Lehrstuhl für Systeme und Informationsverwaltung

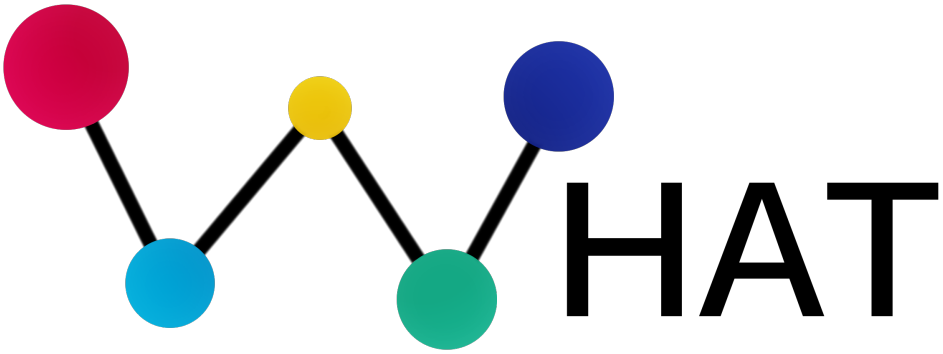
Prof.Dr.Ing. Klemens Böhm
Hoang Vu Nguyen
Marco Neumann

Karlsruher Institut für Technologie

Praxis der Softwareentwicklung (PSE)
WS 2012/2013

Visualizing and Statistically Analyzing Access Behavior to Scientific Databases

QA & Testing



WareHouse Analyzing Tool

March 2, 2013

Functional Specification	**Alexander Noe**
Design	**Jonathan Klawitter**
Implementation	**Anas Saber**
QA / Testing	**Nikolaos Alexandros Kurt Moraitakis**
Final	**Lukas Ehnle**

E-M@il: pse10-group14-ws12@ira.uni-karlsruhe.de

1 Introduction

This document is about the testing and quality assurance phase of the program WHAT WareHouse Analyzing Tool. It contains descriptions of what was tested and how, which results tests produced and what bugs or problems were found with them. There are also some comments about the programming and code style in general.

The tests are grouped by tasks of the program, which mostly were also separate packages. The sections of this document follow this structure and complete it with the general sections concerning the whole program.

The functions which have to show the correct web pages or produce correct SQL query where tested by hand. This means that for a configuration and data we tested, if correct looking MySQL queries where produced and correct web pages or charts where shown. But more about this in the specific sections.

TODO: Travis and such stuff @niko

The last section (11) is about complying with the requirements made in the functional specifications.

Contents

2 General

This section is about things concerning the whole system.

2.1 Coding style

In general value on good coding style was set. This is mostly the aspect of task division to classes and methods. To implement this some classes were totally redesigned and lot's of new classes created. This also helped another important aspect, object oriented programming. So for example a chart host storing all tables, names and filters as Strings and sets, developed to a class containing only one String per axis, Filter objects, which wrap dimension and other objects, and a Measure object, to hold all information, needed for MySQL queries for this chart.

Another aspect was the reuse of code. Reviewing the code, showed that some packages need the same functions, e.g. for reading from JSON object. To bundle this functions and decrease duplicate code some utility classes (3.1) were created.

See the sections of the packages for details.

2.2 Structure

We noticed that packages and folders in the application folder app from the play framework the application and the web tier weren't seperated. This separation was arranged. The data access tier was considerd to be a sub-tier of the application tier.

Change detail

- Restructure folders and packages to fulfill tier separation

2.3 JavaDoc

JavaDoc wasn't written strictly for all packages, classes, methods, attributes and constants. But with the help of CheckStyle (2.4) all of the laziness was found and the JavaDoc written. The quality of JavaDoc is always a matter of opinion, but in the view of the writers it's adequate to it's need.

Change detail

- Missing JavaDoc written

2.4 CheckStyle

When starting CheckStyle first with the standard checks there were about 7290 warnings. Searching where this all came from, we found out, that the library of GeoIP from

MaxMind, which was included as a package to make some adjustments, produced 6060 of them. So there seemed to be about 1230.

Adjusting the CheckStyle configuration to checks we thought were useful left only 535 warnings. Those were mostly handled. Only the warning that there should be no hard coded Strings, where ignored, where HTML code was build.

Change detail

- CheckStyle warnings fixed

TODO: entscheiden was man damit machen will. . . data : 82 config: 57 chart: 85 what: 29 web: 130 controllers: 60 parser: rest

2.5 Error messages

Where not really needed stack traces were replaced and meaningful error messages printed.

Change detail

- Replace stack traces with error messages

3 Facade, Helper

In the highest package of the application tier the Facade and utility classes are placed.

3.1 Utility classes

To bundle some functions needed in more than one package some utility classes were created. They also made error handling and testing easier.

Change detail

- New utility classes created

This classes are described below.

3.1.1 FileHelper

Reading configuration files and log files both need some methods to get the file, check its existence, rights and file extension. This functions where collected in the class *FileHelper*.

3.1.2 JSONReader

The configuration and parts of the communication between web tier and data tier is accomplished with JSON objects. Also reading and extracting from them is bound with possible exceptions and try-catch blocks. To reduce duplicate code and provide easier access the *JSONReader* class was created.

3.1.3 Printer

The class *Printer* was created to unify error, failure, problem, test and success messages.

3.2 Facade

The facade's main functions are initializing everything and then direct incoming requests to the specific mediators.

One problem found was that the creation of tables for the database caused serveral problems for testing and when they changed. So the creation of them was solved dynamically, so that you just had to drop the old tables and when starting the system, it would create them.

Change detail

- Automatic check if tables exist and creation of them

3.3 Testing

The facade is tested via its use. If it wouldn't work or show any problems nothing between web tier and application tier would work.

The Printer works fine.

The other classes have a JUnit test class.

Test details

- test FileHelper with wrong file and nonexisting files
- test whether needed Files for the system exist via FileHelper
- test all methods of JSONReader with correct keys
- test all methods of JSONReader with incorrect keys

4 Configuration file

This section is about the part of the system reading the configuration file and handling its information.

4.1 Requirements

We decided that every configuration file and the order of the rows have to follow this requirements:

- starts with time and time rows contain the String 'time'
- has an IP row which follows time
- any other rows, which must be dimension (level > 0) if they are not of type integer or double
- rows of one dimensions follow each other with rising level
- integer or double rows are automatically measures
- at least 1 measure
- statement is at the last position

For configurations following this requirements the system should run successfully now. Some of the things are checked, some would cause a crash of the build anyway.

Change detail

- Check the requirements to a configuration file

4.2 Creation

The reading of the configuration file was made easier with the *JSONReader* (3.1.2). Because of this the easy creation of the *ConfigWrap* was placed in the *ConfigWrap* itself as static factory without the need of any helper class.

Change detail

- Static factory machine for *ConfigWrap* without special helper class

4.3 StringRow

The *StringRow* had unneeded functions. The idea was to store all possible Strings of a row in it. But not really knowing what could be content of the log files, the Strings are read directly from the log files and stored in the warehouse. So the old functions got unneeded.

Change detail

- Kill unneeded functions of *StringRow*

4.4 Dimension content trees

The actual existing content of a String dimension is requested from the warehouse and stored. This was made with a construct of *HashMap*s and *TreeSet*s of String containing itself or not. It was a bad approach to build a tree.

This was changed with a new class *DimKnot* which forms tree structures. Instances of it also contain information about the level they are, which makes it much more easy to build MySQL queries dynamically from them.

Change detail

- *DimKnot* tree structure instead of bad *HashMap* + *TreeSet* approach

4.5 Space in names

It was not possible to have spaces in names of rows, dimensions or a configurations. This produced incorrect table and column names in the database.

Change detail

- Allow row, dimension and configuration names with space

4.6 Child dimensions

All *DimRow*s had a field for trees and time interval. To make this more efficient and object oriented it was changed by adding to child classes of *DimRow*.

Change detail

- Child classes *TimeDimension* and *StringDim* for *DimRow*

4.7 Testing

The configuration file and the test configurations where parsed and printed. This seems test enough that it is read correct.

The trees of the DimKnots where tested by checking what values are actually in the database and if they all appear on the web page.

Test details

- Two new test configurations

- Parsing with 2 valid test configurations
- Look at web page after parsing if everythin fits too
- Parsing with invalid test configurations

5 Parser

5.1 Task determination

When having too many threads (10+), sometimes a thread stopped working. This caused the counter for finished tasks to be stuck under the number of threads and the program not to get finished. So a watchdogtimer was implemented which replaces a thread by a new one, if said thread doesn't finish a line for a predefinied amount of time. (standard = 2 seconds)

Change detail

- Added watchdogtimer.

5.2 Negative or too high poolsize

The parser works with a pool of threads, which are created when parsing. In the Parser-Mediator there is a variable poolsize, which is the used poolsize for parsing. When using a negative or a really high number (tried Integer.MAX_VALUE), the program crashes. So we implemented a check for the poolsize, which looks if it is between 1 and 50 and stops parsing if it isn't.

Change detail

- Check for invalid poolsize before parsing.

5.3 NullPointerExceptions when correct file doesn't exist

sp_parser.Logfile had a check, if the file which is given to it is actually existent. It used the file-class from java and got a `NullPointerException`, which was caught and used as an indicator, that there is no file for the correct name.

This worked, but to improve our program, we used a class called *FileHelper* (3.1.1). It didn't return a `NullPointerException` when the file doesn't exist, so this check got overrun and the parser 'thought' that there is a correct file, which was not the case. This caused the program to crash.

Change detail

- Use and correct use of *FileHelper*

5.4 Whitespace in entries

Some log files contain unneeded whitespace, which produce incorrect data. This was fixed by adding the `trim`-command in StringRow.split() and StringMapRow.split()

Change detail

- Trim the strings before sending them to the database.

5.5 Anonymous Proxy

GeoIp sometimes returned ’Anynomous Proxy’ instead of nothing for cities or countries it didn’t found. This gets replaced by ’other’ to fit in with the other undefined countries and cities.

Change detail

- Check for anynomous proxy

5.6 Check for missing or empty parts

The parser now checks whether a part of the line parsed is missing or empty. In this case the line is ignored and the count for incorrect lines gets incremented.

Change detail

- Check for missing or empty part in every line
- Flawed lines are not accepted

5.7 Only accept when enough lines submitted

Added `final static double CORRECT`, a double between 0 and 100, which indicates how many percent of the lines need to be submitted correctly for the ParserMediator to return true, singaling parsing was successful.

Change detail

- Only accepts when more than a specific amount of lines are accepted.

5.8 Reset

The parser now resets after he finished parsing a logfile, so that another file can be parsed.

Change detail

- added a reset-method in the ParserMediator

5.9 Testing

Besides the JUnit tests, described below, there were same test by hand. Parsing log files with just a few lines into a empty database. Then compare the content by hand.

Test details

TODO group them, like the comment in the JUnit class TODO not newest version!!

- smallParseTestStandardSize - parse 10 lines, standard (5) poolsize
- smallParseTestSize10 - parse 10 lines, poolsize 10
- smallParseTestTooBig - parse 10 lines, poolsize 100 -> Error #2: ParserMediator.poolsizeParsing is bigger than 50.
- negativePoolsize - parse 10 lines, poolsize -2 -> Error #1: ParserMediator.poolsizeParsing is smaller than 1.
- nonexistentFile - try to parse nonexistent file -> Error #5: The path is wrong.
- wrongFile - try to parse from an empty .txt file -> Error #5: The path is wrong.
- flawedFile - try to parse from a file with a typo in "type" -> Error #11: The configuration file got a different format.
- mediumParseTestStandardSize - parse 1000 lines, standard (5) poolsize
- mediumParseTestSize50 - parse 1000 lines, poolsize 50
- bigParseTestStandardSize - parse 30k lines, standard (5) poolsize
- smallParseTestEmptyLine - parse 10 lines with an empty line in between
- testIntegerMistake - parse 10 lines with year "2013apples" instead of "2013"
- testMissingPart - parse 10 lines with one line with a missing statement
- testEmptyPart - parse 10 lines with one empty statement and one empty year
- doubleParsing - parse 1000 lines twice from seperate files

6 Chart requests

6.1 Object oriented programming

The origin design for this package was dropped in the implementation phase. There was neither a visitor pattern nor classes to wrap the informations needed efficient. This decision was withdrawn.

6.1.1 Chart request workflow

The parameter lists in the methods got longer and longer when implementing the missing filter features and the charts with two dimensions.

Something we didn't want to do first, solved the problem. The problem with the *ResultSet* closing, when closing the statement, made it necessary to write the *JSONObject* of the charts in the data access tier. So the decision was made to send the whole chart instance down to the data access tier.

Change detail

- Chart itself is parameter of chart requests

Instead of requesting all the information now from the chart and create the MySQL queries, this got changed to a more elegant way. The chart itself returns query parts for the things like 'select', 'join' or other.

Change detail

- Creating of query parts in the chart

But it remained a bulge of information stored improperly. This was solved with some classes described below.

6.1.2 Filter class

Instead of requesting all query parts from the config, like table names and keys, a filter stores the dimension on which it is based. First just the lowest level of the filter values was stored, but this was detected to be incorrect. A database may be child of more than one server and a city with the name 'x' may be city in more than one country. For correct filtering, referring to the things selected on the web page, trees where made for this values to build correct filter query parts.

Change detail

- *Filter* class to wrap information of a filter
- trees of filter values instead of sets without parents

6.1.3 Measure class

The approach to store just a string for the measure was neither good style nor made it possible to have other aggregations than `count(*)`. This was solved by the new class *Measure*. It made it also possible to write more information in the JSON sended back to the web page about what the returned chart is.

Change detail

- *Measure* class to wrap information of a measure

6.2 SQL injection

It could have been possible to run SQL injections against the database by using filter values as injection holes. Comparing the values of filters parsed from a char request JSON with the actual values in the warehouse should have closed this hole.

Change detail

- Protection against SQL injection

6.3 Testing

Correct reading of the JSON-Objects was tested by explicte use of filters, axis and measures, then printing the requests and compare with the resulting MySQL queries. We couldn't see the usage for any automatic tests there.

Creating the queries from the chart host was tested in the same way.

7 Data access

7.1 Connection pool

Known problem: No driver there if to often started and not returned. TODO

7.2 TODO

TODO

8 Web

Web seite betreffende Dinge. Meiste wird durch sehen getestet. Aber eben auch Dinge die sicherstellen, dass bei falscher Config nix passiert, dass ohne daten nix passiert

charts einfach so einfuegen

adminzeugs

sprachen

etc

9 Charts

Alles was mit den Charts also, der Anzeige D3 und so zusammenhaengt.

10 Performance

We should test the performance and than check how we could get it better.

10.1 Loading

Parsing of 1k lines with old upload schema:

1. 19048ms
2. 22033ms
3. 11020ms

Parsing of 1k lines with new upload schema:

1. 11016ms
2. 10017ms
3. 11017ms

Lukas hat noch mehr Zeiten, werde diese dann Eintragen und was gescheites drauss machen. Das oben war nur um meine zu speichern.

Werd dann noch Test machen, wie lange es dauert, ganze Monate hochzuladen und danach, wie lange eine Chart anfrage braucht. Sobald der Upload wieder geht

10.2 Extracting

11 Requirements

TODO: wie gut wir die sachen erfuellt oder nicht erfuellt haben, was es mehr gibt oder fehlt und so zeugs