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

WHAT
WareHouse Analyzing Tool

March 4, 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 the components of the program tested, the individual tests, the produced results as well as the bugs and problems uncovered in the process. There are also some comments about the programming and code style in general.

The tests are grouped by tasks of the program, which also mostly were in 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 were tested by hand. This means that for a configuration and data, we tested whether correct looking MySQL queries where produced and correct web pages or charts where shown. But more about that in the specific sections.

We will also describe how we made use of the testing capabilities of the build managers, and the travis continuous integration service.

Additionally, we mention the steps we took to secure the program.

The last section (12) is about the compliance with the requirements set during the functional specification phase.

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 mostly concerns the aspect of task division to classes and methods. To implement this some classes were totally redesigned and a fair number of new classes was created. This also helped facilitate another important aspect, object oriented programming. So for example a chart host storing all tables, names and filters as Strings and sets, was changed to a class containing only one String per axis, Filter objects, wrapping dimension and other objects, plus a Measure class, to hold all information needed for MySQL queries on this chart.

Another aspect was the reuse of code. Reviewing the code showed that some packages needed the same functions, e.g. for reading from JSON objects. To bundle these functions and decrease duplicate code some utility classes (4.1) 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 separated. This separation was arranged. The data access tier was considered 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 uncovered 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 its needs.

Change detail

- Missing JavaDoc written

2.4 CheckStyle

When starting CheckStyle, at first there were about 7290 warnings with the standard checks. Searching where they came from we found out that the MaxMind GeoIP library, which was included as a package to make some adjustments to it, 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. The only warnings that were ignored where those that claimed that there should be no hardcoded Strings. However, we use hardcoded strings at the views of the play framework, to produce HTML code.

Change detail

- CheckStyle warnings fixed

Additionally, 28 Javascript warnings were found, 21 of whom in the code that creates the charts. This relatively low number can be partially attributed to the usage of jshint instead of jslint.

2.5 Error messages

Stack traces were replaced with meaningful error messages where needed.

Change detail

- Replace stack traces with error messages

3 Tooling and workflow

As mentioned above, we used JUnit for testing. Our tests are located in the test folder, and get executed with the ’play test’ command (which just ends up executing ’sbt test’).

3.1 Travis

The continuous integration service travis was set up and heavily utilised. Each time a new commit was made to the repository on github, a computer from travis tried to successfully compile our program. This includes cloning the repository from github, downloading the dependencies (such as Java play), creating the database and users needed for the warehouse, as well as building the program.

It then executed ’play test’ as part of its building process, thus run our tests and notified us if they passed. As a bonus, travis builds on Linux computers, and since many of us used windows during the development process this helped with cross-platform testing.

4 Facade, Helper

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

4.1 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

These classes are described below.

4.1.1 FileHelper

Reading configuration files and reading log files share methods.

This includes getting the file,checking for its existence, rights and file extension. These functions were collected in the class *FileHelper*.

4.1.2 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.

4.1.3 Printer

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

4.2 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 and change of database tables caused several problems for testing. Their creation was solved dynamically - one just has to drop the old tables and when the program starts, they get created automatically.

Change detail

- Automatic check of the table existence and their creation.

4.3 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 non-existing 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

5 Configuration file

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

5.1 Requirements

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

- starts with time rows
  - rows contain the String 'time'
  - 6 time rows
  - order: year, month, day, hour, minute, second
- 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 unsuccessful build anyway.

Change detail

- Check the requirements to a configuration file

5.2 Creation

The reading of the configuration file was made easier with the *JSONReader* (4.1.2). Because of this the easy creation of the *ConfigWrap* was placed in the *ConfigWrap* itself as a static factory without the need of any helper class.

Change detail

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

5.2.1 Unneeded attributes

There were some unneeded and/ or senseless attributes for all rows in the configuration. They got deleted and replaced by 'N/A' in the reading process.

Change detail

- Set deleted and unneeded row attributes to 'N/A'

5.2.2 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

5.3 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 obsolete.

Change detail

- Remove uneeded functions of *StringRow*

5.4 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 on, which makes it much easier to build MySQL queries dynamically from them.

Change detail

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

5.5 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*

5.6 Testing

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

The trees of the *DimKnot*s where tested by checking what values are actually in the database and whether 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

6 Parser

6.1 Task determination

When having too many threads, sometimes a thread stopped working. This caused the counter for finished tasks to be stuck below 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.

Change detail

- Added watchdogtimer

This worked fine till other big problems showed up and no thread worked at all. This caused a endless creation of new threads.

Change detail

- Stop endless creation of new threads
- Kill parsing request if to many threads fall asleep

6.2 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.

6.3 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* (**??**). 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*

6.4 Whitespace in entries

Some log files contain unneeded whitespace, which produced 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.

6.5 Anonymous Proxy

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

Change detail

- Check for anynomous proxy

6.6 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

6.7 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.

6.8 Reset

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

Change detail

- added a reset-method in the ParserMediator

6.9 Testing

Besides the JUnit tests, described below, there were some manual tests. One of those was parsing log files with just a few lines into an empty database, and then comparing the result.

Test details

- Mistakes
  - testIntegerMistake - year "2013apples" instead of "2013"
  - testMissingPart - with a missing statement
  - testEmptyPart - with one empty statement and one empty year
  - smallParseTestEmptyLine - with empty lines in between
  - flawedFile - try to parse from a file with a typo in "type"
- Poolsize
  - smallParseTestSize10 - parse 10 lines, poolsize 10
  - smallParseTestTooBig - parse 10 lines, poolsize 100
  - negativePoolsize - parse 10 lines, poolsize -2
  - mediumParseTestSize50 - parse 1000 lines, poolsize 10
- Just parsing
  - smallParseTestStandardSize - parse 10 lines, standard (5) poolsize
  - mediumParseTestStandardSize - parse 1000 lines, standard (5) poolsize
  - bigParseTestStandardSize - parse 10k lines, standard (5) poolsize
  - doubleParsing - parse 1000 lines twice from seperate files
- Ignored
  - veryBigParseTestStandardSize (IGNORED) - parse a whole month, standard (5) poolsize, ignored because it may take hours to finish.

7 Chart requests

7.1 Object oriented programming

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

7.1.1 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 at 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 creating 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.

7.1.2 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 were 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

7.1.3 Measure class

The approach to store just a string for the measure was neither good style nor was 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 sent back to the web page about what the returned chart is.

Change detail

- *Measure* class to wrap information of a measure

7.2 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

7.3 Two axis of same dimension

By playing with the chart we detected that queries for charts with x- and y-axis of the same dimension incorrect queries were created, resulting in not getting a chart.

Change detail

- Make it possible to create charts with x and y of same dimension

7.4 Testing

Correct reading of the JSON-Objects was tested by explicte use of filters, axis and measures, then printing the requests and comparing 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.

Test details

- Print read data and check for correctness corresponding to web page request
- Print queries and check correctness

8 Data access

8.1 Tables

Instead of creating tables by hand for every configuration file, functions were added to make it possible to create tables dynamically from a configuration. This depended on another feature, storing the configuration files for which the tables had already been created.

Change detail

- Add feature to create tables dynamically
- Add feature to store and check if tables for configuration are already created

There showed up problems with the time and location table which are always the same. Therefore the creating query was changed to `CREATE IF NOT EXITS`.

Below are the queries that create the SkyServer tables.

```
CREATE TABLE IF NOT EXISTS fact_SkyServer
        (time_key INT(5), location_key INT(5), server_db_key INT(5),
         row_elapsed FLOAT,  row_busy FLOAT,
         row_rows INT(3), type_key INT(5));
CREATE TABLE IF NOT EXISTS dim_time
        (row_year INT(3), row_month INT(3), row_day INT(3),
         row_hour INT(3), row_minute INT(3),
         row_second INT(3), time_key INT(5) UNIQUE);
CREATE TABLE IF NOT EXISTS dim_location
        (row_country VARCHAR(40), row_city VARCHAR(40),
         location_key INT(5) UNIQUE);
CREATE TABLE IF NOT EXISTS dim_server_db
        (row_server VARCHAR(40), row_database VARCHAR(40),
         server_db_key INT(5) UNIQUE);
CREATE TABLE IF NOT EXISTS dim_type
        (row_type VARCHAR(40), type_key INT(5) UNIQUE);
```

Change detail

- Change table creation query to allow same tables for more configuration

If dimensions of different configurations have the same name - except time and location - incorrect work will be caused. The table names are created from the dimension names. It was decided to give the dimensions always a new name. If this is not wanted, it could be changed by concatenating the configuration name with dimension name and the dimension table short to create the table name.

8.2 Uploading

8.2.1 Query trunks

With every new request uploading a *DataEntry* the queries for fact and dimension tables were computed completely. To avoid this unnecessary work, the trunks of the queries now get precomputed.

Change detail

- Precompute trunks of upload queries

8.2.2 Generating keys with HashBuilder

For every dimension the key of a row is needed in the fact table. Instead of trying to upload the dimension data and get a auto generated key if possible, otherwise requesting the key by hand, the keys are generated with a *HashBuilder* now. The builder gets all items of a row and returns a hashed key, which is used for the upload of the dimension data and the fact table.

Change detail

- Generating keys with *HashBuilder*

8.2.3 No auto committing

Every dimension and the fact table rows were committed one by one. Decreasing the overhead the statement was set to no auto committing. This made it possible to collect all the upload queries in the statement and send them to the database all at once.

Change detail

- Use no auto committing statements for upload
- Execute all upload queries of on *DataEntry* at once

There was an attempt to use a second connection pool for the no auto committing connections. They got set to no auto committing once. But this didn't work out because it seemed that they didn't store this behaviour. So it was changed back to the old strategy changing every used no auto committing connection directly before the use.

8.3 Extracting

`(SELECT * FROM tablename WHERE ...) AS nickname` was used first to filter the dimensions. This was changed to the more easy to build and, as it seemed, faster version with `JOIN tablenames ON ...`. Making queries got easier because you just have to list the table names, the key filters to the fact table and the conditions of the dimensions.

Change detail

- Use `JOIN` instead of `SELECT *`

As mentioned before (7.1.1) the new approach with the chart host being a parameter of a chart request and the creating of query parts directly in the chart host made it more easy to build the complex queries. Below is a possible query.

```
SELECT count(*),  row_server
FROM fact_SkyServer AS  FT
JOIN  ( dim_server_db AS dim_server_dbID,
        dim_location AS dim_locationID,
        dim_type AS dim_typeID )
ON FT.server_db_key = dim_server_dbID.server_db_key
  AND FT.location_key = dim_locationID.location_key
  AND FT.type_key = dim_typeID.type_key
  AND ((dim_server_dbID.row_server = 'DR1_LONG' AND
         ((dim_server_dbID.row_database = 'BESTDR1' )))
     OR (dim_server_dbID.row_server = 'DR2_LONG' )
     OR (dim_server_dbID.row_server = 'ROSAT_QUICK' ))
  AND ((dim_locationID.row_country = 'Argentina' AND
         ((dim_locationID.row_city = 'Buenos Aires' )
         OR (dim_locationID.row_city = 'Cordoba' )))
     OR (dim_locationID.row_country = 'Canada' AND
         ((dim_locationID.row_city = 'Montreal' )
         OR (dim_locationID.row_city = 'Ottawa' )
         OR (dim_locationID.row_city = 'Vancouver' ))))
  AND ((dim_typeID.row_type = 'OTHER' )
     OR (dim_typeID.row_type = 'PHOTO' )
     OR (dim_typeID.row_type = 'QSO' ))
  AND STR_TO_DATE(CONCAT(CONCAT(CONCAT(CONCAT(
     CONCAT(CAST(row_year AS CHAR),','),
     CONCAT(CAST(row_month AS CHAR),',') ),
     CONCAT(CAST(row_day AS CHAR),',') ),
     CONCAT(CAST(row_hour AS CHAR),',') ),
     CONCAT(CAST(row_minute AS CHAR),',00') ) ,'%Y,%m,%d,%H,%i,%s')
   BETWEEN STR_TO_DATE('2012,4,7,12,14,00','%Y,%m,%d,%H,%i,%s')
   AND STR_TO_DATE('2013,2,11,18,19,00','%Y,%m,%d,%H,%i,%s')
GROUP BY row_server
```

8.4 Closing

There were some problems when the system was started often without restarting the MySQL server. The number of connections getting from a MySQL server is limited. With every creation of a *Facade* with a configuration file, all the mediators are also created, including the *DataMediator*. This triggers the creation of a connection pool. So when the system is started too often MySQL will run out of connections and the connection pool won't be created.

Starting the play server to often will face this problem. This was solved by adding a closing function executed before the system stops, which releases the *Facade*.

In the JUnit tests the *Facade* is reset with nearly every test. So the problem could be solved by releasing all connections when resetting the *Facade*.

Change detail

- Release connections when resetting *Facade*
- Release and reset *Facade* when stoping system

8.5 MySQL queries + Testing

The queries were produced in good conscience. This means it was tested in the MySQL workbench how the queries have to look and what would work. Knowing this the code producing this queries was implemented.

To test whether correct queries are produced they were printed for several requests and checked by hand if they are of the wanted format.

After uploads of small log files the tables were examined whether they contain all and the correct data. It was also checked whether the filter options on the web page contain all the possible data in the tables. For a small amount it was checked whether the requested data of charts was reasonable. Or for filters for which no data exists, whether the data returned is really empty.

Test details

- Print all queries and compare to wanted queries
- Compare data in tables to parsed log files
- Compare String on web page filters to data in tables
- Compare chart data with requests and tables

9 Web

9.1 web package

The package web was created and the packages controllers and views have been moved there to separate everything that is involved with the webpage and the frontend from the application logic in the what package.

Change detail

- added web package, moved views and controllers package to web package.

9.2 Chart history

One problem in displaying the history of the last requests, was to ensure that if an overview was sent to the user and he decided to check one out, that this chart history was still saved on the server side, even though there may have been new chart requests in the mean time. To solve this problem the class ChartHelper was added. It generates an overview of the last chart requests, if a user requests it. In the process it saves the JSON Objects together with a unique user id (uuid). If the user requests a history, the history number will be looked up for his uuid and returned.

Change detail

- added ChartHelper which manages chart history overviews and requests.

9.3 Possibility to add new chart types

A chart consists of an thumbnail for the index page, a configuration file, which saves how many dimensions can be displayed by this chart type, a javascript file, which contains the d3.js code to visualize the data and an optional stylesheet file to style the chart elements. To add new charts, you only have to add these files. But to ensure that no chart types which do not have all these files are displayed, their existence is now checked on application start. For this purpose some methods were added and changed. There are methods to get all valid charts, to get the thumbnail name of a chart, to get the number of dimensions a chart can display and to check if a chart has a stylesheet file or not.

Change detail

- changed and added methods.

9.4 Reuse code in charts

The code of different charts has to be independent, to ensure that additional charts can be added dynamically. But then we wanted to reuse the code from the bubble chart&scatterplot chart in the scatterplot, because they are identical except for the bubble radius. The solution was, that the javascript from scatterplot adds the javascript and stylesheet files from bubble chart&scatterplot dynamically itself, without any hardcoding on the side of the server. While this is by no means a perfect solution, e.g. if the bubble chart&scatterplot is removed, then the scatterplot stops working, it fulfills it's purpose and allows the web server to still handle all chart types and files the same way.

9.5 ChartHelper class

Although the play framework template engine can call Java functions, it was not dynamic enough to design our chart pages solely with its functions. So the ChartHelper class was added, which creates the selection boxes for all charts in every supported language, with all selectable options, which are contained in the warehouse. These are then saved as Html objects, so they don't have to be created on each call to a chart page. The selection for a specific chart can be requested with the getOptions() method and it returns the selections in the right language, as extracted from the user's session.

Change detail

- added ChartHelper class with methods to create option selections for charts.

9.6 languages

The play framework itself supports UTF-8 encoded strings, so using multiple languages shouldn't have been a problem. But as we had to notice the localisation didn't work for languages that don't use the latin alphabet. This was a problem with the Java property files and PropertyResourceBundle, where the localized strings are saved and which natively only supports ISO-8859-1 encoding. To solve this problem the class UTF8Control was added.

Change detail

- added web.controllers.UTF8Control, which allows reading UTF-8 encoded property files.

9.7 Testing

The testing of the webpage was done by hand as the website itself does little besides displaying things graphically. Additionally this decision was enhanced by the fact that there are relatively few functions on the webpage to test. The webpage has been tested in Chrome, Firefox, Chromium but has also ran successfully on Internet Explorer 10 and even some mobile devices.

10 Charts

The charts saw a fair amount of bug fixes and improvements as well.

10.1 Bug fixes

10.1.1 Type of data

The SkyServer contained a significant amount of categorical data. However, their representation on the axes used to cause problems. The charts are now more robust and work with both numerical and categorical data.

Change detail

- Make charts robust against various data

10.1.2 Error

Errors are now thrown to differentiate between the chart with no data, and the chart with an error, or incorrect data (which the user will hopefully never experience).

Change detail

- Show error messages on the chart pages.

10.1.3 Boring bugs

Boring bugs, such as the text on the data was off by a few pixels were fixed too. Other bugs include

- Bar charts not working correctly with only few data points.
- There wasn't enough space for all text labels on bubble charts.
- There wasn't enough space for all the text labels on pie charts.
- There wasn't enough space for the text labels on the axes of scatterplots.
- And so on.

Change detail

- Bar charts with few bars are shown correctly.
- The small labels aren't shown on pie charts (they are shown on mouseover)
- Only as much of the label that fits in each bubble is shown on bubble charts.

10.2 Aesthetics

The charts were made prettier.

Change detail

- Tooltips, or text that appears once the mouse hovers over a data point was added to every chart
- The same fonts were used consistently.
- The CSS were made more elaborate.
- The text now gets rotated on a number of cases to fit more data points. It also makes the charts prettier

10.3 Testing

We tested by hand. Something that made testing in this section easier is that it is possible to know how a chart is suppose to look when correct, as well as the charts consisting of different more or less discrete parts - for example, it is easy to tell when just an axis is broken. Also, it was always nice returning to a chart to see how nicely someone improved the CSS this time.

We also let a person external to the project test and use the charts.

Test details

- Test by selecting many different axis, measure and filter combinations
- Let project external people use charts

11 Security

While our program does not store any user data, we still took precautions to secure it against potential attackers by examining potential attack vectors, and took action to prevent them.

11.1 User Input

There is no free form text user input, and it never gets displayed back. In addition, because of the tiered architecture, the data from the tier above gets checked multiple times for validity. This happens both at the web server level, and at the database access level.

See 7.2 to see a protection against SQL injections.

11.2 Authentication

We use the Java play authentication mechanisms. The user session has a cookie named PLAY_SESSION, which is a long hash, different between computers.

12 Requirements

12.1 Test cases

It showed that the test cases specified in the functional specification don't really guarantee proper work of the system. They just show that the core criteria are fulfilled.

So we made new and more specific tests as described in the corresponding sections. With them the whole system should be covered by tests.

12.2 Functional requirements

It can be said that nearly all functional requirements are fulfilled.

The exclusion being the following: /F110/ Show information about selectable variables With the decision to make all work based on configuration files this got obsolete.On the other hand, from a pragmatical standpoint, the names of the dimensions and measures are pretty expressive and self explanatory.

- /F110/ Show information about selectable variables

12.3 Nonfunctional requirements

12.3.1 Usability

The requirements for usability were the following.

- /N10/ Minimalistic GUI design
- /N20/ Responsive GUI
- /N30/ GUI that is easy to get used to

These are hard to quantify and are to some extend subjective, our highly biased opinion being that the webpage looks great and that they are fulfilled.

However

- The website has barely not functionality except the needed one.
- From a navigational aspect, it isn't any different than a normal website. (Thus if a user can use a normal website they can use our program, too.)

12.3.2 Swiftness

Not using OLAP cubes the chart requests get slower with more data in the warehouse.

Changing the loading of *DataEntry*s (8.2.3) increased the speed of parsing a lot. See the number of lines and the time in milliseconds for the old and the new way.

# of rows	old	no auto commit
10000	14 017	13 016
10000	11 007	9 014
10000	12 005	10 008
10000	11 007	9 012
10000	12 017	9 007
100000	105 009	92 012
100000	106 009	78 010

This shows a decrease of about 18%.

Parsing a bigger file with 1146315 lines showed that it takes about 11.23ms per line.