Emergent Architecture Design

Boudewijn van Groos[1], Chris Langhout[2], Jens Langerak[3], Paul van Wijk[4], and Louis Gosschalk[5]

[1]bvangroos , 4229843
[2]clanghout , 4281705
[3]jlangerak , 4317327
[4]pjvanwijk , 4285034
[5]lgosschalk , 4214528

June 19, 2015

Contents

1 Introduction

This document describes the design of the architecture and it is continuously updated to the current state of the software. It also states the design goals we want to achieve during the project. Furthermore the subsystem decomposition is laid out.

2 Design goals

2.1 Availability

We will have frequent meetings with the clients. During these meetings we want to be able to show a working prototype of the software in the current state. The feedback provided during the meetings can then be used for improvement. If there are extra features the client wants, we can decide if we will implement them.

2.2 Modularity

The program can be split into a number of modules. These modules should be implemented independently of the other modules. Interaction between modules is only possible by making use of defined interfaces. The system is also based on the model-view-controller (MVC) design pattern. See section 4.1 for further details on this pattern.

2.3 Continuous Integration

There should be always a working product. Each sprint should add one or more features.

2.4 Understandability

We must also make the software user-friendly. Even though the users of the product are already familiar with analysis software, our application should be easy to learn. It is essential that the user understands the usage of the scripting language in our application before he/she is able to use it for analysis.

2.5 Code quality

In the project we strive to have good code readability and the code must be easy to understand. This is also an important part of the static analysis. We handle the pull based system on Github. For each new feature that will be implemented the author of that feature must create a new branch and do all his work in that branch so that the master branch always consists of a working version of the software. If someone wants to commit he has to make a pull request first. His code will then be reviewed by at least 2 teammembers and eventually fixed before it can be merged with the working version in the master branch. In addition the code must be diagnosed using static analysis (see section 3)

3 Static analysis

We make use of static analysis tools that provide information about the quality of the software:

Corbertura We use this tool to find out how much test coverage we have in our system. Our goal is to keep the line coverage as well as the branch coverage at at least 75%. This means that not only cases where the program works well should be tested, but also the cases where the program should fail.

Checkstyle To improve code quality in terms of styling we have code conventions which the writer should follow before his code is accepted. This is done by the Checkstyle plugin in the IDE that makes sure these conventions are followed.

FindBugs This tool will help us find where our program may contain some faults. This also makes code reviewing for teammembers easier since the process is automated. If there are faults, the writer should fix them to make the code acceptable.

PMD The program can contain some programming mistakes such as unused imports or dead code. Since we do not want these mistakes in the code we use PMD to detect these mistakes using PMD's built-in rules.

It is up to the writer to fix the warnings produced by these tools, or provide a clear explanation whether the warning will not be fixed.

4 Software architecture views

4.1 Model View Controller

The software is based on the model-view-controller design pattern. The idea is that we want the analyis data in the columns (model) separated from the user interface (view). The interaction between these modules will be accomplished with the controller module. The controller is able to change the view and uses data that is obtained from the model. The user will be able to modify the model by providing operations to the controller which will then change the model.

4.2 Subsystem decomposition

As mentioned before our program consists of modules that work together. In figure 1 these modules are laid out. The model module consists of the data that was read from files and data that was outputted from performed analyses. This data can be used by the controllers to show it to the user. The user can specify an analyis in SUGAR and the controller will then update the model after the analysis is translated into operations.

Input The input module is able to read the data from provided files that are specified in an xml file. With that data a dataModel can be constructed.

Model The model module contains and manages the data that was read and/or analyzed by the user. This data will also be provided to the controller and analysis modules. The data is stored in tables.

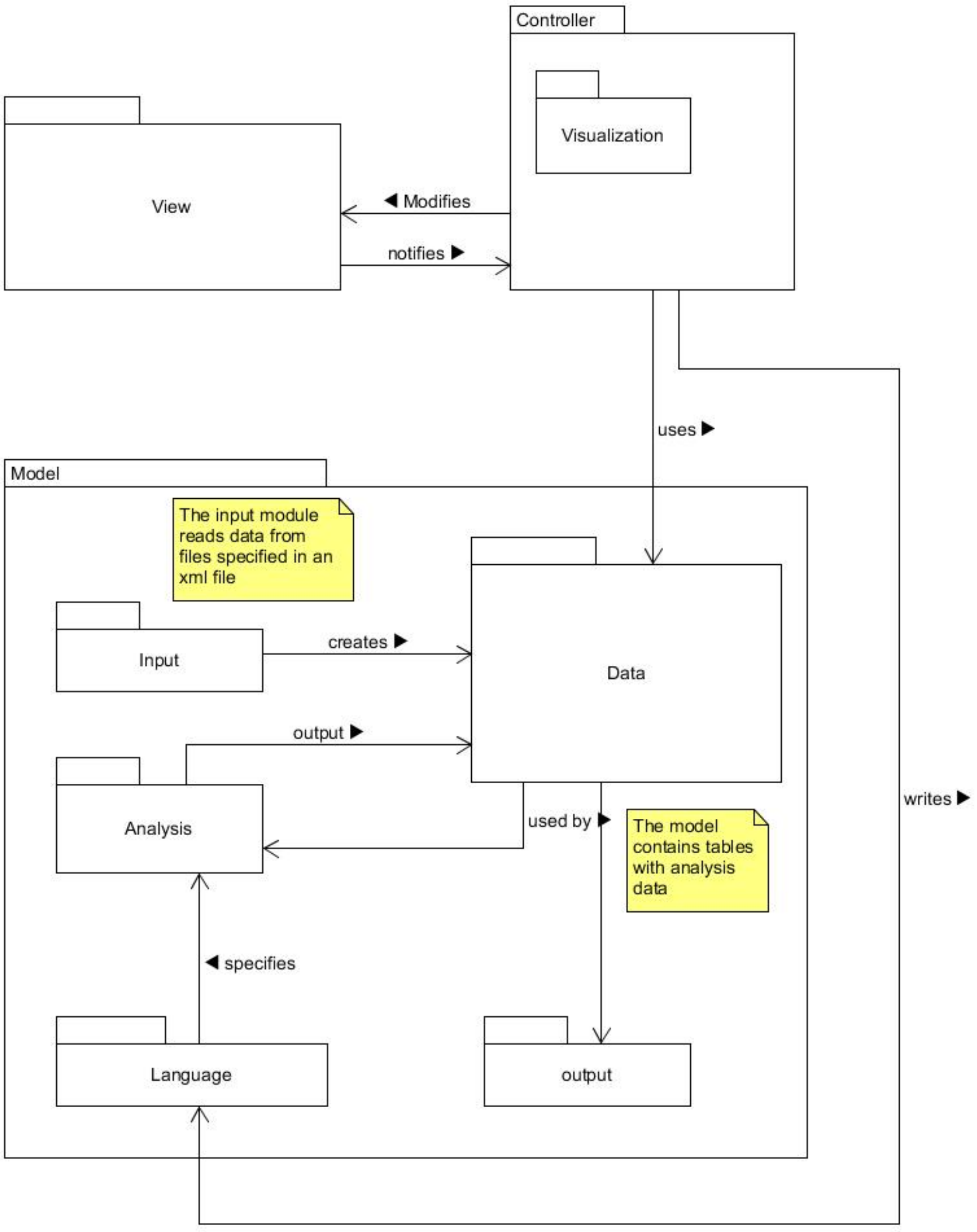

Figure 1: Modules of the system

Analysis The Analysis Module performs the analyses over the dataModel. It contains all the operations (the 8 C's) to make the exploratory sequential data analysis possible. The output of this module is the result of the analysis, this is a table that will be added to the model.

Controller The controller module is responsible for the program's front-end behaviour. It is the module to which the user has access. The controller uses the current state of the model to produce the according view for the user.

Visualization The Visualization module creates a certain visual representation of a dataModel. For example it can create a box-plot of the creatine levels. The visualization is kept up to date by the controller if the data changes.

View The view module is responsible for the appearance of the user interface. Determining by the controller, the view can change accordingly. The user may change the view after which the controller gets notified. The controller can then perform actions on the model.

Language The language module is responsible for parsing the analysis the user gives us into an executable chain of code. The parser is two fold: one is for the process-chain, the other is for the macros. The process chain is turned into various ProcessInfos and the macros are turned into MacroInfos, which can be parsed into various kinds of ValueNodes. The ValueNodes make up the tree in which the constraints, computations, codes and all other operations are contained.

4.3 State diagram

Figure 2 represents the state diagram which describes the flow of the program.

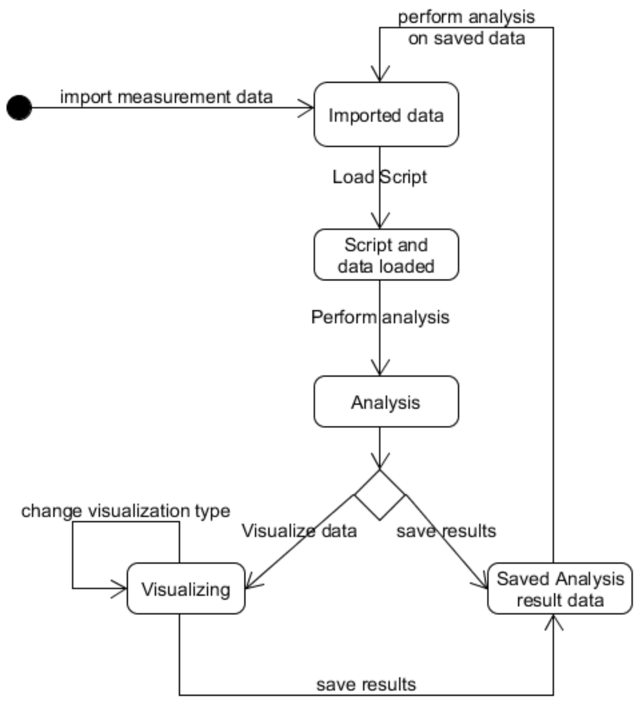

Figure 2: State diagram of the program

5 Software architecture

5.1 Programming languages and auxiliary programs

The software will mainly be written in Java since this was obligatory in the project. For the graphical user interface we make use of the JavaFX framework and is written using fxml. We use Eclipse and IntelliJ IDEA as IDE to write the Java code with. For continuous integration and version control we use Git and Github. Locally we use Maven to build the project. On GitHub we use Travis CI to verify the builds. We also use the JavaFX Scene Builder as a tool to design parts of the GUI because this provides a better preview of what the interface will look like while creating it.

5.2 Data organization

The data that we read from the file is stored in datatables containing columns of specific types. These datatables together form the model on which we can perform the analyses. The values are stored as datavalues inside columns having the same type as the datavalue. These datavalues are divided in subclasses for strings, integers, floats, dates and booleans.

Date values are a bit more complex than, for example an integer. Date values can vary in being a timestamp, period of time, just a date or just a time value. The section for dates of the datavalue structure has been implemented to have multiple types of values for these varying types of dates.

To refer to data inside methods, we use identifiers. These identifiers can point to a column or table by finding the corresponding data for its given string which is the name of the searched data. The software also has a describer implemented, which can be used to refer to proceses like a computation or a single value in a column of a row.

In week 4.8 we found an issue that constructing a datamodel from a large excel file takes up a lot of memory. We thought this could have been caused by the fact that we used hashmaps to store columns in the model. We tried using linkedlists instead because these take up less memory but there was no significant difference. Because of the access speed of hashmaps we continued using hashmaps for the columns.

5.3 Used libraries

JavaFX Creating the graphical user interface. We mainly chose JavaFX because it is already included in Oracle's Java 8 and provides some improvements on Oracle's previous frameworks. One of these is the use of fxml to construct the static part (view) of the GUI. The advantage of fxml is that this static part is separated from the controller part so we do not have cluttered up code and long methods just to build up a window. Some of us already gained some experience with it during the Software Engineering Methods course.

Mockito Mocking objects to test behavior. We did not specifically choose for this framework but we have used it before in previous courses and it did not seem like a big necessity to change to a different mocking framework as this will only require more effort to learn it.

Apache POI Reading data analysis files stored in the Microsoft Excel format. This is one of the main excel parsers and easy to use. We chose this library because it applies very good solution for reading the data that was provided to us in excel formats. Another reason is that it is developed by the Apache Software Foundation which also created Maven so it was easy to include in the build.

Java SAX Reading and writing the xml file that specifies the datafiles. This choice is made because it is easy to use and well documented. The library is also included in the JDK so it did not require any additional installation effort.

Parboiled In the parser department we had lots of choice. We tried to find a framework which was both simple to use and still powerful enough for our use. This lead us to Parboiled. The beauty of Parboiled is the fact that you can write your rules in plain Java. This means you don't get an extra build process. The disadvantage is that Parboiled does some magical stuff which our analysis tools don't fully understand, so you need to be aware of it. All in all we're really satisfied with the strength of Parboiled and feel that we've made the right choice.

JFreeChart For creating the boxplot visualization we made use of the JFreeChart framework after we found that JavaFX (which we use to create barchart visualizations) did not support creating boxplots.

ControlsFX For making dialog windows to notify the user that something happened. This library offers various standard popups that the average computer user is familiar with. The main element we wanted to use was the Alert dialog of JavaFX but it turned out this class is not supported in the older version of javafx and the newer version was is not supported by Travis.

Apache Commons CSV To write the tables to a csv file. We chose for this library because it is also developed by Apache which also made it easy to add it to the Maven dependencies.

6 Testing

6.1 Unit testing

To make sure that all the code works as it should, the author must always write unit tests for the code he has written. We make use of the JUnit framework to write the tests. To verify behaviour of methods we use the Mockito framework to stub certain object needed by the method.

6.2 Code Coverage

Unit Testing is the method, code coverage is the goal. We strive to maintain an 80% line coverage, which means there is room for exceptions to testing but there is a high requirement of code testing set for the programmers. This is an important measure to prove to the customer that our code is performing as expected and resulting correct answers to the opposed problems.

7 Glossarium

Analysis An analysis that can be done on measuring data to come to an answer to a specific question.

Git & GitHub Git is the tool to control the versions in a repository. GitHub is the web-tool to make the versions available for group use.

GUI Graphical User Interface.

IDE Integrated Development Environment.

JDK Java development kit.

MVC The Model-View-Controller design pattern.

SUGAR The scripting language used to create analyses.

Pull Request Are made on GitHub to get your commits to the code verified and merged with the master branch.