Automation of metadata updates in a time critical environment

Johan Karlsteen
c01jkn@cs.umu.se

August 5, 2006
Master's Thesis in Computing Science, 20 credits
Supervisor at CS-UmU: Michael Minock
Supervisor at SMHI: Michael Akinde
Examiner: Per Lindström

Umeå University
Department of Computing Science
SE-901 87 UMEÅ
SWEDEN

Abstract

The Swedish Meteorological and Hydrological Institute, SMHI, handles large amounts of data stored in different databases. Meteorological, hydrological and oceanographical data, called MHO data, have been collected since the 18th century. Apart from the observations, information about the weather stations is stored in the database as metadata and when a station is added or moved the metadata has to be updated.

The purpose of this project is to design and implement an automatic system that performs the updates, which are done by hand as of today. Since the database environment is a centralized time critical system, the meta-update system must not affect the performance on the database.

Contents

1 Introduction **1**

1.1 Background 1
1.2 Metadata 2
1.3 Goal 2
1.4 Organization of this Thesis 2

2 Metadata validation **3**

2.1 The problem 3
2.2 Validating input 4
2.3 Migrating data sources 5
2.3.1 Introduction 5
2.3.2 Data cleaning 5
2.3.3 Data matching 5
2.3.4 Using commercial tools 9

3 Metadata updating system **11**

3.1 Today 11
3.2 Analysis 11
3.2.1 Classes 13
3.2.2 Use cases 13
3.2.3 Functions 15
3.2.4 Requirements 16
3.2.5 System definition 16
3.3 Design 17
3.3.1 Technical platform 17
3.3.2 Architecture 17
3.4 Implementation 19
3.4.1 Metadata input 19
3.4.2 Metadata validation 19

4 Results **21**
4.1 Category-partition testing 21
4.2 Test cases and results 22
4.3 The Metadata Updating System 23
4.4 Problems during implementation 24
4.4.1 ROAD 25
4.4.2 WMO list 25
4.4.3 Values 25

5 Summary and conclusions **27**
5.1 Conclusion 27
5.2 Limitations and restrictions 27
5.2.1 Metadata validation 27
5.2.2 Plain text parsing 28
5.2.3 Station types 28
5.2.4 Spatial queries 28
5.2.5 Using views 28
5.2.6 String matching 29
5.2.7 Testing 29
5.3 Future work 29

References **33**

A Glossary **35**

B Database schema **37**

C Class description and system overview **39**

D Requirements **45**

Chapter 1

Introduction

This report describes the master thesis project "Automation of metadata updates in a time critical environment" performed at the IT department for system development and data warehousing (ITd) at the Swedish Meteorological and Hydrological Institute, SMHI.

1.1 Background

SMHI serves the community with everything from ordinary weather forecasts, analyses, surveys and statistics, expert opinions about the current situation and research. SMHI has the responsibility to raise warnings in case of severe weather forecasts. Without the information about weather conditions airplanes, sometimes, are not allowed to take off.

SMHI handles large amounts of data collected through the years ranging from the 18th century until today. New data is gathered every day from various sources: weather stations, satellites, ships, balloons, buoys and aircrafts. The data is stored in high-end databases and computers analyses the data and produces forecasts. Since several gigabytes of data arrive every day it is important to store it in a structured way.

The databases need to have up-to-date information to supply reliable weather information. Apart from weather observations, data about the weather stations are stored in the database, this is called metadata. The ROAD database system provides information to a large variety of meteorological production systems at SMHI. As the key centralized information system at SMHI, keeping metadata consistent and up to date is important.

The ROAD system is a central database developed at SMHI. ROAD contains MHO data from more than 12000 world wide weather stations and handles large amounts of data every day. The weather stations send data to ROAD at a rate of 150 observations per second. ROAD has to have an uptime of 99.98 percent, this means less than 5 minutes of downtime per month. Applications that needs the observation data retrieves it from ROAD and the system is under heavy load every sixth hour when the number of concurrent applications accessing the database peaks.

ROAD is the newest database system at SMHI and is developed with metadata in mind, this makes it the best developing platform for a metadata updating project.

1.2 Metadata

Metadata is descriptive data about the data itself and in this case it is information about the weather stations: station name, geographical position, altitude and type. The metadata is stored in the same database as the data and updated when needed, for example: a station is moved, the altitude of the barometer is altered etc. SMHI uses a geospatial metadata standard developed by FGDC.

A weather station is not static over time and when they are moved the metadata inside ROAD has to be updated, if not the weather information delivered to the end customers may be wrong. If a station for example is moved 2 kilometers north of its current position and reports fog it may be reported at the wrong location.

Today the metadata updates at SMHI are performed manually. A metadata administrator receives metadata from different sources and SQL queries are created. Another person reviews the updates and a third one creates a patch which is sent to the person responsible for the database and it is inserted into the running environment. This is a very long chain of operation and it is time consuming, especially for the metadata administrator.

WMO (World Meteorological Organization) issues a list once a week that contains stations under WMO's supervision. This list contains information about 12 000 stations but it is not used today. SMHI would like to use the WMO list as a source for metadata but today there is not enough time.

1.3 Goal

It is important to keep the metadata about the weather stations in ROAD up-to-date. Performing the updates manually introduces a high risk for human errors. SMHI would like to minimize the time between an update in the physical world and the reflection inside the database.

The goal of this project is to design and implement an automated system for the metadata updates. The system would gather the necessary information about the weather stations and update the database with the new information. This system will save time, lower the risk for human errors and make it possible to use the WMO list as a metadata source.

The solution is complicated by the sensitive nature of ROAD (centralized time-critical system), traceability requirements and the lack of a standard format for delivery of metadata (among the formats used in the past there have been MS Word documents).

1.4 Organization of this Thesis

Chapter 2 gives a background of the problem of validating metadata and some previous work on the subject. Chapter 3 explains the analysis, design and implementation of the Metadata Updating System, MUS. Chapter 4 contains the results and the tests. Chapter 5 is a conclusion and summary.

Chapter 2

Metadata validation

This chapter discusses some of the problems that arise when the metadata are to be validated. One major problem is to find a matching station in the database when there may be errors in the metadata sources.

2.1 The problem

In a perfect world where all data sources were correct and human errors did not exist this section would be unnecessary, but this is not the case. The different sources for metadata may contain errors and the goal is to minimize them. A metadata source may be computer generated, human written or verbally transferred between humans so it is assumed that errors may be encountered.

The errors could be syntactical or semantic. A syntactical error is an error where the value is invalid, a station height parameter of "seven" is syntactically incorrect because the system expects the value to be "7". A semantic error is harder to detect since the value itself is of correct type but in the environment it is not, saying that "Stockholm" is in "Norway" is wrong but a parser will not see this unless it possesses some kind of intelligence.

Handling the syntax errors in the metadata sources is very easy and it could be done with a strict parser that uses regular expressions for matching the metadata. This requires a little effort from the user but it will result in correct data entry. The semantic errors on the other hand will have to be dealt with after the data reading stage.

For a human eye it is easy to discover obvious errors like a Norwegian station saying it is located in Africa or a station height of 2000 meters in The Netherlands but the computer will have to try to see the whole picture in order to see these kinds of errors. On the other hand the computer could easily see if a station latitude and longitude values are wrong and this may be harder for a human. The solution is maybe to let the computer test for the errors it can easily find and leave the rest to the user. If a station is considered valid in this sense the problem of finding a matching station in the database still remains.

2.2 Validating input

After a station is entered into the system it is assumed that the data itself is at least syntactically correct but still it could be invalid. Spelling errors in the parameters will cause trouble when it is to be matched against the database and in the worst case a new station could be added that contains exactly the same information as a station already inside the database, this kind of redundancy should be avoided. The different fields will be explained in detail here [14].

Identifier

The identifier is either a WMO source number, if the station is used under WMO, or an identifier used by the Swedish and Norwegian meteorological institutes. The WMO number that identifies the station on the Global Telecommunication System, GTS, is a five digit number where the first two (called block number) says in which region the station is situated (Africa, Asia, South America, North and Central America, South-West pacific, Europe or the Antarctic). If the station is a Swedish or Norwegian it will have a five to seven digit number where the first five digit is an identifier and the last one or two says which type of ADAC station it is (ADAC1 to ADAC12).

Area and region

The area and region parameters are plain text and could contain digits, spaces etc. The area corresponds to the block number in the identifier. Both parameters are written in a dual language way with a French translation in the end if there is any. Example: "SWEDEN / SUEDE" and "AFRICA / AFRIQUE". The region is always written on the "ENGLISH / FRENCH" form but the country is only written on the "ENGLISH" form if there is no translation.

Name

The station name could be a local city name but since many stations are not inside or near cities this is no guarantee. This makes it harder to use spelling correction suggested later in this chapter.

Latitude and longitude

There are two different forms in which these values could be written, either on the DEGREES, MINUTES and SECONDS form or the decimal form. The decimal form is used in the database so a conversion has to be made if it is entered in the first form.

Station and barometer height

The altitudes are in meters and the station height has to be entered. A non existent barometer height says that the station does not have a barometer.

Station type

A station on the WMO list is either a "WMO S" or "WMO U" station (Surface or Upper Air) and other station may be of the type "ADAC1-ADAC12", AIRPORT etc.

A WMO station could be both "WMO S" or "WMO U" and in this case two stations should be in the database with the same identifier but different station types.

2.3 Migrating data sources

The stations that are read have to be matched against the stations inside the database and those that are already there have to be updated instead of added. The problem is how to find the corresponding stations if there are semantic errors. The problem of finding fuzzy duplicates inside databases are a big research area and during the years a lot of time has been devoted to finding a solution. This kind of issue often arises in the area of data mining and financial databases where very large databases are to be merged in short time. The area is called data cleansing and different strategies for solving this problem is discussed in this section.

2.3.1 Introduction

The data merging process is quite simple; it consists of 3 steps that can be done individually:

1. Data cleaning
2. Data matching
3. Data merging

The first step takes care of all errors found in the source data. Typically some kind of spelling checker is used to eliminate spelling error.

In the next phase data from the two sources are matched pair wise. This process should maximize the number of correct matches and at the same time keep the number of false positives at a minimum level.

The last phase compares the matched elements, removes any duplicates and merges the elements that are not equal based on some kind of notion about what is correct.

Even though this sounds easy it is not. The following sections will describe why.

2.3.2 Data cleaning

There are many types of data cleaning approaches, the simplest one is to run a spell checker over the data and this is probably a good idea if the data to be cleaned contains names and addresses. Weather stations are spelled in different languages so in this case it is not a good idea.

A database with correctly spelled station names could be used instead but since the number of stations is quite large it is probably not worth the effort [2]. Someone would have to make sure that the station names in the database are correctly spelled and it would probably take too much time.

2.3.3 Data matching

A couple of algorithms for record matching are presented and they are explained in the next three subsections [12]. The following subsections presents other algorithms and the last subsection describes the simplest solution, brute force, and then I present the pros and cons of each algorithm and how they can be used on this specific problem.

A recursive record matching algorithm

This algorithm is a domain independent algorithm for matching strings (records). The algorithm is recursive and records A and B gets a matching value between 0.0 which means no match and 1.0 in which case they are identical (or one abbreviates the other).

The matching is done by dividing the record into i sub records, where it is assumed that A_i of A corresponds to B_i of B with which it has the highest score. The score of the match between A and B the equals:

$$score\,(A, B) = \frac{1}{|A|} \sum_{i=1}^{|A|} max_{j=1}^{|B|} score\,(A_i, B_j)$$

This algorithm handles abbreviations as well by using heuristics. The patterns for matching abbreviations are:

- the abbreviation is a prefix of its expansion, e.g. "Univ" abbreviates "University" or
- the abbreviation combines a prefix and a suffix of its expansion, e.g. "Dept" matches "Department" or
- the abbreviation is an acronym for its expansion, e.g. "UCSD" abbreviates "'University of California, San Diego"' or
- the abbreviation is a concatenation of prefixes from its expansion, e.g. "'Caltech"' matches "'California Institute of Technology"'.

The algorithm is easy to implement, no special data structures are used. The strings are used to store and access the sub records. First the sub records are determined by using a list of delimiters (one for each level of nesting) that splits the record A and B into sub records.

In the next step the record matching function is called recursively for each corresponding pair from A and B. These comparisons determines which sub record of B that matches every record in A. The recursion stops when A or B cannot be decomposed any further.

At this point the heuristics are used to match the strings.

The Smith-Waterman algorithm

The Smith-Waterman algorithm [17] were originally designed to match evolutionary relationships between DNA sequences. It is a domain-independent algorithm as long as the records have the same schema. The algorithm requires the records to consist of alphanumeric characters.

The Smith-Waterman algorithm uses dynamic programming to find the lowest cost series of changes that makes the compared strings identical, this is often referred to as the "'edit distance"'. The costs for mutations, insertions, deletions are passed to the algorithm as parameters. An alphabet is chosen for the algorithm and all other characters are removed prior to execution.

The parameters for the Smith-Waterman algorithm are m, s and c. Given the alphabet, $\sum$, m is a $\sum \times \sum$ matrix of match scores for every pair of symbols in the alphabet. The matrix contains scores for exact matches, approximate matches and non-matches. The idea is that the matrix should compensate for regular typing errors.

One of the most powerful features with the Smith-Waterman algorithm is that it can introduce gaps in the records. Gaps can be thought of as wildcards. The parameters s and c are used to define the penalty for *s*tarting a gap and *c*ontinuing a gap. The ratio between s and c greatly affects the behaviour of the algorithm, for example if c is much smaller than s a few number of large gaps are preferred instead of many small.

The algorithm computes a score matrix E where one of the strings are placed along the vertical axis and the other along the horizontal axis. An entry $E(i,j)$ is the best possible matching score between the prefix $1 \dots i$ of one string and the prefix $1 \dots j$ of the other string. When the strings match the optimal alignment can be found along the main diagonal. For approximate matches the optimal alignment can be found close to the main diagonal. A formal definition of $E(i,j)$:

$$E(i,j) = max \begin{cases} E(i-1,j-1) + & m(letter(i), letter(j)) \\ E(i-1,j) + c & \text{if } align(i-1,j-1) \text{ ends in a gap} \\ E(i-1,j) + s & \text{if } align(i-1,j-1) \text{ ends in a match} \\ E(i,j-1) + c & \text{if } align(i-1,j-1) \text{ ends in a gap} \\ E(i,j-1) + s & \text{if } align(i-1,j-1) \text{ ends in a match} \end{cases}$$

The algorithm can be faster by just computing the values of $E(i,j)$ for values close to the main diagonal.

The hybrid algorithm

Since the recursive algorithm cannot handle records that contain errors and the Smith-Waterman algorithm are unable to handle out-of-order strings, a combination of the two is a better approach and this is achieved with the hybrid algorithm.

The hybrid algorithm uses the recursive algorithm down to a level L where the Smith-Waterman algorithm takes over. This will make it possible to both deal with out-of-order strings and errors.

Sorted neighbourhood method

This method is described in [6]. This paper is pretty old, 1995, but still it is often referred to in papers about databases, data cleaning and merging.

The merge/purge problem assumes that there is not reasonable to compare each element one database with the elements from the other and that there are too many elements to be kept in main memory. The time complexity and the number of passes over the elements have to be kept at a minimum. The data is assumed to be corrupted and the goal is to maximize the number of matches while keeping the number of false positives at minimum.

This is done by gathering all the records from the databases in one table and then calculate a key for every record. The key should consist of relevant information from each attribute. In the next step the elements are sorted according to their key and finally a fixed size window is moved through the list and every time the window is moved the new element is compared with the ones already in the window. Matching records are merged and the process continues until all elements have been processed.

This method is pretty straightforward and the only difficult part is to construct a key. Sorting the list is easy and it does not require a lot of computing power. The key of course has to take care of the problem with the semantic errors or else there will

be problems. Before the key is computed the data could be automatically corrected and spelling errors could be removed. In an example in the paper US city names are corrected against a list with the correct names and this would require a list with all station names. Since the names are changed over time it is not reasonable in this case.

A possible key for station metadata would be the WMO number concatenated with the station name and the geographical position.

Approximate string joins in databases

This technique is discussed in [5]. In this paper the matching is done by comparing small substrings of length q, called $q-grams$, and the technique is similar to the Smith-Waterman algorithm [17].

First of all the strings are broken down into $q-grams$ of length q. If a string would be "'Stockholm"' the $q-grams$ would be:

$$\{(1, \#\#s), (2, \#st), (3, sto), (4, toc), (5, ock), (6, ckh)$$

$$, (7, kho), (8, hol), (9, olm), (10, lm\$), (11, m\$\$)$$

For the string "'Stokcholm"' the $q-grams$ are:

$$\{(1, \#\#s), (2, \#st), (3, sto), (4, tok), (5, okc), (6, kch)$$

$$, (7, cho), (8, hol), (9, olm), (10, lm\$), (11, m\$\$)$$

The strings are compared by checking how many of the $q-grams$ that match. In this case 7 of 11 $q-grams$ match. Another important thing is the order of the $q-grams$ since it is more probably that two strings match if the order is the same.

The edit distance of two strings, s1 and s2, is defined as the minimum number of point mutations required to change s1 into s2, where a point mutation is one of:

1. Change a letter,
2. Insert a letter or
3. Delete a letter

The calculations are performed in a database and this is the formalization of the problem:

Given tables R_1 and R_2 with string attributes $R_1.A_i$ and $R_2.A_j$, and an integer k, retrieve all pairs of records $\left(t, \hat{t}\right) \in R_1 \times R_2$ such that the edit distance $\left(R_1.A_i\left(t\right), R_2.A_j\left(\hat{t}\right)\right) \leq k$.

The string join is performed by first selecting rough candidates to the string with a method that does not allow false positives. When this is done the candidates are compared in-memory using edit distance.

Functions for executing this is easily implemented in for example Oracle or DB2 (Informix is DB2).

Brute force method

The simplest approach to this problem is to take the station metadata and loop through all elements already inside the database and find the best match. If there is no match that is good enough the station can be added or else it is updated. The comparison between the stations will be very time consuming especially if every station will have to be compared with the approx 13000 ones in the database. The comparison between stations could still be done with a key that identifies the station.

If there were no errors this method would be easy, just compare the keys with each other and see if they match. To be able to deal with changes in the data this comparison has to be done using some kind of sub key. A sub key would be a part of the key that could be changed from time to time, for example the station name or the geographical position.

Comparing 13000 stations with another 13000 stations is not reasonable but by using the SQL server to deliver rough matches this could be improved. ROAD has got a geodetic datablade which allows for geospatial SQL queries to be executed [8]. This could improve the process by only comparing the stations that are close to the station being examined, assuming that the latitude and longitude values are correct.

2.3.4 Using commercial tools

Some of the methods mentioned earlier are quite old and nowadays there are a couple of commercial applications available and DBMS like SQL Server 2005 and Oracle 10g have these functionalities built in.

SQL Server 2005

In Microsoft SQL Server 2005 two features called ”Fuzzy lookup” and ”Fuzzy grouping” are introduced [4]. Fuzzy lookup and fuzzy grouping is using a combination of ”‘edit distance”’, the number of tokens in the string, the order of the tokens and the relative frequency. According to Microsoft this will result in a better result than traditional approaches.

This technology is available in the SQL Server 2005 Integration Services and the lookup is constructed via a wizard. Since this is unavailable at SMHI this will not be explained further.

Chapter 3

Metadata updating system

This chapter evaluates the situation of today by finding the actors who are involved and specifying use cases for them. The functions are specified and a system definition is created.

3.1 Today

Today the metadata is updated by a person called "Metadata Administrator", MA. The MA is responsible for updating the metadata inside ROAD. The sources for metadata are: e-mails, the WMO list and other SMHI employees. The e-mails are sent directly to the MA from other meteorological institutes, for example the Norwegian Met.no, and contain information about one or more weather stations where the metadata has been changed. The e-mails are formatted as plain text and the sender decides the format of the content, most commonly only the important information will be written like station name, WMO number and the new metadata. The MA must then find the station in ROAD and alter the metadata by constructing SQL queries. The database schema for ROAD is shown in figure B.1 and B.2

The WMO list is published by the World Meteorological Organization, WMO, and it contains weather stations connected to the Global Telecommunication System, GTS, and these stations may be used without having to ask the country that owns the station. The structure of the file can be found in figure B.3. More about the structure can be found in [14]. The WMO list arrives once a week, and even if there is no change in the station metadata, this file contains all stations.

After the SQL queries are constructed they are sent to the Technical System Administrator, TSF, for review. The TSF checks the SQL queries for syntax errors.

The TSF then sends the queries to the Content Manager, CM, who will create a patch. A patch consists of the SQL queries and a version document.

The patch is sent to the System Operator for ROAD who will patch the running environment.

3.2 Analysis

After interviewing people that are involved in the task of updating the metadata the following rich picture have been created in figure 3.1. The concept of a rich picture is

discussed in [10]. Some information were gathered from old thesis's performed at SMHI ([13] [11]).

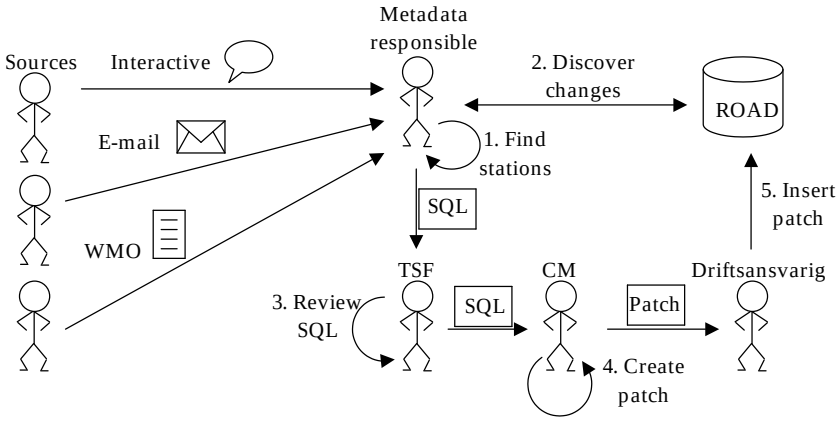

Figure 3.1: Rich picture created from the work flow at SMHI

Source

The source actors deliver metadata information to the metadata administrator. There are 3 different types of sources: e-mails, interactive updates from SMHI employees and the WMO list. The e-mails and interactive updates can be thought of as event driven while the WMO list is scheduled and issued once every week.

Metadata administrator

The metadata administrator receives metadata information from the source actors.

To decide which stations that have been changed the MA has to query ROAD. This is done by executing SQL queries that extract information about the current state of a station. Every weather station on the WMO list has a WMO-number that uniquely identifies it worldwide but inside ROAD there is another primary key and the MA has to retrieve it for every station. ROAD contains a lot of metadata but the MA is only responsible for updating the station metadata.

The geographical coordinates are often written in minutes, degrees and seconds and has to be converted to decimal format using a separate application. For every new or updated station an SQL query that inserts or updates the information in ROAD has to be written, this is done by hand with a lot of cut and pasting involved, this introduces a risk for human errors.

When this is done the user sends the SQL queries to the TSF for review.

TSF

The TSF checks the queries for correctness.

CM

The CM uses the queries and creates a patch that consists of the queries accompanied with a version document. Sometimes a roll-back patch has to be created and this is done at this time. A roll-back patch is the opposite of the patch and it will restore the database to its previous state. The roll-back patches are used in case of major updates to the ROAD system. The patch is sent to Driftsansvarig.

SO

When the SO receives the patch it is applied to ROAD and the state of the metadata is updated.

3.2.1 Classes

The different classes:

Name	Description
Metadata	Information about stations
Station	Weather station, read below table
Patch	A patch created from queries
ROAD	the database

Metadata

The metadata is a collection of stations.

Station

A station represents a weather station and its metadata. The attributes are: station name, WMO number, source number, source type, latitude, longitude, station height, barometer height, region, country.

Patch

The patch is created from the stations after they have been compared with the stations inside ROAD. A patch is the SQL query for each updated station and a version document.

ROAD

The MA compares the stations from the metadata with the ones inside ROAD. A patch is applied to ROAD.

Actions

The combined classes and actions table is in figure 3.2.

3.2.2 Use cases

Figure 3.3 gives the use case model.

	Classes			
Actions	Metadata	Station	Patch	Database
station read	X	X		
station examined		X		X
queries created		X		
queries reviewed			X	
patch created		X	X	
patch delivered			X	

Figure 3.2: Action Class Table

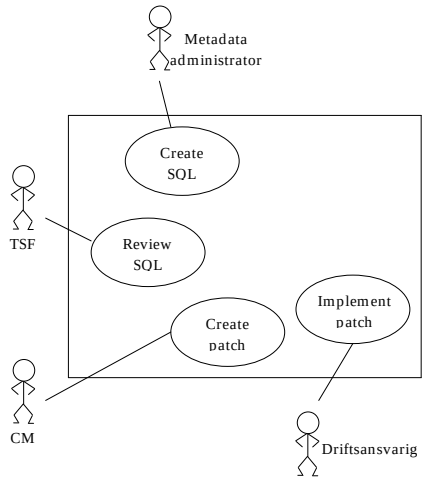

Figure 3.3: Use case model

UC1: Create SQL

1. The MA receives metadata updates

   E-mail

   The e-mail is read.

   WMO list

   The list is read (not done today).

   Interactive update.

   Information received.

2. For every station in the metadata

   Station is in database:

   Retrieve current source number.
   Write SQL-query for the updated data (UPDATE).

   Station is not in database:

   Retrieve the first vacant source number.
   Write SQL-query for new data (INSERT).

   For the WMO list (not done today).

   Find station in database even if it is misspelled.

(If the station is not in the database it is added with an SQL-query).
Compare station with station in database.
Check the attributes for errors.
Severe errors are reported to the user for review and changes.
Decide which version of the station that is correct.
If the new station is the correct one then construct SQL-query (UDATE).

3. All queries are gathered in one text document.
4. Queries are sent to the TSF.

UC2: Review queries

1. The SQL-queries are read.
2. Syntactical errors are corrected.
3. Queries are sent to CM.

UC3: Create patch

1. CM receives queries.
2. If a roll back patch is needed.

   TSF creates a roll-back patch.

3. Current version of database is checked.
4. Version document is created.
5. Queries and version document are compressed in a tar.gz-file.
6. Patch is ready for insertion.
7. Driftsansvarig is notified.

UC4: Apply patch

Driftsansvarig applies the patch to the database.

3.2.3 Functions

Name	Complexity	Type
Read sources.	complex	reading
Detecting and verify changes	complex	calculation
Create SQL queries.	simple	calculation
Create patch	simple	calculation
Create a rollback patch	normal	calculation

Reading source

Source	Formatted	How often
E-mail	No	Irregular
Interactive	No	Rare
WMO list	Yes	1/week

The e-mails have no predefined format and the MA has to read them and collect the necessary information. Most often the e-mail will only contain the station name and the changed values.

An interactive update will only contain minimal information and the MA will have to write this down.

The WMO list is a structured tab separated text file and contains all information about a station every time.

Reading unstructured data is harder for the computer than for the human and all sources are assumed to contain errors.

Detecting and verifying changes

Every station entity that has been extracted from the sources will be checked against the database. If there is no match in the database the station is a new one and can be inserted without problems. An updated station has to be compared with the old values and the attributes will be checked for semantical errors.

3.2.4 Requirements

The requirements have been captured and documented using the Voliere process ([16]). Detailed voliere cards can be found in appendix B.

3.2.5 System definition

The metadata updating system, MUS, is a system that will be used to update metadata inside ROAD. MUS will discover what should be updated and create a patch. MUS should be fast and reliable; errors should be kept at a minimum level. The user should feel comfortable with MUS and that using it saves time. MUS should be developed in close contact with the future users of the system.

Since the database environment is time critical it is very important that MUS does not lower the performance more than necessary. To ensure this thorough testing of the system in the production environment will be performed before deployment. The platform used will be Windows XP and Java. The system should be easy to extend in the future since the environment may change.

The table below gives the system definition with regards to the FACTOR design model [10].

Functionality:	Read metadata information. Detect metadata changes. Create a correct patch with new or updated metadata.
Application domain:	Administration of metadata updates and creation of patches inside ROAD.
Conditions:	The system shall work with the different sources for updates. The system has to comply with the guidelines for updates used by SMHI. The system should not lower the performance of the surrounding environment.
Technology:	Windows XP with Java and JDBC.
Objects:	sources, stations, queries, patch, database.
Responsibility:	MUS should as far as possible be automatic but user interaction will be needed. MUS should be easy to extend in the future.

3.3 Design

3.3.1 Technical platform

The system platform will be Windows XP and MUS will run as a standalone application. Java will be the programming language because it is easy to use and should be portable. It is assumed that ROAD is available on the LAN and JDBC will be used for database connectivity. The graphical user interface will be created with swing technology.

3.3.2 Architecture

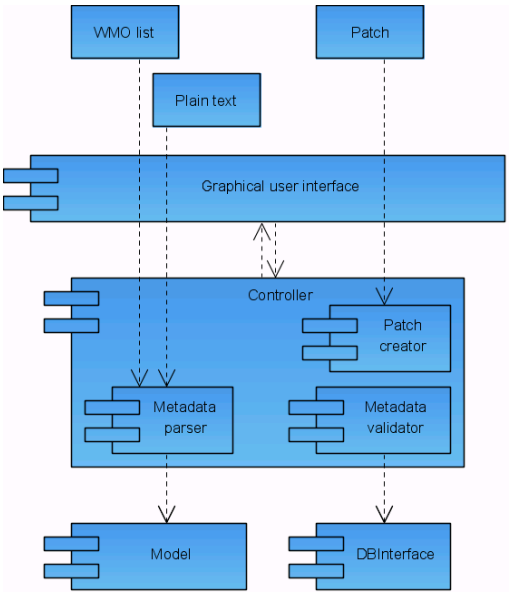

Figure 3.4: The architecture of MUS

The architecture (figure 3.4) uses the Model View Controller, MVC, paradigm [7] [1] [3]. MVC encapsulates the data from the graphical user interface, GUI, and the controller holds the main functionality. Keeping the model isolated from the GUI and the controller makes it very easy to extend the functionality in the future and also the MVC model forces the system to be loosely coupled.

MUS will read metadata from a WMO list supplied by the user or from an input form inside the GUI. The metadata will be parsed by the system and for every station that is represented in a metadata source a station object will be created. The model will keep a data structure for all stations read into the system. Since there is a risk for error in the input the metadata has to be checked at parsing time.

When all metadata update information has been parsed into the program it will be validated against ROAD, validation means that station metadata already inside ROAD must be compared with the new station metadata. Given the risk for misspellings of station names and station numbers the system will have to try partial matching if the station is not found.

If a station is found the metadata is compared and if there is no difference nothing will be done i.e. the station metadata will not be included in the patch. If the metadata

is different the new information will be checked to avoid entering errors like switched latitude and longitude values. If the new station metadata is valid it will be marked as either "minor change" if the change is very small and "major change" if the change is big, for example moved 200 kilometers.

If the station is not found then it is added to the patch since it is a new station.

After the validation stage the model holds a list of stations that should be included in the patch. The user will have to approve all stations that are marked as "major change" while the "minor change"-ones can be automatically approved.

For every station on the list which is marked for patch and that has been approved an SQL query is created and the queries together with a version document is created.

Right now the database schema is under development and will be changed pretty soon so all database operations will be performed against a view. A view is easy to maintain even after a database schema change.

Figure 3.5 shows the workflow of MUS.

A complete class description can be found in the appendix C.

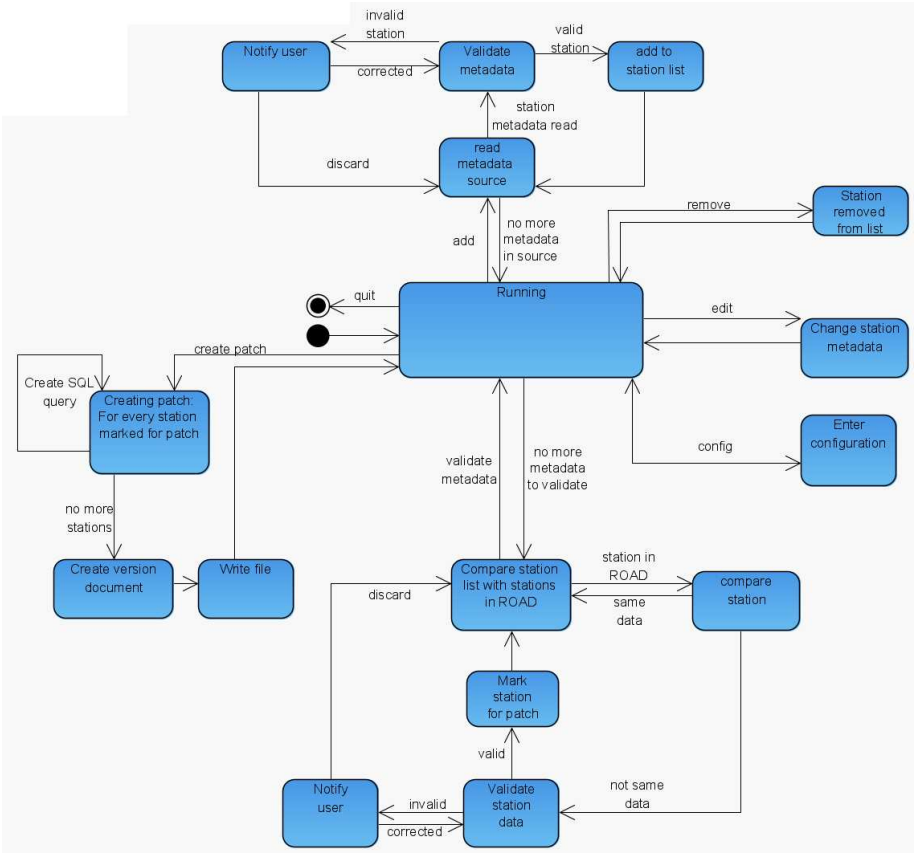

Figure 3.5: Workflow of MUS

3.4 Implementation

3.4.1 Metadata input

Metadata about weather stations is read either from a WMO file or by entering the data manually in a form. The metadata is passed on to the internal data structure holding the stations and a new station object. The station object tries to parse the metadata first as a WMO list line and if that fails a regular expression is used. If the parsing is successful the station is added to the station list or else the user will be informed.

If an old station is to be changed this can be done by entering the WMO number of the station and the qualifier, the system will try to find the station in the database. If the retrieval of the old station is successful the station metadata is presented to the user that may change the information.

3.4.2 Metadata validation

When validating the stations the system connects to the database and tries to find a corresponding station in the following order:

1. Name and WMO number matches.
2. WMO number matches and name and is LIKE (SQL operator).
3. WMO number matches.
4. Name is LIKE and station is at most 4000 m from the new position.
5. Station at most 1000 m from the new point.

If a query above returns a station the station will be tested for likeliness. The criteria's for a station being alike another is the following:

1. If the station names are considered alike.
2. If the WMO number is the same.
3. If the new position is less than 7845 m from the old one (this is a change in latitude and longitude of less than 0.05).

Together the three variables are used and a station will be a match if:

- 1 and 3 are true.
- 2 and 3 are true.
- 1 and 2 are true.
- The name is equal (not string match but strict equal).
- The latitude and longitude are equal.

Chapter 4

Results

SMHI requires that all software used in and against ROAD have to be tested and the category-partition testing will be used for the black box testing. Unit testing was performed during development using JUnit [9] but this will not be documented.

4.1 Category-partition testing

1. Identifying categories for each input parameter.
2. Each category is partitioned into choices. A choice is a specific test value for a category.
3. Determine constraints among the choices.
4. Generate test cases by enumerating all choice combinations.

A formal definition for category-partition testing can be found in [15]. The test specification is found in figure 4.1.

Category	Partition	Sample
Data source:	WMO file	correct, incorrect, partially correct, no file, empty file
	manual	correct, incorrect
Config file:	Config file	correct, incorrect
Station data:	Station name	correct, incorrect
	Source number	correct, incorrect
	Lat/Lon	correct, incorrect
Database:	Credentials	correct, incorrect
	Server name	correct, incorrect
	Database name	correct, incorrect
	Driver	correct, incorrect
	Network	available, unavailable
Station catalogue:	Station exist	major change, minor change, unchanged, not found

Table 4.1: Test specification for category-partition testing

4.2 Test cases and results

The following test cases are created from the category partition table. Station data and database matching will be the primary categories. Config file, Data source, Config file and Database are not-primary categories. To speed up the testing each test case will be run with at least one station from each sample in the station catalogue category. Incorrect means that it is a misspelled value but it has still passed the parser.

Test case 1: All attributes correct

The station from the test set where inserted into the program and all stations were found without any changes.

– Test passed

Test case 2: Incorrect name

The station where altered so that the names were different from the first test set. This test set corresponds to a series of stations where the names have been changed and since the name is the only altered parameter all stations are found and the patch produced contains the name changes.

– Test passed

Test case 3: Incorrect source

This test set has wrong WMO numbers for the stations but since the names and lat/lon values are unchanged all stations are found. The patch produced from this test will change the WMO-numbers in the database.

– Test passed

Test case 4: Incorrect lat/lon

Incorrect lat/lon is equivalent to a station movement and this is the most common case that the program is assumed to work with. All stations are found and the patch contains the new latitude and longitude values.

– Test passed

Test case 5: Incorrect name and source number

The only thing that is not changed is the position. Depending on how different the name is the program will either find the station by matching the name together with the lat/lon or by using a geodetic query.

– Test passed

Test case 6: Incorrect name and lat/lon

The source number is correct but this is no guarantee that the system will find the correct station. If the name is matched by the program then it will be treated as a station with a changed name that has been moved or else a new station will be the result. This may be wrong but it is very hard to decide if a station is a changed one or a new one. If the position is at most off by at most 4000 m (with the current config file) it will be found. This test is very much dependent on the string matching method and improving it would improve the results. A little less than half of the stations in the test set were not found.

This test is equivalent to a station that has been moved and name changed and this may happen. Hopefully the station is not moved too long and the program finds it.

Test case 7: Incorrect source number and lat/lon

If the name is exactly the same and the station is not moved more than 4000 meters it will be found. The patch will change WMO number and position. The probability that the source number is incorrect and the latitude and longitude values is at most 4000 m from a station that has a name that looks like this one is low.

Test case 8: All attributes incorrect

Finding a station where all variables are incorrect is hard and the results from the tests are poor. Still it is not likely to happen that the source number is incorrect, the name is completely wrong and the new position is too far from the previous position.

4.3 The Metadata Updating System

The system is launched and the data structures containing the weather stations, the database interface and the GUI is initialized. Configuration data is read from an XML file. In figure 4.1 the program has just been started and no stations have been added.

Basic functions:

- ”Add” adds a single station or a station list.
- ”Remove” removes a single station.
- ”Edit” are used to change metadata for a single station.
- ”Quit” exits the program.
- ”Validate” will validate the stations against ROAD, after validation all stations are moved to the ”Patch list”-pane.
- ”Approve” approves a single station on the ”Patch list”-pane.
- ”Patch” creates the patch from all the stations on the ”Patch list”-pane that are appproved.

When editing data or editing a single station the dialogue in figure 4.2 is shown.

- The fields correspond to the attributes of a station.

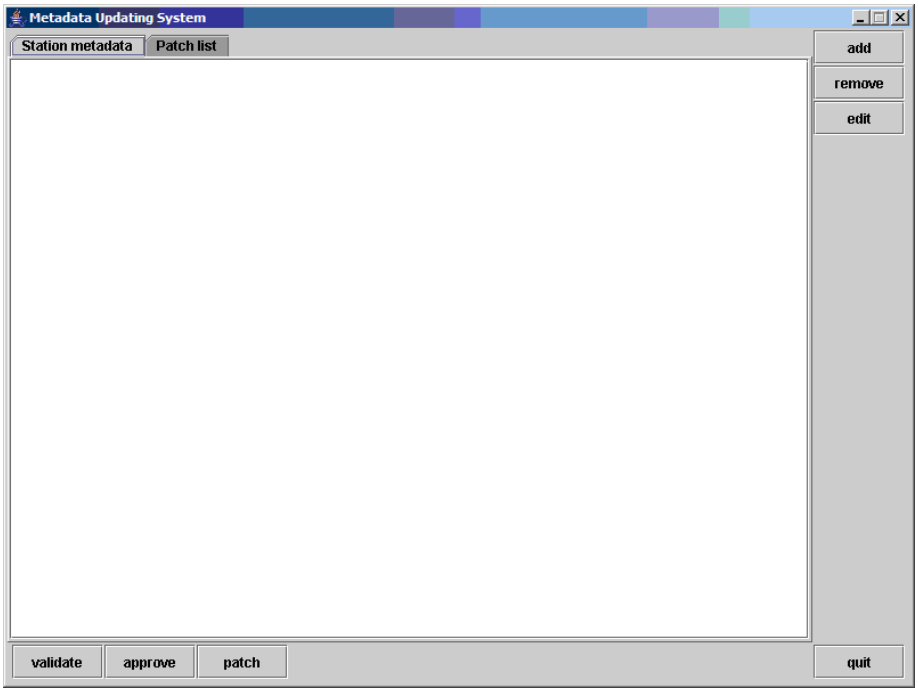

Figure 4.1: Screenshot of MUS main page

- The "Update"-button tries to parse the metadata, if there is an error the user will be notified.
- The "Retreive"-button is used to retrieve metadata about an old station from ROAD, the MA just have to enter the source number and the station type and the rest of the fields will be populated
- The "Close"-button closes the window without saving any information.

After stations are added and "Validate" is pressed the screen may look like figure 4.3

- A green station is an old station that has been found in ROAD and been approved.
- A yellow station is an old station found in ROAD not yet approved.
- A red station is a new unapproved station.

All stations that are approved will be included in the patch when the "patch"-button is pressed and all other stations will be removed.

4.4 Problems during implementation

One common problem was to gather information about ROAD since there are many people involved and no one knows everything.

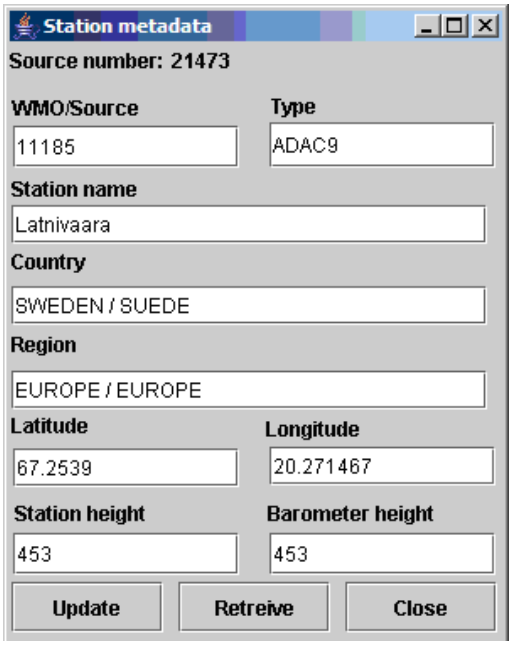

Figure 4.2: Screenshot of MUS edit dialogue

4.4.1 ROAD

There is a lot of old and erroneous metadata in the database and this makes it hard to use. Most of the stations are inserted in 1999 and the metadata has not been changed since. Some of the stations have been inserted with switched latitude and longitude values.

A good thing to do before using this program is to drop all old WMO stations and add a new WMO list.

4.4.2 WMO list

About one week before the testing of the system was to begin I found out that some of the elements on the WMO list were to be treated as two separate stations. There are two kinds of stations on the list: surface and upper air. This resulted in a major change of the station class.

4.4.3 Values

When integers are inserted into the database and there is a leading zero it will be removed by the DBMS. This will make it hard to find the station later on. This can be fixed with an SQL query but still the program is ignoring this right now. If a station is found and the leading zero is missing it will still match.

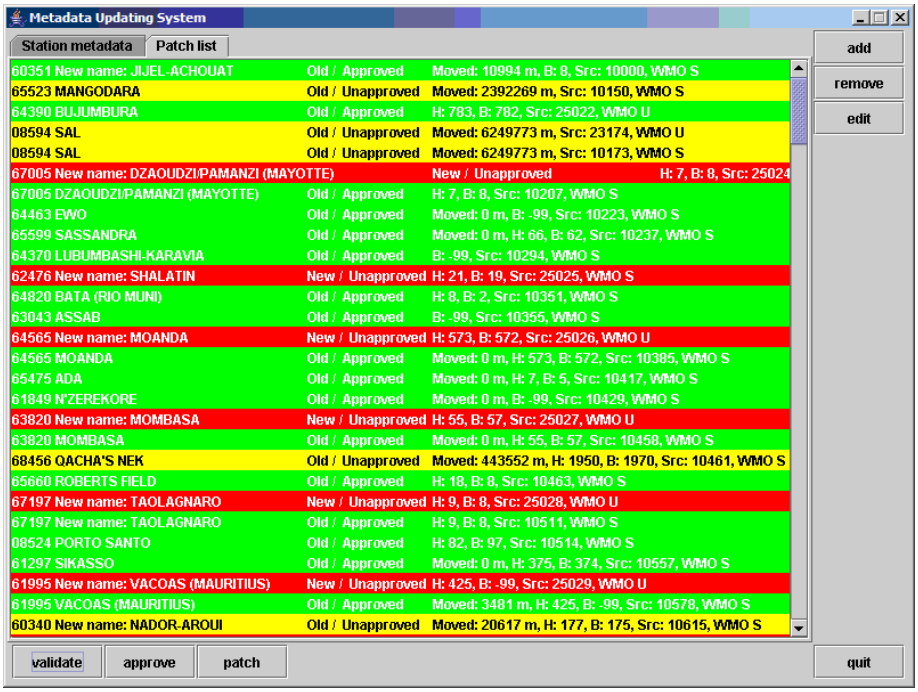

Figure 4.3: Screenshot of MUS "Patch list"-pane

Chapter 5

Summary and conclusions

5.1 Conclusion

When automating these kinds of updates on a running database environment that has been online for six years there are some issues that will arise. If the database schema is not extremely strict there will be anomalies in the data itself, this can almost be guaranteed. When testing the program i ran into several records that were wrong and this will make it very hard to implement the new order.

If it is possible it should be considered to purge the old database and start up with a new one, but this is seldom applicable. The other approach is to perform some serious data cleaning on the old data before even starting to build the new system. This is a common problem when merging old databases into new ones and the data contains minor errors that works ok in the old environment but will fail on insert in the new one.

Performing the data cleaning is not trivial and with large databases this will likely require a lot of effort from all the people who are involved in the process. At SMHI the "purge" discussed earlier is not an option and since the time was limited there was no room for data cleaning before implementation.

5.2 Limitations and restrictions

5.2.1 Metadata validation

The first solutions to the data merging problem described in chapter 2, "'The recursive algorithm"', "'Smith-Waterman"', "'The Hybrid algorithm"', "'Sorted Neighbourhood"' and "'Approximate string joins"' all have one downside, they are constructed assuming that the data that are to be compared is not changed. In this case the data could be changed and some kind of probability measure must be used to determine if the new metadata is wrong or changed.

The main idea with this application is that it should be able to extract metadata from the sources and find the old entries in the database even if the new data is changed. On the attribute level of each station some kind of string matching algorithm may be used but only if the attribute is not changed.

Comparing the stations as strings works but instead of letting one string represent each station there should be one string per attribute. Deciding if two stations are equal are done by using a threshold for how many attributes that have to match. The hard

part with this solution is that the threshold values have to be set carefully and probably after testing but as of now this is the only solution.

5.2.2 Plain text parsing

In the requirements it is stated that the system should be able to read plain text e-mails but this was not implemented. The main reason is that there was no time and it was considered unnecessary.

5.2.3 Station types

The system can only handle stations of type ADAC, WMO S (Surface) and WMO U (Upper Air) right now. This is the most common station types but there may be scenarios where other types are needed.

Some of the stations on the WMO list perform both surface readings upper air readings (weather balloons). ROAD uses the same number for both types of stations but their source type is different (WMO S and WMO U). To allow this the station parser always assumes that there should be a surface station and if an upper air station is needed the station will duplicate itself and add it to the station list.

When parsing a WMO list there are sometimes stations on consecutive lines that have the same WMO number, this is because meteorologists want to have the same number when two stations for different readings are placed in the same location. This is impossible to have in the database since there is a unique constraint and this is handled by the station list by looking at an index variable on the list, if the index is 1 then the station is the second one and a surface station will not be created.

5.2.4 Spatial queries

The geodetic datablade works very well but it is slow for latitude and longitude queries (about 1.5-3 seconds each). When a WMO list with 11000+ stations is validated this is an issue. This will only be the case during the first time a list is added and many stations are new. If the station list is read once a week most of the stations should have a correct WMO number and name. The lat/lon query feature can be disabled in the configuration file.

The smartest thing would be to remove all old WMO stations and add a new WMO list with the feature disabled or else there will be a spatial query for every station. After this is done the feature can be enabled assuming there will be a small number of changes between the following lists.

5.2.5 Using views

SMHI is changing the data model right now and this will affect the system. To avoid future problems the system is designed to work against a database view rather than the tables. This approach would allow changes to the model as long as the view is correct.

One bad thing with views are that they do not really take advantage of the indexing in the database for some SQL questions. The solution was to add a sub query to the SQL statement that uses one of the indexed tables in the database and this resulted in a change in execution time from 1.5 seconds to 0.015 seconds per query.

5.2.6 String matching

Since the notion of equality is subjective the test for likeliness has it flaws and is far from perfect. The following stations will result in errors:

- Station that has the same WMO number but a name that does not match and a position more than 7845 m away. This will result in a patch that will try to add a station where the WMO number is already in the database and this will yield an error. A possible solution would be to remove all the old records first.
- If a station, A, has another stations, B, exact position and B has a new position (which is very likely) then A should also have different WMO number. This will result in B being name changed and the source number will also be changed by A (assuming A is the first station on the list).

A different strategy to avoid problems would be to remove WMO stations that are not in the patch from ROAD since these stations are not active, this may result in problems for the new archive system that are to keep all stations ever used in the database.

5.2.7 Testing

There was not enough time to perform any documented testing of the performance impact on ROAD. The testing of the resulting SQL queries was performed with about 250 stations which is far to few to draw a conclusion whether or not the system is correct.

5.3 Future work

The more intelligent the string matching function is the better this program will handle misspelled names.

Making the station class more general would be good and this would make it possible to use the system on SMHIs other database systems. There were a lot of comments about this kind of system and hopefully someone will have the time to customize the system for other departments.

Acknowledgments

I really would like to thank Michael Akinde who has been my supervisor at SMHI for all the help during my thesis, without Michaels hard work there would probably not be a thesis. Thanks to my internal supervisor Michael Minock. Thanks to Gunilla Mild for her time with the analysis and the testing of the system. Thanks to Esa Falkenroth for interesting discussions and valuable comments. Thanks to ITd at SMHI for all help. Thanks to SMHI for letting me perform the thesis and for all employees who have shared their experiences with me during numerous coffee breaks.

References

[1] Joseph Bergin. Building graphical user interfaces with the mvc pattern, 2005-10-18. http://csis.pace.edu/ bergin/mvc/mvcgui.html.

[2] Michael Allen Bickel. Automatic correction to misspelled names: a fourth-generation language approach. *Commun. ACM*, 30(3):224–228, 1987.

[3] Jason Cai, Ranjit Kapila, and Gaurav Pal. Hmvc: The layered pattern for developing strong client tiers, 2005-10-17. http://www.javaworld.com/javaworld/jw-07-2000/jw-0721-hmvc_p.html.

[4] Surajit Chaudhuri, Kris Ganjam, Venky Ganti, Rahul Kapoor, Vivek Narasayya, and Theo Vassilakis. Data cleaning in microsoft sql server 2005. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 918–920, New York, NY, USA, 2005. ACM Press.

[5] Luis Gravano, Panagiotis G. Ipeirotis, H. V. Jagadish, Nick Koudas, S. Muthukrishnan, and Divesh Srivastava. Approximate string joins in a database (almost) for free. In *VLDB '01: Proceedings of the 27th International Conference on Very Large Data Bases*, pages 491–500, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[6] Mauricio A. Hernandez and Salvatore J. Stolfo. The merge/purge problem for large databases. In *SIGMOD '95: Proceedings of the 1995 ACM SIGMOD international conference on Management of data*, pages 127–138, New York, NY, USA, 1995. ACM Press.

[7] John Hunt. You have got the model-view-controller. 2005-10-01. http://www.cs.txstate.edu/ rp31/papers/Model-View-Controller.PDF.

[8] IBM. Geodetic datablade module user's guide, version 3.11, 2006-05-07. http://publib.boulder.ibm.com/epubs/pdf/8675.pdf.

[9] Junit. Junit test infected: Programmers love writing tests, 2005-11-29. http://junit.sourgeforge.net/doc/testinfected/testing.htm.

[10] Lars Mathiassen, Andreas Munk-Madsen, Peter Axel Nielsen, and Jan Stage. *Objektorienterad analys och design*. Studentlitteratur, Lund, 2001.

[11] Daniel Melander. A meta data harvester for geospatial data. Master's thesis, Linköpings Universitet, 2004. LITH-ITN-ED-EX-04/014-SE.

[12] Alvaro Edmundo Monge. *Adaptive Detection of Approximately Duplicate Database Records and the Database Integration Approach to Information Discovery.* PhD thesis, 1997.

[13] Amna Music. Grafiskt användargränssnitt för digital geospatiell metadatasökning. Master's thesis, Linköpings Universitet, 2004. LITH-ITN-MT-EX-04/009-SE.

[14] World Meteorological Organization. Wmo publication no. 9 - weather reporting: Volume a - observing stations, 2005-08-15. ftp://www.wmo.ch/wmo-ddbs/OperationalInfo/CD/VolumeA/VolumeA.pdf.

[15] T. J. Ostrand and M. J. Balcer. The category-partition method for specifying and generating fuctional tests. *Commun. ACM*, 31(6):676–686, 1988.

[16] Suzanne Robertson and James Robertson. *Mastering the requirements process.* ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1999.

[17] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147:195–197, 1981.

Appendix A

Glossary

CM	Content Manager
DBA	Database administrator
Informix	IBM RDBMS
FGDC	The Federal Geographic Data Committee (www.fgdc.org)
Geodetic datablade	Database plugin allowing spatial queries
JDBC	Java database connectivity technology
MA	Metadata Administrator
MVC	Model View Controller
ROAD	Realtids Och Arkiv Databas
SMHI	Sveriges Meterologiska och Hydrologiska Institut
TSF	Teknisk System Förvaltare

Appendix B

Database schema

column name	type name	column size
source	int	4
name	varchar	50
datechanged	date	4
sourcetype	varchar	20
description	lvarchar	2048
geo	geoobject	2048
regionname	varchar	30
countryarea	varchar	110
indexsubnbr	char	1
obstype	int	4
county	varchar	30
barometerheight	float	8
stationheight	float	8

Figure B.1: The metstation table contains information about the weather stations, the primary key is source

column name	type name	column size
source	int	4
qualifier	char	10
key1	int	4
key2	int	4
key3	varchar	20

Figure B.2: The sourcexref table contains the source number of a weather station, the qualifier (ADAC, WMO S, WMO U, etc) and the WMO number. The primary key of the table is a composite key of source, key1, key2, key3

Index number	A station index number in the form IIiii is included in the reports of meteorological observations made at land meteorological stations or aboard lightships using land code forms. This group allows the identification of the meteorological station at which the observation has been made. The station index number is composed of the block number (II) and the station number (iii). The block number defines the area in which the reporting station is situated.
Name	The station name
Latitude	Latitude in degrees and minutes. The positions of stations north (N) or south (S) of the Equator are indicated by the appropriate letters after the minutes figures.
Longitude	Longitude in degrees and minutes. The positions of stations east (E) or west (W) of the Greenwich meridian are indicated by the appropriate letters after the minutes figures.
Station elevation	The elevation of the station in metres.
Barometer heigth	Official altitude of the aerodrome, is given for stations located on aerodromes.
Pressure level	For those stations not indicating air pressure reduced to mean sea level
Surface synoptis observations	The symbol X means that surface observations are made regularly in accordance with a fixed schedule. If figures are indicated instead of an X, e.g. 02 in the column 03, this means that observations are made regularly at 0200 UTC instead of at the standard observation time of 0300 UTC.
Hourly observations Half hourly observations	This column indicates the hourly and half-hourly observations made at the station. Hourly observations are shown by the letter "H" followed by the period of the day during which they are made. Similarly, half-hourly observations are shown by the letter "S" followed by the period of the day during which they are made. Except in a few cases the period begins and ends on the hour, and the times are represented by two figures. When the period begins or ends at the half-hour the full four-figure time is given.
Upper-Air observations	Upper-air observations are indicated by means of one or more appropriate letters below the corresponding standard observation time.
Other observations and remarks	This column provides information on additional observations (meteorological, geophysical, hydrological, etc.) made at the station, such as nephoscopic, radiation, seismological, phenological, river stage, aircraft reconnaissance observations, etc.

Figure B.3: Structure of the WMO file

Appendix C

Class description and system overview

Figure C.1 shows the class diagram for the intended system and is followed by a detailed description of each class.

MetaDataUpdater

This is the central class of MUS. It initiates the data structures and the GUI. The constructor initializes the station list, the GUI and the database interface.

addManual

Manually entered data is passed on to a Station object that will parse it.

approveStation

Approves one station, this will only be applicable to stations that have been marked as major change.

readFile

This method reads metadata from a WMO list, the parameter is a file pointer to a list. The line is passed on to the populate method of a Station object that parses the information.

getListModel

Returns the list model for the station list data structure, any change to the list will be reflected in the GUI. The list model is of the type DefaultListModel and follows the MVC paradigm.

validateStations

All stations on the station list has to be checked against ROAD. First of all the stations will be identified using the WMO number since these are unique in the metadata sources. If the corresponding WMO number is found in the database then the station name will

be checked just to make sure that the WMO number is not misspelled. If the WMO number cannot be found the station name will be used instead but if the name is misspelled this may lead to errors in the database.

A station that is not changed will be removed from the list. A changed station will be marked as changed and a station that is not found will be marked as new. The changed stations will also be marked as major or minor change.

createPatch

SQL queries for all stations on the station list that are marked for patch will be created. The version document will be created as well according to SMHI standards and if needed a roll-back patch will be created.

GUI

The GUI draws a graphical user interface and calls methods in the MetaDataUpdater class. The station list is graphically presented as a JList and the data model is retrieved from the station list. There is no communication between the GUI and the station list class.

StationList

This class keeps station metadata for all stations that has been read from the metadata sources. Stations with their metadata are kept in a list. After a station has been compared with ROAD it will either be moved to a list containing stations that will be included in the patch or it will be deleted.

Station

The station class represents one weather station and contains the metadata about the station. The station also contains the queries that are executed against ROAD as well as methods for comparison. The idea is that if the program will be used with a different data model or metadata in the future only the station object will need to be changed.

populate

A station is instantiated with a string that contains station metadata. The string is parsed and the internal variables are populated. The string will be parsed according to the WMO list syntax first and if it fails the string will be treated as a manually entered string and be parsed with regular expressions. The parser is very strict and does not tolerate errors with the only exception that it can handle geographic coordinates entered in both minutes, degrees, seconds and decimal form.

createQuery

This method creates the SQL query that will be included in the patch. Depending on whether or not the station is a new one either an update statement or an insert statement will be created. The station has to be approved or else no query will be created.

alike

Compares the station supplied as parameter with current one and returns true if they are considered alike. Alike means that their WMO numbers (or source number) are the same and the names are alike.

compareTo

Compares two stations and orders them according to their name.

equals

Returns true if two stations are equal.

populateStationInfo

Takes a stationInfo object as parameter and calls the populateFields method with necessary parameters.

printStation

Prints all station parameter. Used for debugging purposes.

removeEqualities

Takes a station as parameter and removes all equal parameters.

stationInfo

Used by the GUI to present information about a station and let the user change the parameters. Changes are parsed by the station and if the metadata is not accepted by the parser it is discarded.

DBInterface

The DBInterface keeps the connection to ROAD and contains functionality for retrieving data.

getStation

A station is the parameter and it returns the corresponding station from the database.

getLikeStation

Retrieves a station from the database that matches the WMO number (or the source number) of the station supplied and partially matches the station name.

getStationByID

Retrieves a station from the database that matches the WMO number (or the source number) of the station supplied.

getStationByName

Retrieves a station from the database that matches the name of the station supplied.

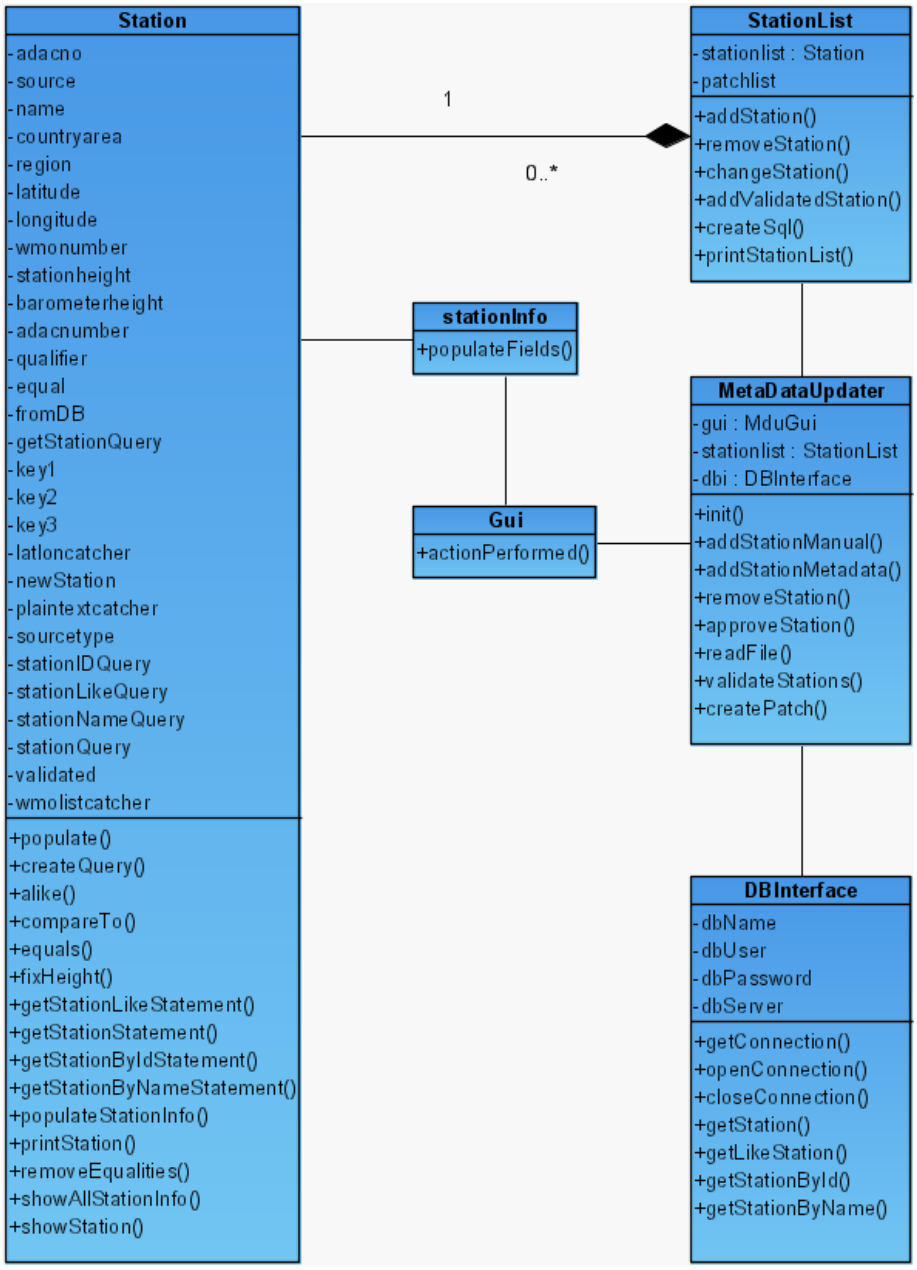

Figure C.1: Class diagram of MUS

Appendix D

Requirements

Functional requirements

Requirement:	FR1 Use case: UC1
Title:	Read WMO list
Description:	The system should be able to read the WMO list
Fit criteria:	System capable of parsing WMO list
Priority:	High

Requirement:	FR2 Use case: UC1
Title:	Read e-mail
Description:	
Fit criteria:	System able to read e-mails and extract station information.
Priority:	Low

Requirement:	FR3 Use case: UC1
Title:	Read interactive updates
Description:	The system should be able to read interactive updates formatted as plain text and extract station information.
Fit criteria:	System able to read and parse formatted plain text.
Priority:	Medium

Requirement:	FR4 Use case: UC1
Title:	Input validation
Description:	Metadata update information should be checked for misspellings and they should be corrected.
Fit criteria:	System is able to discover misspellings and correct them.
Priority:	Medium

Requirement:	FR5 Use case: n/a
Title:	Read SQL
Description:	System should be able to read SQL queries.
Fit criteria:	System able to read SQL input.
Priority:	Low
Origin:	TSF

Requirement:	FR6 Use case: n/a
Title:	SQL syntax validation
Description:	The system should check syntactical correctness of SQL queries.
Fit criteria:	System able of checking SQL for syntactical errors.
Priority:	Low

Requirement:	FR7 Use case: n/a
Title:	SQL input information validation
Description:	The system should check the metadata in an SQL input against ROAD and the rules.
Fit criteria:	System able to decide if data entered as SQL is valid.
Priority:	Low

Requirement:	FR8 Use case: UC1
Title:	Detect changed stations
Description:	The system should detect which stations that should be updated.
Fit criteria:	System able to find updated stations.
Priority:	High

Requirement:	FR9 Use case: UC1
Title:	Validate updates
Description:	The system should make sure that new station information does not contain erroneous information.
Fit criteria:	System is able to discover if the new metadata is valid.
Priority:	High

Requirement:	FR10 Use case: UC1
Title:	Create SQL queries
Description:	The system should create syntactically correct SQL queries.
Fit criteria:	System creates syntactically correct SQL queries.
Priority:	High

Requirement:	FR11 Use case: UC3
Title:	Create patch
Description:	The system should create a correct patch.
Fit criteria:	A correct patch is produced every time.
Priority:	Medium

Requirement:	FR12 Use case: n/a
Title:	Extendability
Description:	The system should be extendable.
Fit criteria:	System easy to maintain easily be extended or updated.
Priority:	Medium
Origin:	SMHI guidelines

Non-functional requirements

Requirement:	NFR1 Use case:
Title:	Easy to use
Description:	The system should be easy to use by SMHI employees.
Fit criteria:	
Priority:	High