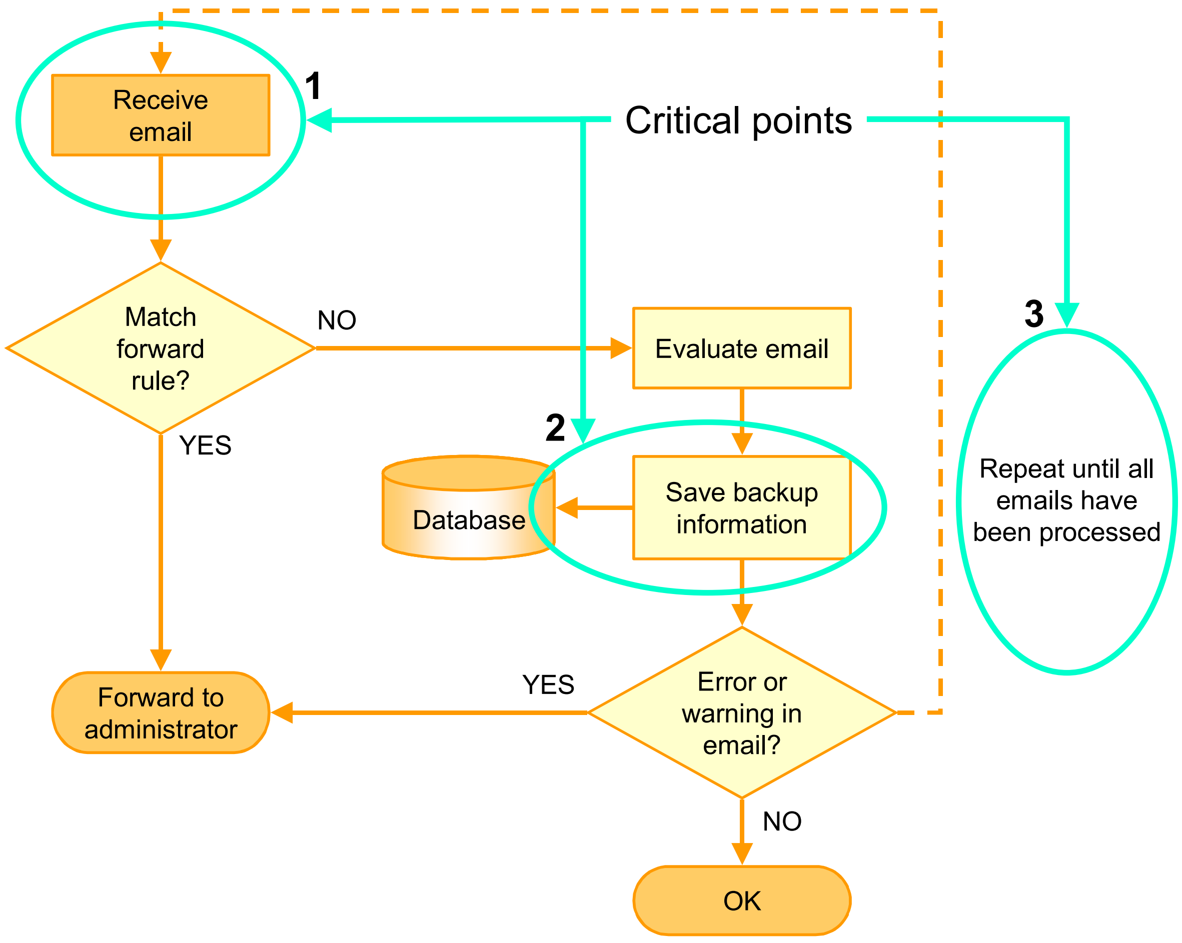

Figure 6 – Critical sections in the flowchart of EmailParser

Of course, the programme flow in principle is a critical section as well. If the programme crashes during the evaluation, the other protections for the data do not matter, because they cannot intervene without the runtime environment. Hence, the programme has to be developed in a very sensitive way like catching any kind of exceptions that the other parts of the programme are not affected. However, the critical sections shown in the upper figure have to be handled in a logical way. This will be discussed in the next chapter.

6.2.5.2 Precaution of the critical sections

The first critical section – the lost of emails during the reception of the emails – can be prevented by not deleting all emails from the email server until the last one has been received. This is an option, which standard email servers provide in general.

For the second one, there is no direct way to prevent the lost of backup information during the storage into the database, because if the programme crashes during the storage, the information has irrevocably gone. From this matter the decision has been made to safe the emails each as a text file to the hard disk. If the backup information is lost, they can be recovered from the saved text file.

The third critical section is much more difficult to handle, because the backup information must not be lost even if the programme crashes. The securest solution is to process only one email after another. This means that each email has to be received one after another, because if EmailParser receives all emails at once and crashes during the evaluation of an email the other not yet evaluated emails will be lost as well.

Of course, it is possible to receive the emails one after another from the email server. Unfortunately in the practice, the runtime of the programme would increase enormously. The **reception time** consists of the times to connect to, talk with – which means to send and receive handshakes and commands – and disconnect from the email server, which will be called **protocol time** in this context and the **transfer time** to transfer the emails. The transfer time of course is the same whether the emails will be received one after another or all together and consequently it will be not taken into account of the consideration. Therewith, the increment of the reception time will be caused by the protocol time and is about five to ten seconds each.

In a system with a small amount of emails like 50 to 100 each day, the time needed for the connections would not cause many problems. But as EmailParser has to be developed regarding to the nearest future, where the number of emails could be increase up to 1000 or more each day, a process time about five to ten seconds each email would rise the running time of EmailParser enormously and would foil the advantage even in an automated system.

Two possible solutions can solve this problem. The first one is a compromise, which is a mixture of the requirements and security aspects as discussed. Therefore, EmailParser will neither receive each email after another nor all emails at once but a configurable set of emails (as shown in the next figure). The emails will be processed and then the next set will be received until all emails have been evaluated.

The second solution is to execute some instances of the EmailParser in parallel. Each instance will receive only one email, evaluate it and store it into the database. Thereafter the email will be deleted from the email server and the next one will be received. The new requirements and resulting problems to parallelism and scalability will be discussed more detailed in chapter 6.2.6.

The first solution is simple to implement and adequate for the prototype, hence it will be used. Nevertheless, the second solution will be kept in mind for a future solution.

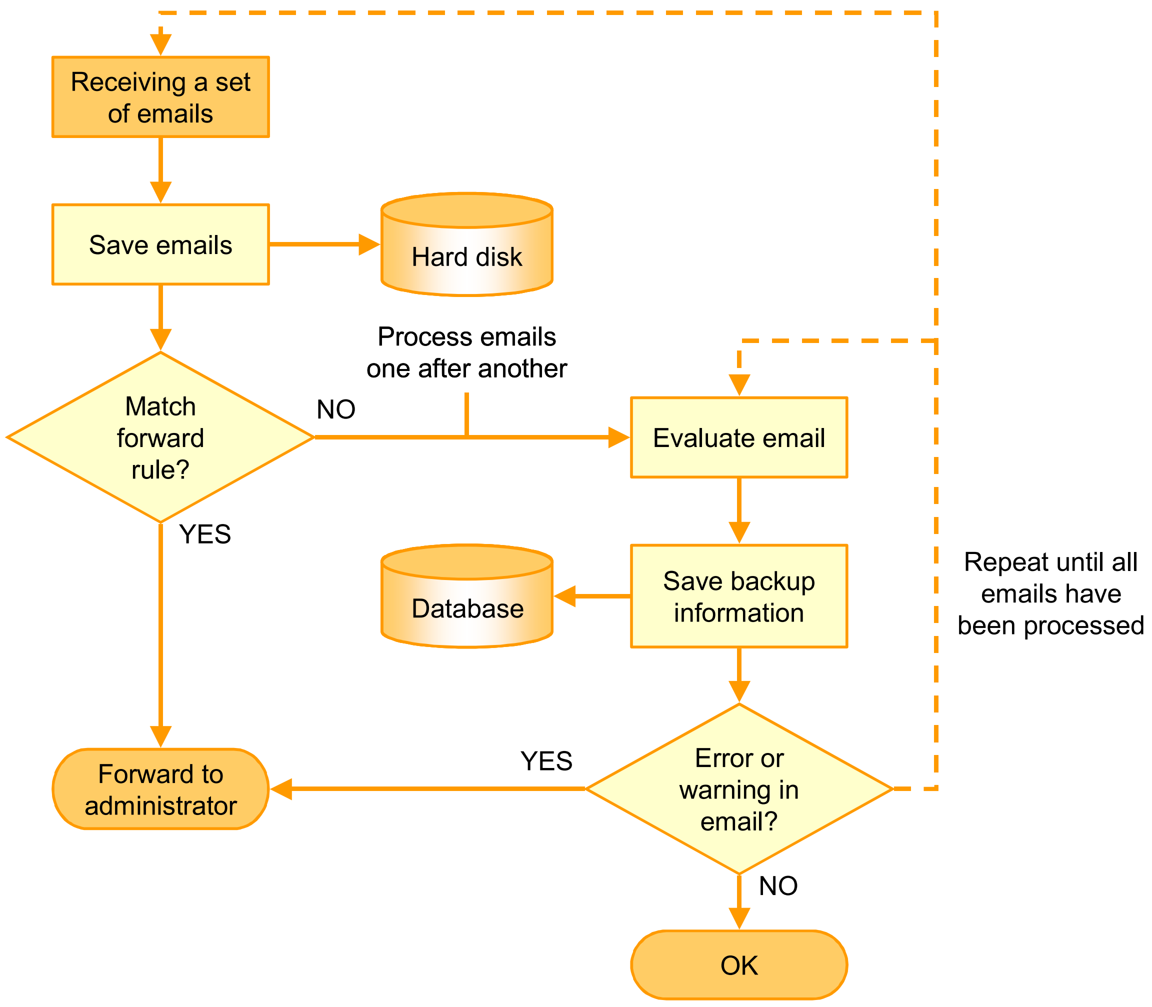
Receiving a set of emails
Save emails
Hard disk
Process emails one after another
Match forward rule?
NO
YES
Evaluate email
Save backup information
Database
Repeat until all emails have been processed
Forward to administrator
YES
Error or warning in email?
NO
OK

Figure 7 – The final flowchart of EmailParser

However, as a matter of course the database connection causes the same time problem as the connection to the email server. To write all backup status data into the database at once would mean to hold all backup status information from the emails in the programme – in other words non-persistent. If the programme crashes during the evaluation the information are lost. Thus, because of data safety the connection and transfer time to the database server must not be taken into account. The backup status information will be written immediately into the database.

To fathom possible problems discussed in the last chapters and therewith to increase the safety of EmailParser all steps during the process of EmailParser will be logged in a log file.

As already announced in the paragraphs above, we will have a look in the next chapter, how scalability and parallelism can be helpful with a high load of EmailParser.

6.2.6 Scalability and Parallelism

In chapter 6.2.5.1, we have seen that regarding to the future, EmailParser will have to handle a large number of mails. In that chapter, EmailParser has been optimised to handle the assumed amount of mails appearing in the nearest future. However, what would happen if the number of emails becomes very huge? This consideration will add some new requirements the software has to deal with:

- Performance: How can the programme handle a huge amount of emails? This is the most common requirement on software.
- Scalability: Is the programme or parts of the programme scalable to use the benefit of many computers or processors?
- Portability: Is the programme executable on computers with different hardware and operating systems?

To fulfil the additional requirements, parallel computing would be a usable solution. As EmailParser will be developed in the platform independent programming language Java, the point Portability will be fulfilled as long as a JVM (Java Virtual Machine) is available for the platform. However, the most important point is the Scalability. It is necessary to know whether more than one instance of the programme can be used in parallel or not. Hence, the parts of the programme have to be reviewed, which are affected by this problem. First, we have to know in which ways EmailParser can be run in parallel.

6.2.6.1 Scenarios

In the first scenario, EmailParser will be executed on different processors or computers and each instance is working standalone as shown in the next figure. For example, each instance receives 100 emails, evaluates them and writes the data into the database. Thereafter it will receive the next 100 emails until all emails have been evaluated. Of course, this would cut down the process time for evaluating all emails according to how many instances of EmailParser will be created.

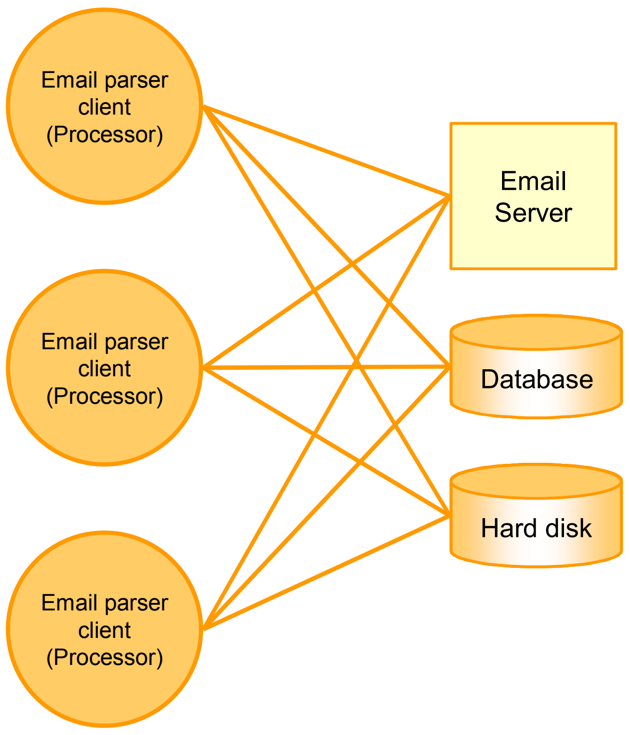
Email parser client (Processor)
Email parser client (Processor)
Email parser client (Processor)
Email Server
Database
Hard disk

Figure 8 – EmailParser running in parallel

Indeed the possibility of increasing the number of processors with instances of the EmailParser is finite. The restriction is effected by the email server, the database and the file server, which writes the email as a text file to a hard disk. The email server e.g. cannot serve more than one request at the same email account. Hence, the instances of EmailParser have to receive the emails sequential. It could happen that the first instance of EmailParser, which has received the first hundred emails, has already evaluated them, if it is the turn of the 50th instance to receive emails from the email server. However, in this case no specific part but the whole EmailParser has to be able to run in parallel.

Another way is to run the computer-bound parts of EmailParser in parallel (see Figure 9). In this case the EmailParser will receive emails and split of up to as many parsing or evaluation modules as emails have been received and distributes them to processors or computers. The evaluation modules will only return the results of the evaluation to the main module, which saves them to the database.

The advantage of this method is the increasing failure-tolerance of the system. If an email goes lost during the transfer or by a crash of the evaluation module the main module can detect this and will resend the email to another evaluation module. Of course this will not protect from the lost of data by a crash of the main module.

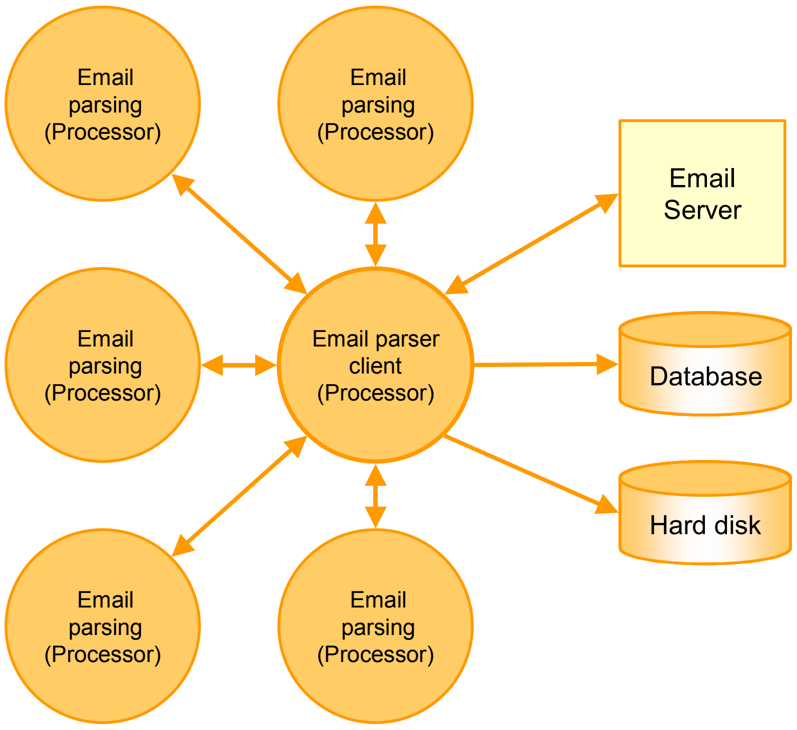
Email parsing (Processor)
Email parsing (Processor)
Email Server
Email parsing (Processor)
Email parser client (Processor)
Database
Hard disk
Email parsing (Processor)
Email parsing (Processor)

Figure 9 – Running the parsing of emails in parallel

As a matter of course, the number of instances of the parsing modules is restricted by the ability of the main module to handle the data.

A suitable solution is the usage of both parallel methods as shown in the next figure. That way both advantages would be used and the problems would be restricted. In the hybrid solution, a couple of instances of the whole EmailParser will be generated, which split off each some parsing modules to run the evaluation in parallel. The database and email server access will be shared by only some instances of EmailParser but the evaluation time will be cut down enormously. The failure tolerance of EmailParser will be increased as described above and the risk of loosing all data can be reduced by distribution on different computers.

As an enhancement, the instances of EmailParser could connect each other to transfer emails to another instances, which are not fully stretched. This kind of technique is also known as **load balancing**. As a matter of course, this would reduce the evaluation time of all emails again.

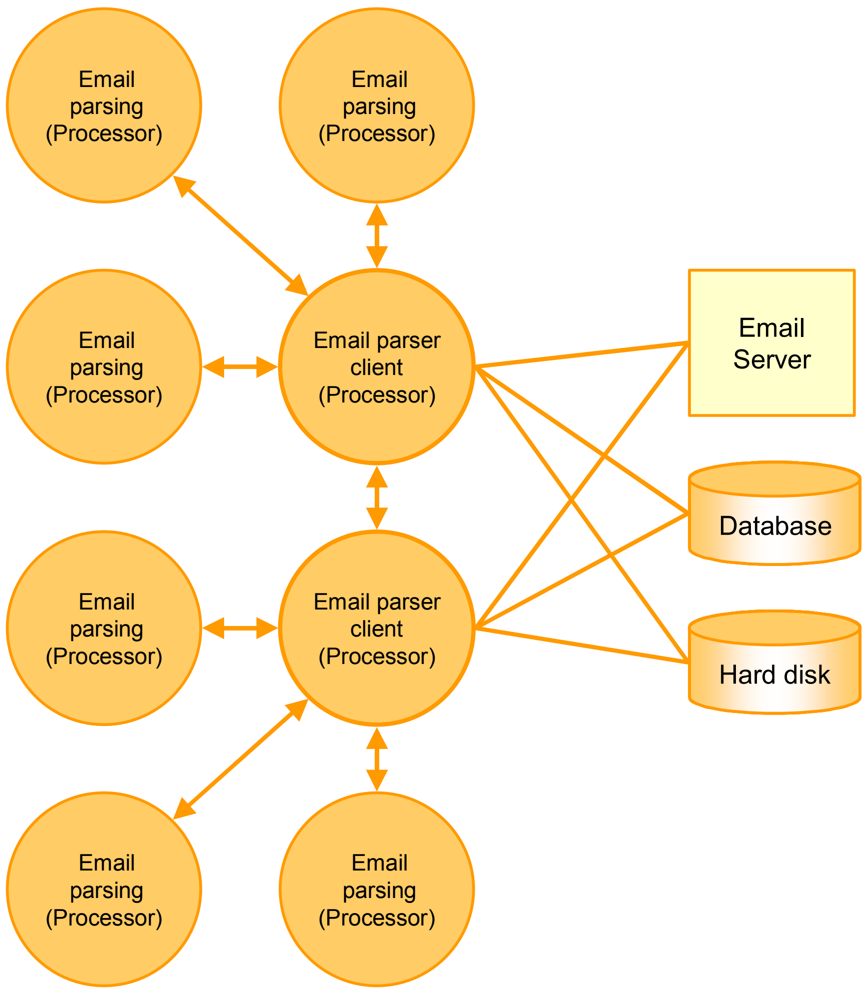
Email parsing (Processor)
Email parsing (Processor)
Email parsing (Processor)
Email parser client (Processor)
Email Server
Database
Email parsing (Processor)
Email parser client (Processor)
Hard disk
Email parsing (Processor)
Email parsing (Processor)

Figure 10 – Running EmailParser hybrid

Indeed it does not matter which kind of solution will be used, working in parallel causes some problems, which have to be taken into account. In the next chapter a typical problem will be described, which has to be solved if data is accessed from parallel processes.

6.2.6.2 Mutual Exclusion

The only critical points are to receive emails from the email server and to write data into the database. As shown above the email server cannot serve parallel requests but it can be blocked by a not properly closed connection. Hence, no other instance of EmailParser can open a connection to receive emails. The solution is to regard that the connection has to be closed immediately after working with the email server.

Likewise, parallel instances of EmailParser can cause problems during competing **transactions**. A transaction is the block from the beginning of the access to data until the end. An example with pseudo code shows this problem (see next figure).

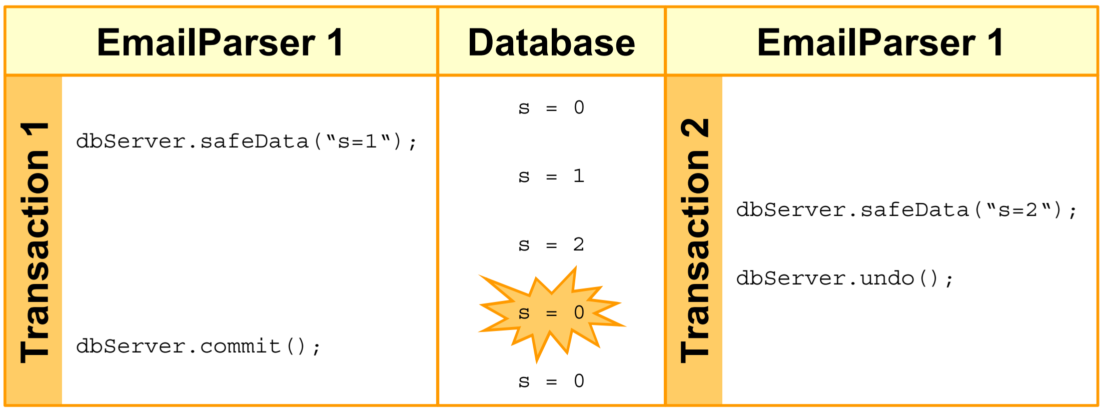

Figure 11 – Pseudo code example showing a competing access problem

Let us assume that two instances are writing the same data into the database. If the first one has not committed its changes yet and the second one undoes its changes, the changes of the first instance will be taken back as well. Hence, the database connection has to be designed to regard competing access on the same target. The solution for this problem is **mutual exclusion**. Mutual exclusion means that critical sections – like a transaction – are protected and will not be used by several processes in parallel. This can be done with **semaphores**.

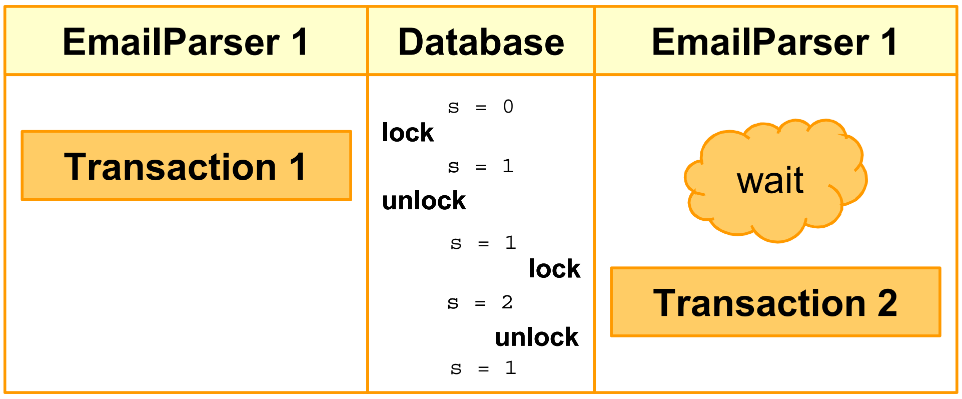

Figure 12 – Mutual exclusion with semaphores guarantees transactions

As Java will be used for the implementation, a monitor can be used to protect the methods. With the keyword `synchronized` a synchronized block is built around critical methods. Hence, in the implementation each transaction will be done in a synchronized block. Additionally each transaction will be explicit completed by the SQL command `commit`. Therewith the database will be forced to save the changes and the problem, as described above, will be avoided.

Considering the results of the analysis, the architecture can be designed, which will be described in the next chapter.

6.3 Architecture

Reviewing the requirements together with the results of the analysis shows that three different servers will have to be used. Two servers are already defined by the requirements: the email server and the database server. In chapter 6.2.5, the decision has been made to save the emails as a text file to the disk for persistence of the data. As the hard disk needs not to be local, a file server can be used to save the files as well. Hence, the list of servers will be expanded by adding a file server.

The following abstract figure shows EmailParser together with the three different servers: database server, email server and file server. Hence, the architecture will be a client-server architecture, also called 2-tier architecture.

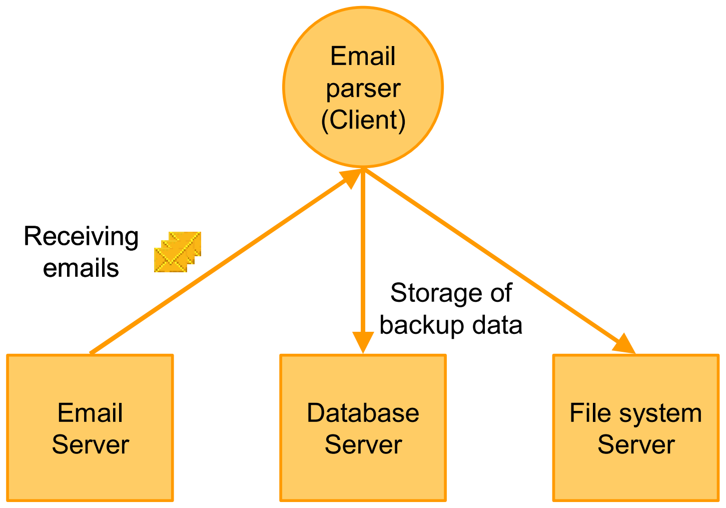

Figure 13 – Client-Server (2-tier) architecture of the programme

The current 2-tier architecture as shown above unfortunately has a big disadvantage. The following scenario will demonstrate the problem.

Let us assume there is an Oracle database in the system to store the backup status information. One year later, the database from Oracle has been replaced by a database from another company. The connection to this new database is completely different as the old one. This causes to change the programme completely to make the new connection possible. Indeed this has to be avoided as demanded in the requirements.

To prevent this problem, an application server will be used in the middle of the EmailParser and the three servers. An application server is a server keeping the logic of an application. The new architecture with the third level (see next figure) will be also called a 3-tier architecture.

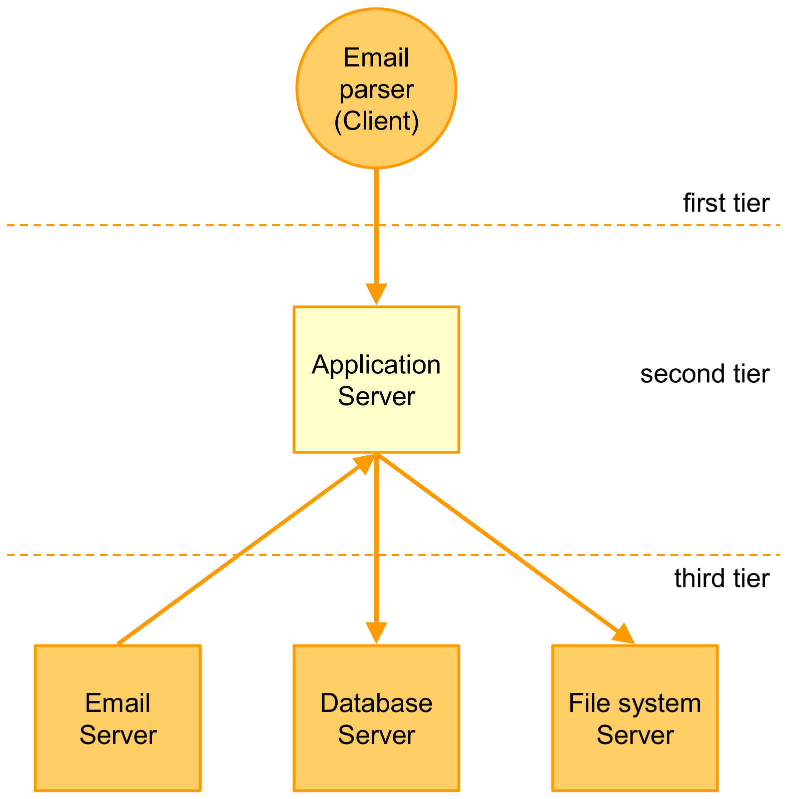

Figure 14 – 3-tier architecture of the programme

6.3.1 3-tier Architecture

In a 3-tier architecture, the application is divided into three tiers. The **first tier** is the client-layer responsible for the user interface to present data and acceptance of user input. The **third tier** is the data-layer responsible for the data attitude. This is usually a database server or could be a file system server or an email server like in this case as well. The new layer between the client and the data server – also called **second tier** – is the application server. As the client layer only handles the user interface and the data layer handles the storage of the data, the second tier is the most important layer in the 3-tier architecture, because it contains the whole logic of the application.

Using a 3-tier architecture, results in some advantages with a view to the requirements and the results of the analysis:

- A clear separation of the data and the logic as postulated by the object-oriented paradigm
- By separating the user-interface from the rest of the application, more clients are able to connect to the server application like required by a usage in parallel (see chapter 6.2.6)
- The modularity of the application makes the change of parts of the application much more easier, because it does not affect the whole application.

- The test phase will be much more easier as not the whole application has to be tested but small autarchic modules on their own
- The application will be distributable, so the client and the servers can be started on different computers

With the predefinition of the architecture, almost the whole analysis is completed. The following chapter will only shortly describe additional external techniques, which will be used in the project.

6.4 Additional used techniques

The following two standard APIs will be used in the project.

6.4.1 JDBC with an Oracle database

One of the specifications of EmailParser is to save the backup information into an Oracle database. To work with the database ODBC (Open DataBase Connection) from Microsoft will be used. This standardized API allows connecting to different databases with the aid of ODBC-compatible drivers in an easy way and uses SQL (Standard Query Language) for the data access. As the prototype will be realized in Java, the ODBC-based Java API JDBC (Java DataBase Connection) will be used [Sun99].

6.4.2 Java Mail API

To receive and resend emails, the Java Mail API version 1.1.2 with the POP3 API version 1.1 from Sun will be used [Sun01]. These APIs are simple to use as they are designed for the usage with Java. They fulfil all necessary requirements for the EmailParser as receiving emails from a POP3 account and resending them via SMTP as well.

Therewith the analysis is completed and the design phase of EmailParser can be started.

6.5 Design

This chapter will show the design of EmailParser. It describes the interfaces and classes and some sequence diagrams for comprehension. The diagrams are based on UML 1.3.

6.5.1 EmailParser

Figure 20 shows the important classes of the system. The class `EmailParser` is the main class of the programme.

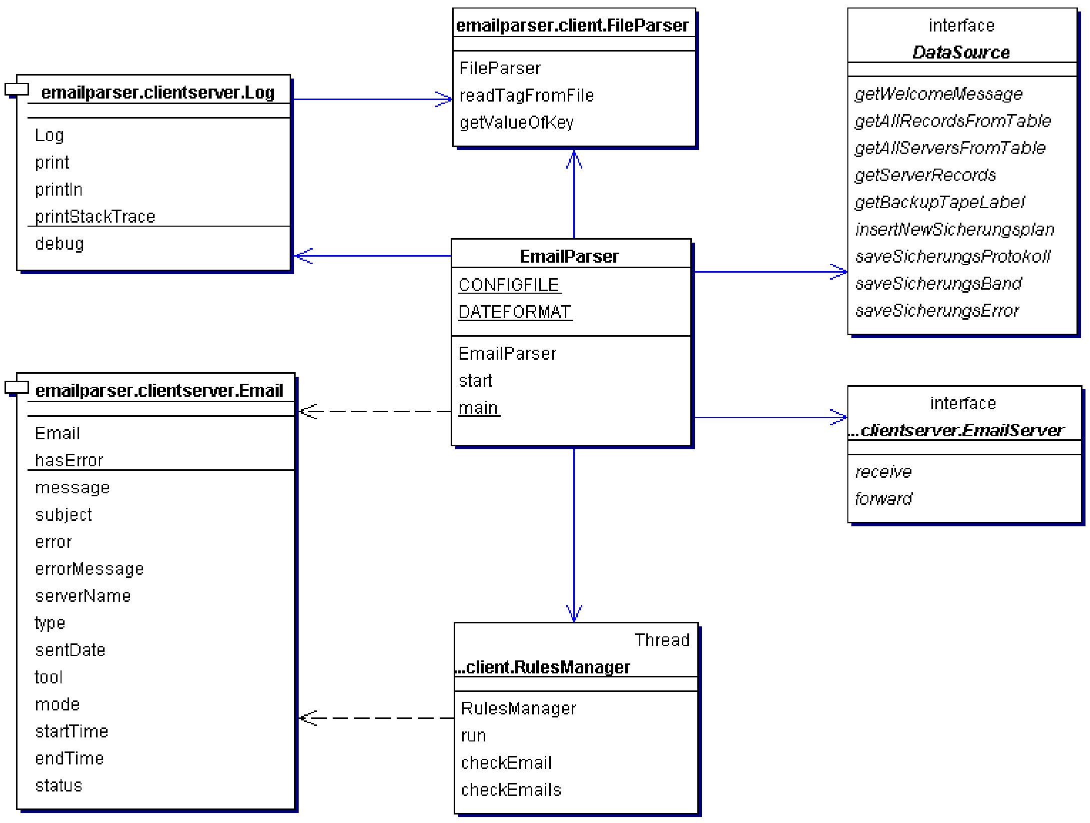

Figure 15 – Class diagram of EmailParser

As described in the end of chapter 6.2.5.2, all steps of EmailParser will be written in log files encapsulated by the class `Log`. The `FileParser` enables EmailParser to read the configuration file. This mechanism will be described in chapter 6.5.4. The mails from type `Email` will be received from a POP3 email server, which is accessible over the interface `EmailServer`. The connection to the email server will be described in chapter 6.5.3. After the reception of the emails, the `RulesManager` will evaluate the emails regarding the rules. The rules defined with XML will be parsed by an already existent extern standard XML-parser. The EmailParser will not get a result from the rules manager, but the emails will keep their own state. Hence, they "know" for example if they have to be forwarded or if they contain an error. Thereafter, EmailParser will save the state of the emails to the database over the interface `DataSource`. The connection to the database and the usage will be described in the next chapter.

6.5.2 Database connection with JDBC

The connection will be realized with a JDBC-ODBC bridge to the physical database. To make the connection and database exchangeable the following architecture will be used.

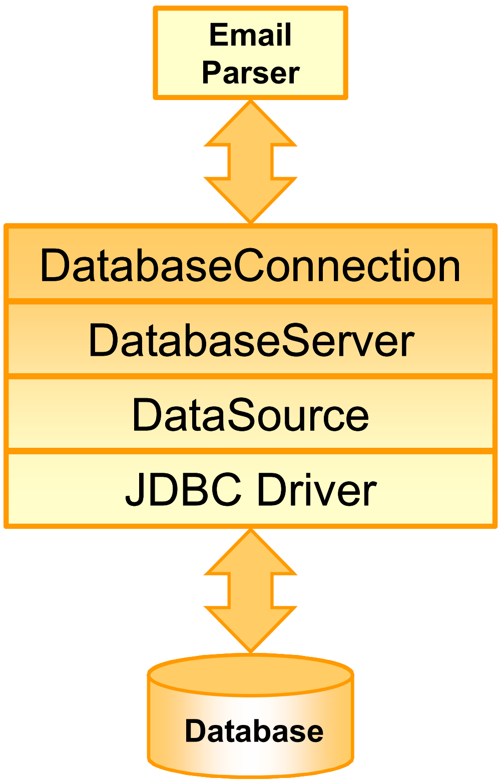

Figure 16 – JDBC Architecture

The architecture is composed of different layers. The first layer is the kind of connection, the second layer is the kind of database server and the third layer is the wrapper to the database itself. The JDBC driver is the adapter to the physical database. Each layer is exchangeable without affecting the other parts, because the layers are independent from each other. This will be shown in the following paragraphs as well as that EmailParser gets the connection to the database through these layers without knowledge of the real implementation.

The design of the database connection is based on the **interface pattern** [Gran98]. By this pattern, an object will be kept independent from the implementation of other objects, even though it uses their data and services. The other instances will be only accessed by interfaces. The interface pattern is related to the **delegation pattern**.

Using interfaces makes a **dynamic binding** possible. That means, the accessing object does not have to know and does not have to care about, which kind of real implementation is behind an interface. The object behind the interface can be set at run-time. Likewise, it is possible to replace objects with the same interface with each other. This **substitutability** is also called **polymorphism**.

The next diagram shows the class diagram of the database connection:

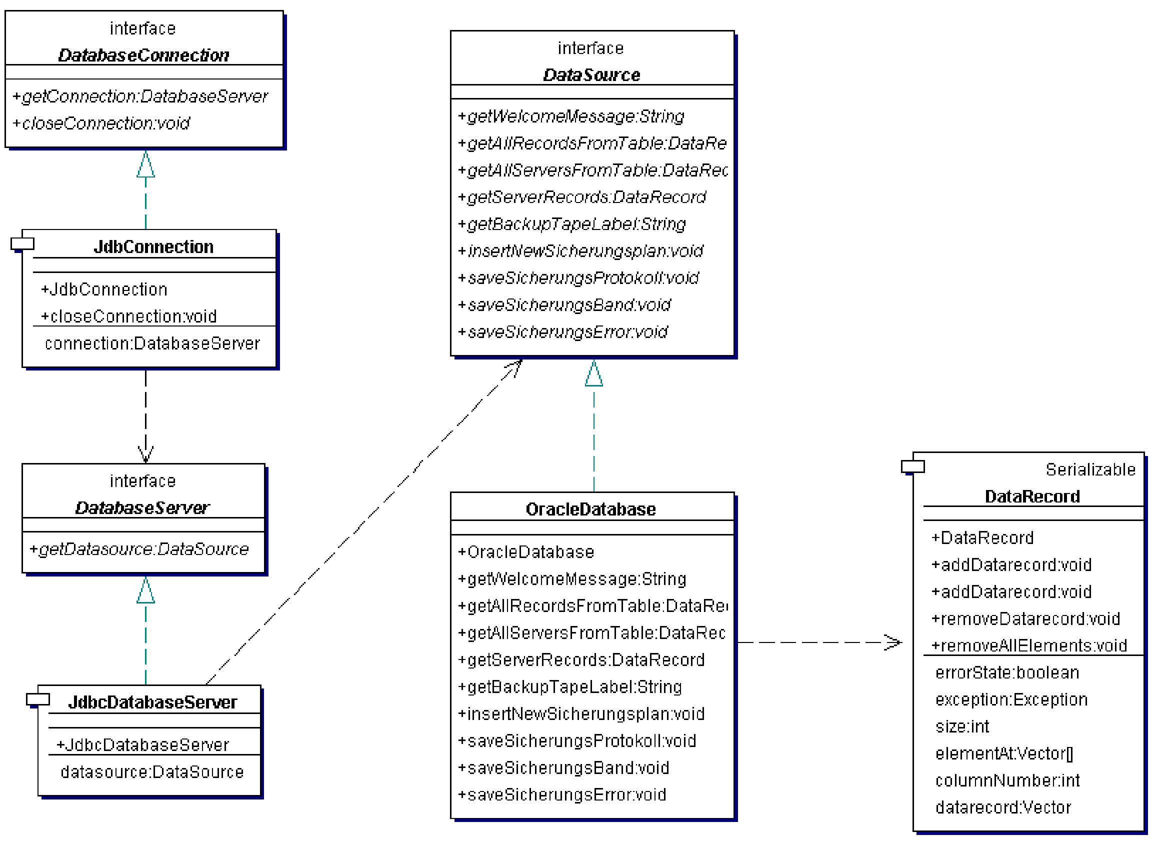
interface
DatabaseConnection
+getConnection:DatabaseServer
+closeConnection:void
JdbConnection
+JdbConnection
+closeConnection:void
connection:DatabaseServer
interface
DatabaseServer
+getDatasource:DataSource
JdbcDatabaseServer
+JdbcDatabaseServer
datasource:DataSource
interface
DataSource
+getWelcomeMessage:String
+getAllRecordsFromTable:DataRe
+getAllServersFromTable:DataRec
+getServerRecords:DataRecord
+getBackupTapeLabel:String
+insertNewSicherungsplan:void
+saveSicherungsProtokoll:void
+saveSicherungsBand:void
+saveSicherungsError:void
OracleDatabase
+OracleDatabase
+getWelcomeMessage:String
+getAllRecordsFromTable:DataRe
+getAllServersFromTable:DataRec
+getServerRecords:DataRecord
+getBackupTapeLabel:String
+insertNewSicherungsplan:void
+saveSicherungsProtokoll:void
+saveSicherungsBand:void
+saveSicherungsError:void
Serializable
DataRecord
+DataRecord
+addDatarecord:void
+addDatarecord:void
+removeDatarecord:void
+removeAllElements:void
errorState:boolean
exception:Exception
size:int
elementAt:Vector[]
columnNumber:int
datarecord:Vector

Figure 17 – Database connection with JDBC

The connection will be established with three interfaces. EmailParser does only have to know the interfaces beside the class `JdbConnection` for the connection. The SQL queries are wrapped in the class `OracleDatabase`. The results of the query will be returned as the `serialized` object `DataRecord`. The serialization of the object `DataRecord` is necessary, as it will be send over the network to the database server. The following sequence diagrams show how a connection can be established and how a database request will be performed.

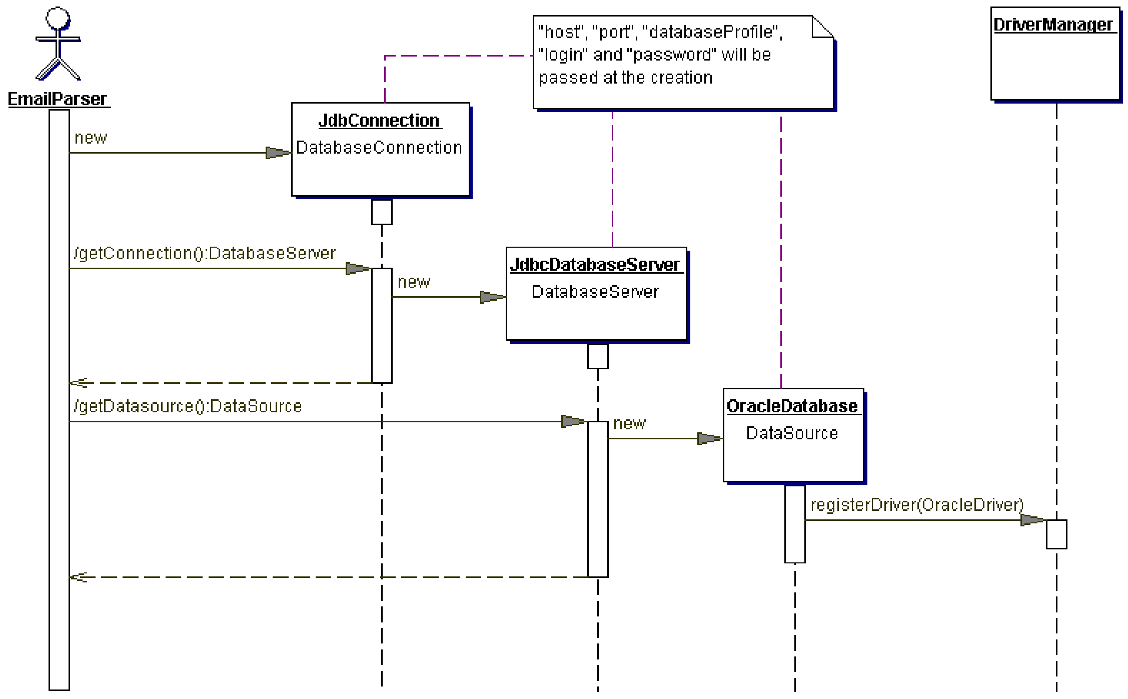
"host", "port", "databaseProfile", "login" and "password" will be passed at the creation
EmailParser
JdbConnection
DatabaseConnection
JdbcDatabaseServer
DatabaseServer
OracleDatabase
DataSource
DriverManager
new
/getConnection():DatabaseServer
new
/getDatasource():DataSource
new
registerDriver(OracleDriver)

Figure 18 – Connection to the Oracle database

To establish a connection to the database a new `JdbConnection` has to be created. Invoking the method `getConnection` on `JdbConnection` a new `JdbcDatabaseServer` will be created, which provides the real wrapper class (`OracleDatabase`) to the database. The calling object will only see the interfaces `DatabaseServer` and `DataSource`. During the creation of `OracleDatabase` the JDBC driver to the physical Oracle database will be registered in the `DriverManager` to make connections to the physical database possible. The following diagram shows how to save data to and to get data from the database.

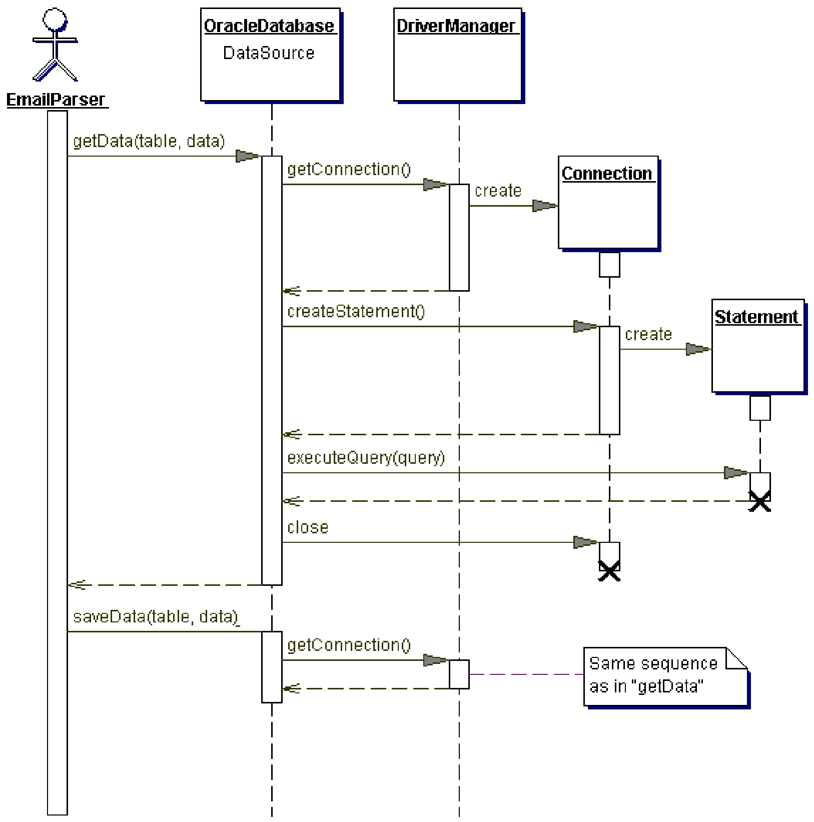

Figure 19 – Requesting and saving data to the database

To save data into or get data from the database the class `OracleDatabase` provides different methods, which invoke the method `executeQuery` to execute a specific SQL query on the database. These methods can be summarized to the general methods `getData` and `saveData`. If they have been invoked, `OracleDatabase` opens a new connection by calling the method `getConnection` in `DriverManager` from the JDBC-API. The connection is based on an URL in the notation `jdbc:oracle:thin:@[hostname]:[port]:[dbProfile]`. Each database request will be closed immediately after receiving the result set to prevent inconsistency in the database as described in chapter 6.2.6.2.

6.5.3 Email server connection

The connection to the email server is accessible over an interface as well. The next figure shows the class diagram.

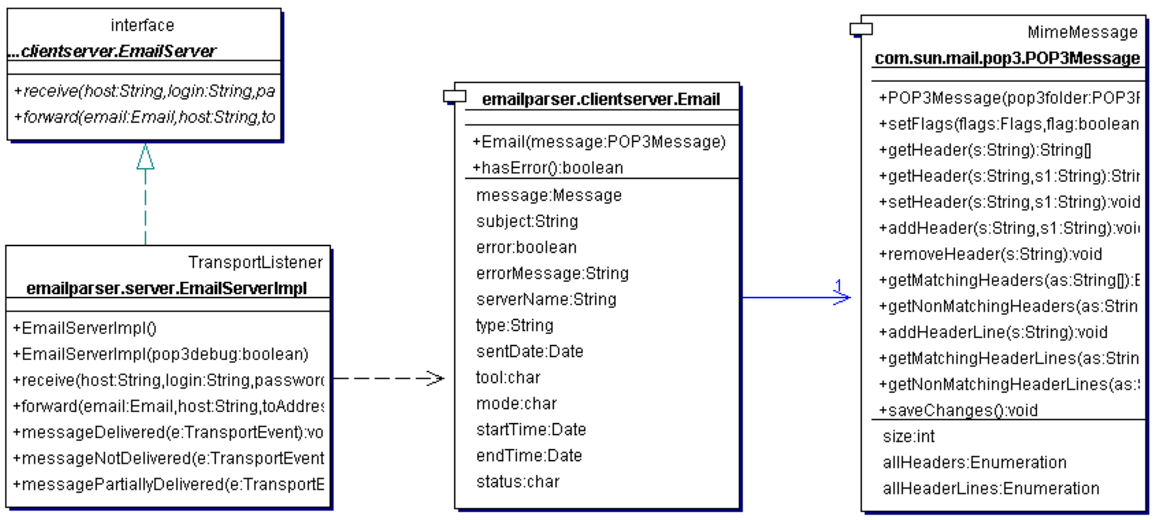
interface
...clientserver.EmailServer
+receive(host:String,login:String,pa
+forward(email:Email,host:String,to
TransportListener
emailparser.server.EmailServerImpl
+EmailServerImpl()
+EmailServerImpl(pop3debug:boolean)
+receive(host:String,login:String,passwor
+forward(email:Email,host:String,toAddre
+messageDelivered(e:TransportEvent):vo
+messageNotDelivered(e:TransportEvent
+messagePartiallyDelivered(e:TransportE
emailparser.clientserver.Email
+Email(message:POP3Message)
+hasError():boolean
message:Message
subject:String
error:boolean
errorMessage:String
serverName:String
type:String
sentDate:Date
tool:char
mode:char
startTime:Date
endTime:Date
status:char
1
MimeMessage
com.sun.mail.pop3.POP3Message
+POP3Message(pop3folder:POP3F
+setFlags(flags:Flags,flag:boolean
+getHeader(s:String):String[]
+getHeader(s:String,s1:String):Strin
+setHeader(s:String,s1:String):void
+addHeader(s:String,s1:String):voi
+removeHeader(s:String):void
+getMatchingHeaders(as:String[]):E
+getNonMatchingHeaders(as:Strin
+addHeaderLine(s:String):void
+getMatchingHeaderLines(as:Strin
+getNonMatchingHeaderLines(as:
+saveChanges():void
size:int
allHeaders:Enumeration
allHeaderLines:Enumeration

Figure 20 – Email server connection

Using the interface `EmailServer` makes the implementation of the email server wrapper `EmailServerImpl` and therewith the kind of physical email server exchangeable without changing the client. This is possible, because the client only knows the methods `forward` and `receive` to handle emails. The client does not care about, how the emails will be received or sent in reality.

The emails are from type `Email`, which instantiates the real kind of message `POP3Message`. Hence, the email server type and the kind of email conceals behind representatives. The client only works with `EmailServerImpl` and `Email`.

With this architecture, the requirement making the email server exchangeable will be fulfilled. The following sequence diagrams show how to connect to the email server and how to send or receive emails.

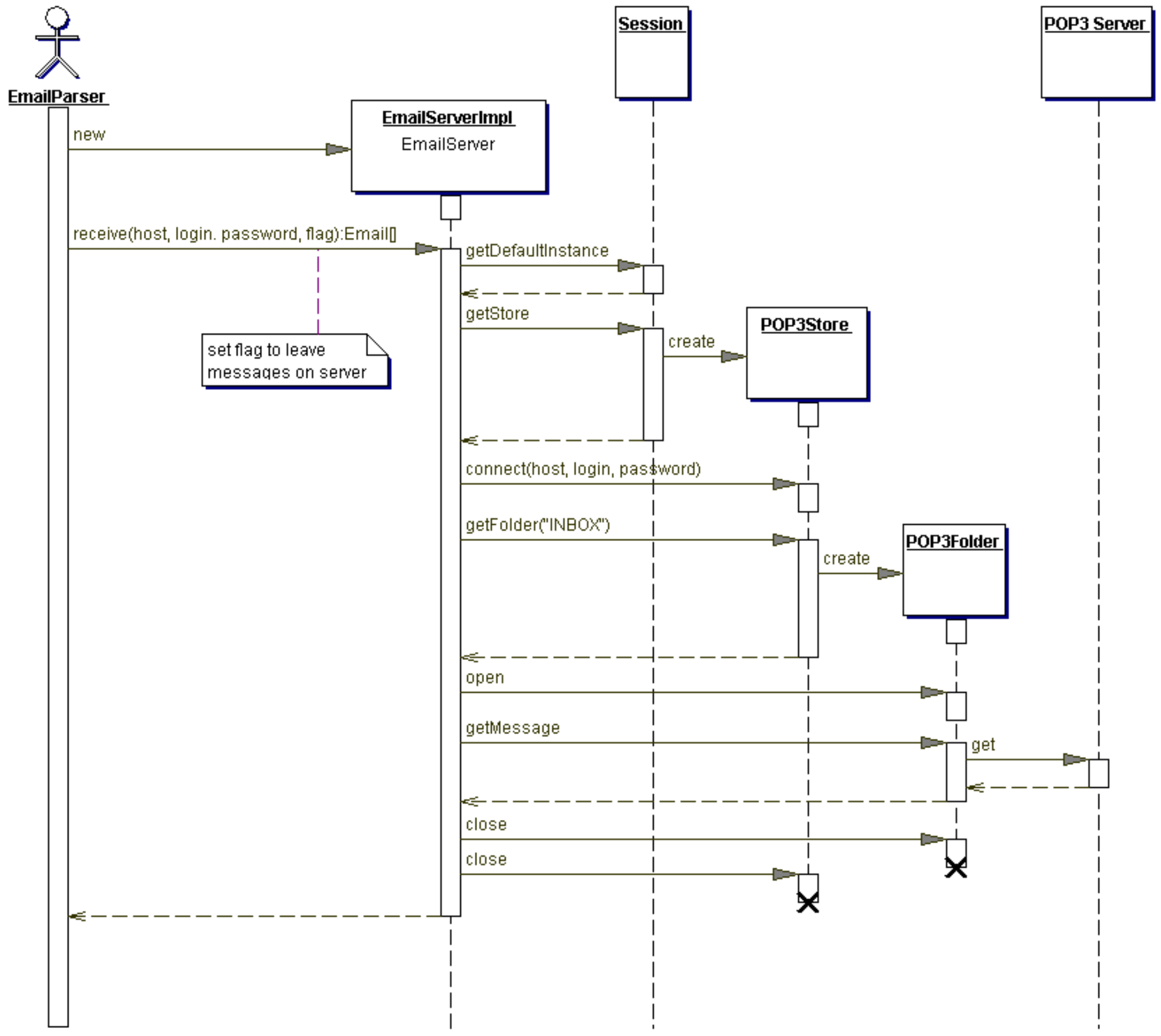

Figure 21 – Connecting to the email server and receiving emails

To receive or send emails (as shown in the next figure), EmailParser has to connect to the email server first. By invoking one of the methods `receive` or `forward` the necessary data to connect and to forward or receive emails will be transferred. Connection to a POP3 server offers the possibility to leave messages on the server. So, they will not be deleted after successful reception. This can be necessary to run tests repeatedly working with the same emails.

The Mail API [Sun01] uses the static object `Session` to connect to the mail store on the real email server. The store is wrapped in the object `POP3Store`. The desired folder is accessable by invoking the method `getFolder` together with the specific folder name – in this case the inbox folder. Different methods make it possible to handle the emails directly on the email server or to receive some of them or all at once. At the end, the folder and the store have to be closed to release the session to the email account for other requests.

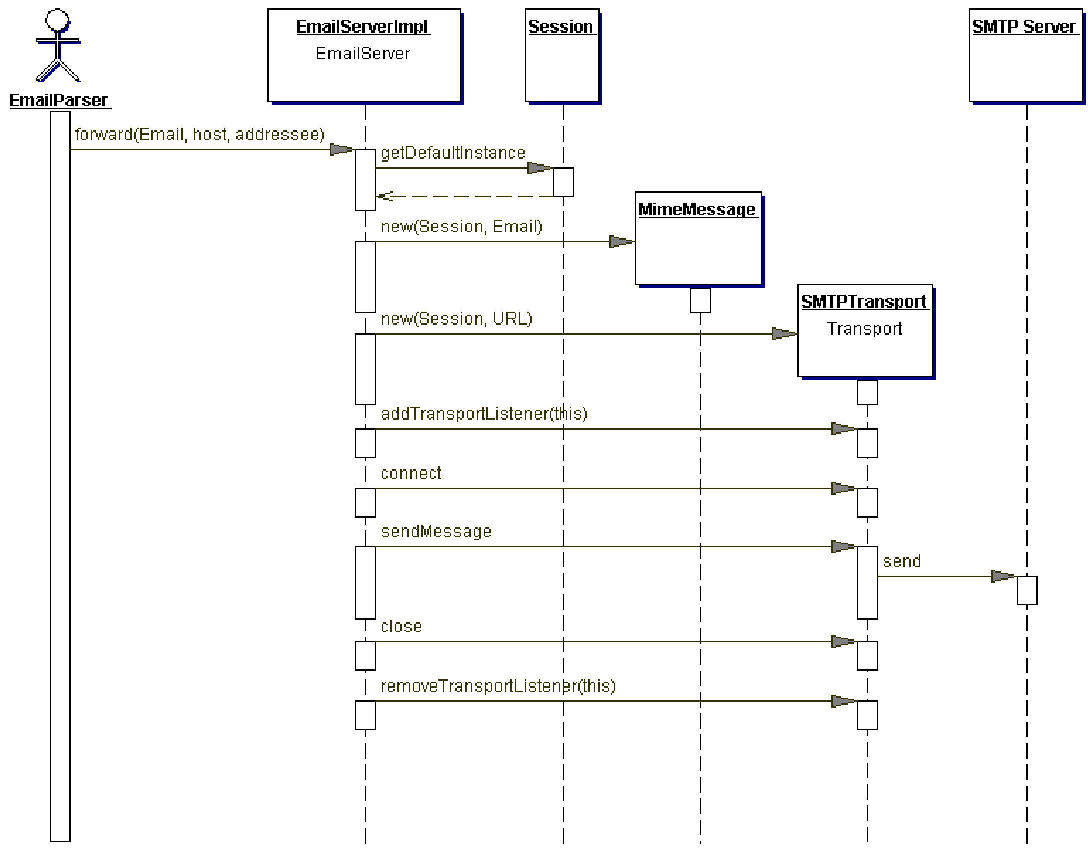

Figure 22 – Forwarding emails

The forwarding of the emails is nearly similar. The object Session connects to the physical email server. The connection will be established by transferring the SMTP data collected in an URL to the transport object `SMTPTransport`. The URL consists of the string `smtp://[host]:25`. Therewith, the transport object knows the SMTP server wherewith the message has to be sent. To know the sending state, like whether the transport was successful or not, the sending object can register itself as a `TransportListener` to the transport object. Hence, it is possible to react if the message could not be send.

6.5.4 Configuration

The configuration is stored in a configuration file. Tags are used to distinguish between categories like SMTP, POP3 or the JDBC data. This increases the clarity of the file. The tags are kept in squared brackets in the form: `[tag]`. Each tag is containing miscellaneous keys completed by a semicolon in the form: `key = value;`. Keys are for example the login name or the password. The next table shows an example of a configuration file.

```
[Tag1]
key1 = value;
key2 = value;
[Tag2]
key1 = value;
key2 = value;
```

Table 4 – Configuration file example

To get the configuration from the configuration file, the class `FileParser` will be used. The following sequence diagram shows how the keys will be accessed.

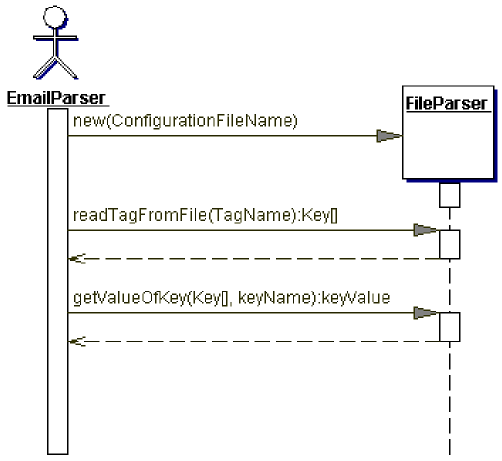
EmailParser
FileParser
new(ConfigurationFileName)
readTagFromFile(TagName):Key[]
getValueOfKey(Key[], keyName):keyValue

Figure 23 – Getting the configuration from the configuration file

During the creation of `FileParser`, the configuration filename will be set. Invoking the method `readTagFromFile` the keys of the tag will be returned in an array. To get the value of a specific key the method `getValueOfKey` will be used by transferring the array of keys and the name of the needed key.

After the design phase, the implementation of EmailParser can start. During the integration, tests have to be done, which will be described in the following chapter.

6.6 Test

The tests for EmailParser will be performed in two ways: the white-box test and the black-box test. The white-box test is made during the implementation. Small parts of the programme will be tested stand-alone by building small test units. The black-box test however is made to test the whole application and its functionality, whether reaction of the programme on different cases is the same as expected. This will be done for example by sending emails with specific contents. Aside of the functionality tests, especially load tests and failure-tolerance tests will be done. During these safety-related tests the following aspects are tested:

- Can the programme handle a large amount of emails and a bulk number of state backup information as well?
- Will there be problems if any of the configuration data is invalid (for example if a path does not exist)?
- May data be lost if any unexpected event happens, like an unreachable server?
- Is the data persistent, even if the programme crashes during the evaluation? Respectively are the critical sections as described in 6.2.5.1 really protected against a crash?

The last point is a little bit difficult to be tested, because unexpected programme crashes can be only assumed and simulated. However, the critical sections of the programme can be specifically tested for error handling. As the data will be saved to disk, before the programme starts the evaluation, the persistence is guaranteed in any case. Hence, it does not matter if the programme crashes or is running fine, because of the persistence of the data, the backup information will never be lost. However, the stability of the programme can be only tested by a long-term test.

During the tests of EmailParser, it could be seen that cases occurred, which had not been assumed. Hence, the rules for the evaluation had to be adapted. In addition, unexpected circumstances partly caused by extern influences like changing network paths had happen, which made it necessary to adjust EmailParser and the configuration.

During the simulated load tests, it could be seen that the handling of a large amount of emails and data caused a very long evaluation time. Hence, the considerations from chapter 6.2.6 will have to be taken into account, if the assumed email load will be exceeded enormously.

7 Conclusion and outlook

This dissertation shows the development of the programme EmailParser. The aim of the project was to automate an existing manual evaluation system of backup information sent by email done so far by human beings. The new programme should decide by dynamical modifiable rules whether backups were correct or not. As the backup information is very sensitive, because this data is necessary to recover backups, it had to be stored persistent.

Despite a forced break, the project could be successfully finished in time. The break was necessary due to a job change. EmailParser had been developed in Java, connected via JDBC to the Oracle database. The programme is working fine and has replaced successfully the old manual evaluation system.

The requirements persistence and failure-tolerance as well as the rule-based behaviour of the programme exposed to be the main problems. The problem of the first two points could be solved by reviewing the critical sections of the programme. These sections have been especially protected to meet failure-tolerance. To prevent from the lost of backup information, the backup data will be held persistent. The persistence has had to be ensured in any case, even if the programme crashes in an unexpected circumstance. Hence, the data will not only be stored to the database, but also to a hard disk before the evaluation – as a kind of emergency backup system. If the programme stops working before the data could be stored to the database, the data will not be lost.

The rule-based behaviour however was very difficult, because a concept had to be deliberated. It was not enough to compare the emails with defined keywords. Rather a system had to be developed, which uses rules for the evaluation in a complex way. This means that the antecedence and the consequence of a rule and the rules themselves had to be able to be combined. The combination has been realized by Boolean operations. To describe the rules and the operations, the standardized technology XML has been chosen, because of its high propagation. This makes it possible to use existing tools, to edit and to parse the rule XML file.

During the implementation of the database connection, a problem occurred with a version conflict between the JDBC driver and the JDK: Generated SQL dates were between 12/31/1999 23:59:59:999 and 01/01/2000 00:00:00:000 – in other words a not existing date. This problem was not so easy to find, because the dates in the textual output were correct. The problem only occurs during date operations like comparison.

Additionally to the EmailParser, a new programme will be developed, which will generate web pages to show the evaluated backup information stored in the database in a clearly presented way. This new programme can re-use parts of EmailParser like the database connection.

In the future, an increasing amount of emails might have to be handled by EmailParser. Hence, EmailParser or parts of the email parser like the evaluation module could be run in parallel on different computers in a later version. To handle complex situations, the current combination of the rules might not be adequate. The usage of the weighting factors of the rules might be a useful solution.

The next step from the automation of the backup evaluation system could be a mobile agent system. Thereby, autonomous small programmes, called agents, would be sent into the network to get the backup information directly from the servers. They could evaluate the data already on the server and if necessary react for example on errors by nudging another process. The agent could transport only the necessary backup information back to the start server without the redundant information or save the backup information directly to the database. This would lower the network traffic.

Another advantage is the representation of the agents as parallel processes. A huge amount of data could be processed in very short time. In addition, the agents are able to save their aggregate state during the whole journey through the network. Hence, they can transport for example a state list of the backups to the server where they have been started. Additionally, they can communicate with each other to exchange information as backup server addresses. Mobile agents could be useful, if a backup server system is scaling. This means, that the server addresses change all the time. The agents could have logic to find new servers and to inform the rest of the system about them. In the beginning, the start system would not have to know the backup servers in the system.

Appendix A

Used Software to develop EmailParser

The following software has been used for the development of EmailParser:

- Together Control Center 4.1 by TogetherSoft Corporation:
  UML diagrams for the analysis and design
- Visual SlickEdit 5.0b by MicroEdge Inc.:
  Implementation
- JDK 1.3 by SUN:
  Programming language for the implementation of EmailParser
- JDBC Driver Version 8.1.6.0.0 by Oracle:
  Database connection with Java
- Java Mail API 1.1.2 with POP3 API Version 1.1 by SUN
- Oracle Enterprise Edition 1.6.0:
  Working with the Oracle database profiles
- JUnit by Erich Gamma and Kent Beck:
  Testing
- T.O.A.D. 4 by ToadSoft / Quest Software:
  Creation, modification and controlling of the tables in the database

Management of the project

Due to a change of my job, the project plan had to be changed end of august. As a new job always consumes a lot of time for contraction and a lot of work had to be done, a break was irrevocably. This break lasted from beginning of September until the end of the year. Nevertheless, the EmailParser project had been finished in time. However, the dissertation had to be finished after the break. The following diagram shows the project plan:

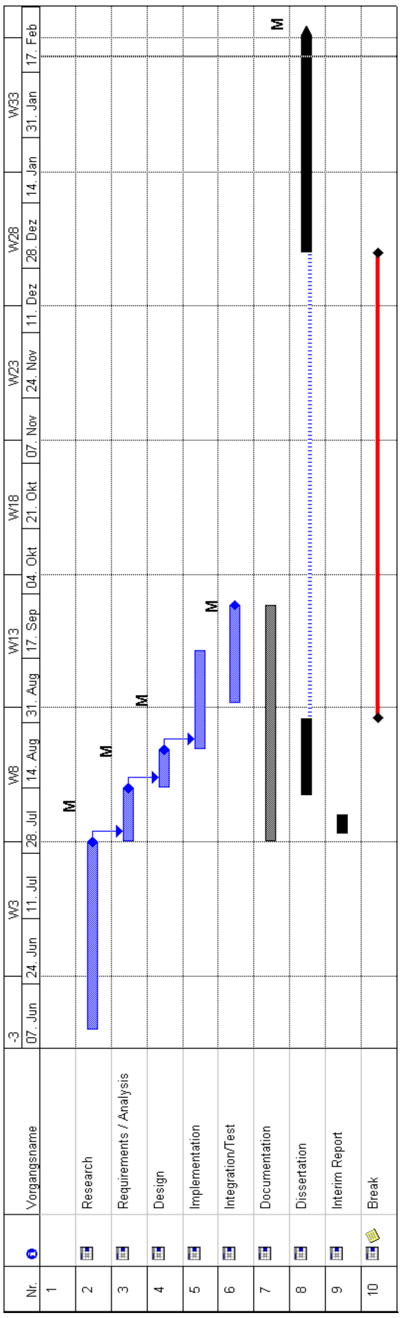

Figure 24 – The project plan

Because of the break, the milestones have been expanded with a fifth one:

M1	End of research phase, beginning of requirements and analysis	31/07/2000
M2	End of requirements and analysis phase	13/08/2000
M3	End of the design phase, begin of the implementation	20/08/2000
M4	End of integration and tests	30/09/2000
M5	End of dissertation	28/02/2001

The research phase had contained already parts of the analysis. This was the most complex and time intensive part of the project and needed up to six weeks. The design phase was done quicker than planned. Indeed the saved time was needed during the integration and test phase, as this phase was costlier as imagined. This was caused by testing the rules by different cases, which were more complex than initially assumed.

Appendix B

Literature

[Behm00] Henning Behme, Stefan Mintert:
"XML in der Praxis"
Addison-Wesley, 2000

[Cimo00] Cimoch, See, Pazzani, Reiter, Lathrop, Fasone, Tilles:
"Application of a Genotypic Driven Rule-Based Expert Artificial Intelligence Computer System in Treatment Experienced HIV-Infected Patients"
Center of Special Immunology, University of California, Irvine

[Fegh98] Jalal Feghhi, Jalil Feghhi, Peter Williams:
"Digital Certificates, Applied Internet Security"
Addison-Wesley, September 1998

[Fuhr00] Kai Fuhrberg:
"Internet Sicherheit"
Hanser Verlag, 2000

[Gamm96] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides:
"Design Patterns",
Addison-Wesley, 1996

[Gran98] Mark Grand:
"Patterns in Java",
Wiley, 1998

[Haeb90] Haeberle, Huber, Mangold, Ruckriegel, Schleer, Schuh:
„Mathematik für Elektroniker", Ausgabe I
Europa Lehrmittel, 1990

[INAS97] INAS Datentechnik GmbH:
"QBase Referenz Handbuch", 1997

[Mask00] Saulius Maskeliunas:
"Extension of a Rule-Based System Usability"
Institute of Mathematics and Informatics, Akademijos 4, 2600 Vilnius, Lithuania

[NISS92] National Information Systems Security (INFOSEC) Glossary
NSTISSI No. 4009, June 5th, 1992

[Orac00] Oracle:

"SQL reference"

[Pro99] Roger Prowse:
"Lecture notes of Software Engineering"
Brunel University

[Scha99] Stephen R. Schach:
"Classical and Object-Oriented Software Engineering with UML and Java"
McGraw-Hill, 1999

[Schn96] Bruce Schneier:
"Applied Cryptography"
Wiley, 1996

[Sun00] Sun Microsystems:
"Java^{TM} 2 Platform, Standard Edition, v 1.3 API Specification"

[Sun01] Sun Microsystems:
"JavaMail^{TM} API Design Specification Version 1.1.2"

[Sun99] Sun Microsystems:
"Getting Started with the JDBC API"
Sun, September 1999

[Wilh99] Gerhard Wilhelms, Markus Kopp:
"Java^{TM} professionell"
mitp, 1999

Glossary

API	**A**pplication **P**rogram **I**nterface
AI	**A**rtificial **I**ntelligence
CRC	**C**yclic **R**edundancy Check: Checksum for error detection.
DTD	**D**ocument **T**ype **D**efinition: Declaration for an XML file.
HTML	**H**yper**T**ext **M**arkup **L**anguage: Markup language using Tags to format strings. Used in the WWW
HTTP	**H**yper**T**ext **T**ransfer **P**rotocol: Protocol used in the WWW to transfer information like HTML-pages
Java	Object-oriented programming language developed by SUN
JDBC	**J**ava **D**ata**B**ase **C**onnection
ODBC	**O**pen **D**ata**B**ase Connection: A standard API defined by Microsoft to connect to and use databases using ODBC-compatible drivers. To work on the database, SQL will be used.
OOA	**O**bject-**o**riented **A**nalysis
OOD	**O**bject-**o**riented **D**esign
Oracle	Company, which develops a very powerful database system
POP3	**P**ost **O**ffice **P**rotocol: Protocol to receive emails from a POP3 server.
SMTP	**S**imple **M**ail **T**ransport **P**rotocol: Simple protocol to transport mails from an email client or server to another. Will be used to send emails.
SQL	**S**tandard **Q**uery **L**anguage: Standard language to access data in a database.

SUN	Company, which has developed Java
Tags	Labels used in markup languages to define segments. Tags are defined in peak brackets (`<Tag>`)
UML	**U**nified **M**odeling **L**anguage: Technique for the object-oriented design of software.
URL	**U**niform **R**esource **L**ocator: Address to get resources by a transfer protocol like HTTP or SMTP
WWW	**W**orld **W**ide **W**eb: Part of the internet.
XML	E**x**tended **M**arkup **L**anguage: Markup language, which allows defining own tags by a DTD.

Table of Figures