**Lehrstuhl für Systeme und Informationsverwaltung**

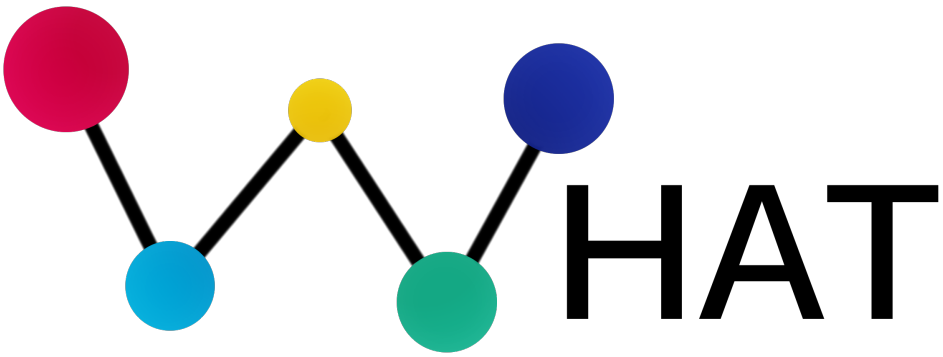
Prof.Dr.Ing. Klemens Böhm
Hoang Vu Nguyen
Marco Neumann

Karlsruher Institut für Technologie

**Praxis der Softwareentwicklung (PSE)**
WS 2012/2013

Visualizing and Statistically Analyzing Access Behavior to Scientific Databases

# Implementation




WareHouse Analyzing Tool

January 25, 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

# Contents

# 1 Changes in web tier

# 2 Changes in Application tier

## 2.1 Parser

### 2.1.1 ParserMediator

*threadPool* : Added a variable poolsize and a method *setPoolsize()* to set it to another number. If the poolsize is smaller than 1 there will be an error displayed.

*readyQueue , addToDatabase()* : Deleted, because the ParsingTask sends the finished dataEntry-Object directly to the DataTier.

*entryBuffer , extractLine()* : Deleted, because the ParsingTask requests a new line as soon as it needs it.

*configDB* : Renamed to ConfigWrap.

*parseLogFile(String str)* : Deleted configDB, because ParsingTask knows about the Parser-Mediator and can request it by itself.

*error(String str), boolean fatalError* : Added a new error() method, which sets fatalError to true, which will make parseLogFile(String str) return false to the Facade.

*increaseFT(), increaseLinedel()* : Added new methods to count finished threads and deleted lines.

*createThreadPool()* : Creates a new threadPool with *poolsize* threads and starts it.

*ParserMediator(ConfigWrap cw)* : Creates the parserMediator.

### 2.1.2 LogFile

*readLine()* : readLine() had bigger problems than expected, because the SQL-statement may contain several endOfLines. it's now fixed by setting a mark, reading another line, looking if it is actually a new line and if it is, resetting the mark and returning the complete line with the whole statement.

*Logfile(String path, ParserMediator pm)* : Checks if path is actually a valid csv-file and calls the Verification tool to check the correctness of the formatting in the file.

### 2.1.3 Task

Deleted, because there are only ParsingTasks, no LoadingTasks and so there is no need for a superclass.

### 2.1.4 LoadingTask

Deleted, because the parsingTask sends the DataEntry directly to the Data Tier making the LoadingTask obsolete.

### 2.1.5 ParsingTask

*ParsingTask(ParserMediator pm)* : Creates a new ParsingTask.

*splitStr* : Added the splitted string to the attributes of the ParsingTask so that it doesn't have to be splitted by every tool.

### 2.1.6 DataEntry

The DataEntry was completely remade to improve the general usability and multifunctionality. It now has only an Object-Array of variable size which is filled due to the Config.

### 2.1.7 VerifyTool

Renamed to VerificationTool.

Completely remade the Verification tool, because the verify-part is already done by the splitting tool. Instead it checks, if the file is actually in the format from the config.

### 2.1.8 LocationTool

Renamed to GeoIpTool

*setUpIpTool(ParserMediator pm)* : Sets the GeoIPTool up and checks if the geoLiteCity.dat - file which is needed for this tool is at the correct position.

### 2.1.9 SplittingTool

Splitted *split(ParsingTask pt)* in 3 parts to create smaller methods.

# 3 Working with a config

# 4 Changes in Data and Data access tier

# 5 Things that went wrong

A biased and last minute perspective on the problems we faced and the things we learned during the implementation phase

Since everybody else is busy bringing the program to run, I wrote this. //It's still a part of teamwork, right? //Please edit and censor and change The things we learned and the problems we faced can be grouped into technical, and not technical. I will mainly focus on the non technical, because it turns out they were actually more important in the end. This is shocking, and I did not expect this when I chose to study computer science.

Technical -git. We learned to use git, more or less, although it still ocassionaly annoys us. It is not unusual that someone still complains in the internal skype chat once a day. -libraries Since we split the areas of responsibility quite early on, everyone more or less used a different library or framework and also had to learn different things, among them minor quirks and annoyances. Most are not really worth mentioning. //Do we have any worth mentioning? //Do we have any nice perls of wisdom, or (my)sql riddles?

Non technical

automation We lost so much time trying to get software to install. (We still didn't manage to install some of it, hello, mondrian!) Automated dependency management sounds like a dream. The building process, however, and the configuration of gradle took us a significant amount of time. It probably gets better with experience with the tools and more time spent using them, but for us it was hard, as it often was the first time.

Broken builds Don't you hate it when people commit code to the central repository that doesn't compile? We did that too. I think we treated the central repository as some sort of magical backup device. I don't think we do it anymore.

Splitting the program into parts This deserves a mention. When your code relies on other people's code to run, their code relies on other people's code to run and so on and so forth for some recursion depth, and their code hasn't been written yet, then, oops, writing and testing your code becomes a lot harder. What do you do? Do you wait for the other people to write their code first? Do you commit it to the repository without, ehm, compiling? (The answer probaby involves mock objects and junit, but we didn't really use that yet. Hopefully the next phase will enlighten us.)

Our program had a tiered architecture, and this especially applied to us. We didn't run it from start to finish until today, 25/01/2013, despite many parts working on their own before.

Licenses. We didn't quite know much about the details and the limitations of the different open source licenses, as well as their compatibility with closed source software like oracle's. It turns out the formal, legal language the software licenses are written in does not make fun bedtime reading. This hurt us, as mentioned below.

Things change Admittedly, we probably were a bit optimistic and started a bit late. However, it turns out planning perfectly is hard (yes, we were also bad at it). Here is an example.

We registered the program at ohloh.net. It's an open source analysis program, that analyzes things like lines of code, contributors and programming languages used in open source projects. Then it gives estimates of project maturity, cost, etc. It uses the COCOMO Model we were taught at the software engineering lecture. The verdict? Our program took 4 person-years to create.

Can you always rely on everything going according to schedule? What happens if unexpected events still happen? We even had a tight specification, no indecisive clients that might change their minds or lose interest (or even many potential clients whose intentions we would have had to guess indirectly through market surveys). Is the waterfall model partially to blame?

In any case, we had to deal with unexpected changes, like the use of mysql instead of oracle for the database, the writing of sql queries instead of a warehouse and other changes mentioned above.

What if only one person understands some code and something happens to them? Thankfully, this didn't happen to us but was a disaster waiting to happen. We were but a crisis away.

Crises What happens at unexpected negative events in a group project that is, well, more or less egalitarian and doesn't have a person in charge? What are possible reactions of team members? Do they help where needed, or do they react negatively? Thankfully most of us weren't like me and the program will be functioning this evening.

Communication Lastly, it turns out is really important. Better communication and understanding between us would have meant that we wouldn't have had to rewrite some things, spend less time arguing, explaining and reexplaining things and more time actually programming.