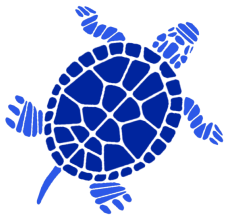
Architecture Design [Draft]



Delft University of Technology
Faculty of Electrical Engineering, Mathematics and Computer Science.
Mekelweg 4,
Delft

Team Members:

Name:	StudentID:	E-mail:
Boning Gong	4367308	boninggong@yahoo.com
Clinton Cao	4349024	C.S.Cao@student.tudelft.nl
Michiel Doesburg	4343875	M.S.Doesburg@student.tudelft.nl
Sunwei Wang	4345967	S.Wang-11@student.tudelft.nl
Tim Buckers	4369459	TimBuckers@gmail.com

SE TA:
Bastiaan Reijm

Course:
Context Project (TI - 2806)

TU Delft

Table of contents

1. Introduction

This document provides a view and explanation of the architecture of the system, that is going to be built during the context project (Tools for Software Engineering). It provides a high level overview of the components and interaction of the system. For this project we have to develop a software tool that is able to analyze the output files of the ASATs e.g. CheckStyle, and based on all the information that I gathered from the analysis, the tool should produce a visualization.

The following topics will be discussed in the rest of this document: We will first discuss about our design goals. Then we will discuss about the software architecture views.

1.1 Design goals

The following design goals will be maintained throughout the project:

- Modularity: our project lends itself extremely well to a modular oriented design. We want to split the code analysis and visualization part into two different engines; the analyser and visualizer respectively. These components will run independently of each other, and as such that each component will be easy to replace or adjust without needing to touch the other component. Modularity will also be maintained internally in these two components; adding another ASAT for example should be easy to do. Each ASAT should be handled independently by the analyser, and in the end producing one unified output. If another ASAT is added, it should simply latch on to the queue and append its output to the unified output in some way. The same holds for the visualizer. Adding a new visualization technique will not interfere with any visualization technique that is already present.
- Code quality: the aspect of code quality is an obvious necessity. Code will be structured in a unified style. Many good coding practices will be applied, like making classes carry just the right amount of responsibility (not too long, not too short), methods shouldn't be too long, using easy to understand variable names, etc… Checkstyle, PMD and code reviews will, among other things, be used to guarantee code quality.

2. Software architecture views

2.1 Subsystem decomposition

As said before, the system consists of two main components; the analyser and the visualizer. The user provides the analyser with the source folder of their project through the use of the *GUI*. The analyser mostly consists of parsers which parse the *.xml* files provided by Maven to generate a list of defects. The visualizer will use the output of the analyser to create a visualization.

The figure below gives a high-level overview of the whole system:

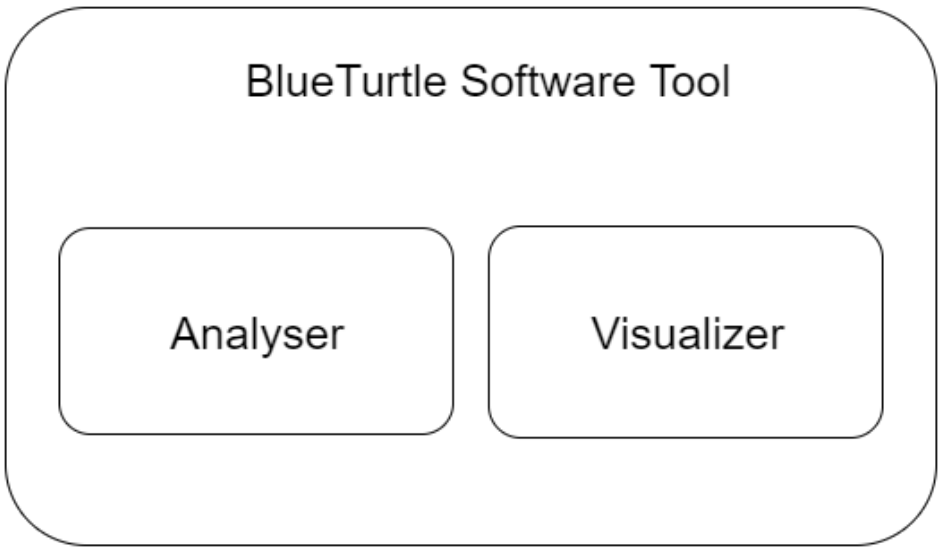


Figure 1. High-level overview of the system.

2.1.1 Further decomposition of the Analyser.

As mentioned above, the analyser consists mostly of parsers. The parsers are all put into the same package: *BlueTurtle.parsers* . Besides the parsers that are needed to parse the output files of the ASATs (produced by Maven), there is another parser, namely the *GDCParser*. This parser is needed for parsing the *HTML* file, that contains the information of the the General Defect Classifications(*GDC*)[1] . The information of the *GDC* is needed for classifying the defects.
Besides the parsers package, there are also other packages:

- *BlueTurtle.finders*: This package contains classes that is used to find information.
- *BlueTurtle.computer* : This package contains a single class that is used for computing information.
- *BlueTurtle.groupers* : This package contains a single class that is used for grouping the defects together e.g. group the defects by the package that they belong to.

- *BlueTurtle.gui* : This package contains classes that are used creating the *GUI* of the analyser.
- *BlueTurtle.Summarizers* : This package contains classes that are used for summarizing the results are returned from the parsers e.g. summarise all the warnings for a specific class.
- *BlueTurtle.Warnings* This package contains classes that are used for representing a defect/warning of an ASAT e.g. CheckStyle.
- *BlueTurtle.TSE* : This package contains classes that are used for integrating all the components/classes together.
- *BlueTurtle.writers* : This package contains a single class that is used for writing the output of the analyser to a file, that the visualizer will be using, to produce the visualization.

The figure below gives a high-level overview of the composition of the analyser.

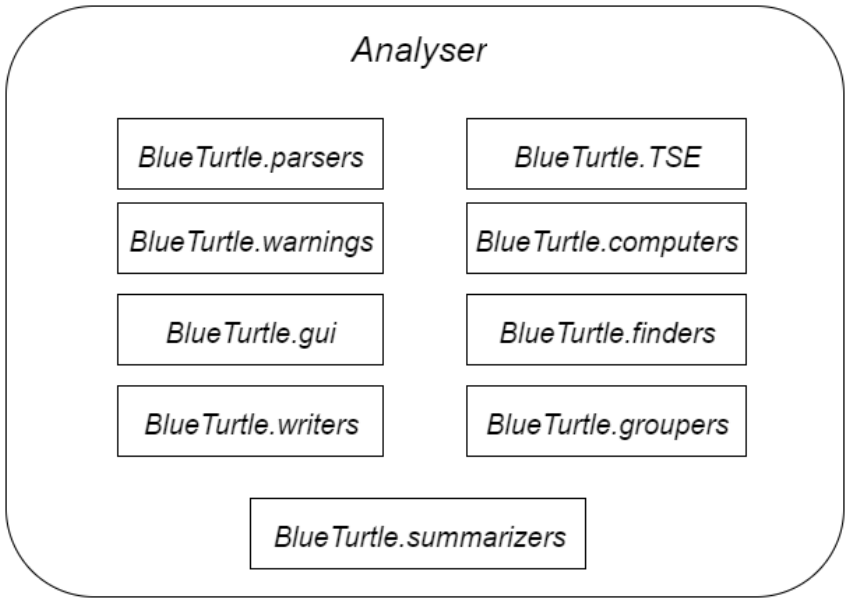


Figure 2. High-level overview of the composition of the analyser

2.1.2 Further decomposition of the Visualizer

The visualizer consists of different types of files. It mainly consists of JavaScript, but also has some *HTML* and *CSS*. *Main.html* will be called by the analyser to create the visualization. When the *Main.html* is called it will run *mainDriver.j*s and this file will perform multiple tasks.

Here we will give a description of the most important files:

- *Main.html* : This is the file that will be called to make a visualization in your browser. It contains all visual elements from the checkboxes and toggles to the labels and treemap.

- *mainDriver.js* : This is the main JavaScript file that will control the interaction in the *GUI*.
- *inputData.js* : This file is the output from the analyser. It is a JavaScript file but it really just contains one variable with the output JSON declared to it.
- *jsonBuilder.js* : This file will parse the variable in *inputData.js* and can return objects with different structure. These will be used by the treemap.
- *treeMapBuilder.js* : This is the file that will create the treemap visualization. It uses the *jsonBuilder.js* to get an object that it can use. This file will also handle all clicks in the treemap.
- *styleSheet.css :* This file is self explanatory. It gives the styling to the whole *GUI*.
- Other small JS files : This map is called *helperJSFiles* and consists of multiple JavaScript files. It contains:
    - *backgroundGradient.js* : Sets the different gradients for the treemap.
    - *gradientCalculator.js* : Calculates the boundaries and surface within each node in the treemap for the different number of warnings.
    - *handleClicks.js* : File with functions that handle all toggles and checkbox clicks from the user.
    - *handleGDCHovers.js* : File that handles all functions needed to show the descriptions of each category when you hover over it.
- *Libraries* : The libraries that we used for our product are:
    - *Bootstrap*
    - *CodeMirror*
    - *D3*
    - *jQuery*

The figure below gives a high-level overview of the composition of the visualizer:

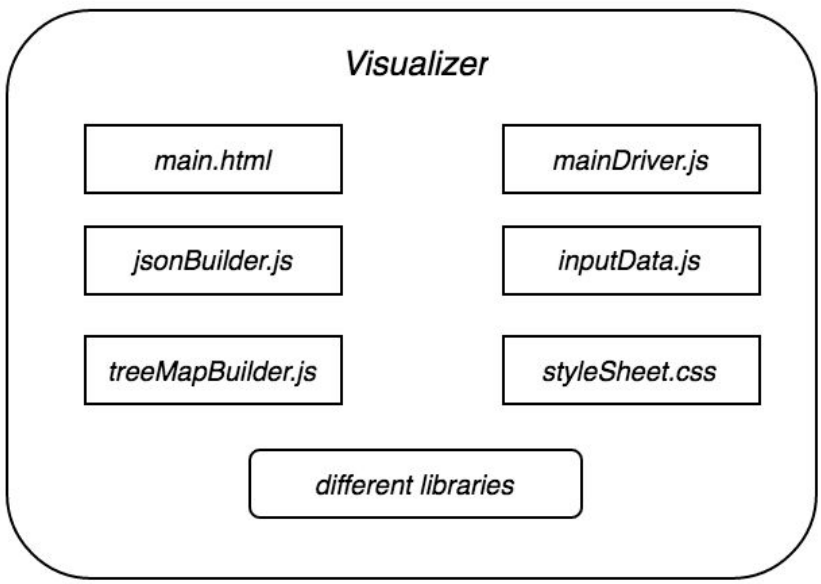


Figure 3. High-level overview of the composition of the visualizer

2.1.3 From Analyser to Visualizer.

After the analyser has finished its job, the visualizer will produce a visualization. The defects are unified using the *GDC*. The list of general defects and their relevant information are passed to the visualizer as JSON output. The figure below illustrates an example of a defect:


```
{
  "line": 1,
  "message": "Name Brain must match pattern ^[a-z]+(\\.[a-zA-Z_][a-zA-Z0-9_]*)*$.",
  "classification": "Naming Conventions",
  "fileName": "Speler.java",
  "type": "PMD",
  "filePath": "C:\\users\\bob\\Downloads\\src\\Brain\\Speler.java",
  "ruleName": "PackageName"
},
```


Figure 4. Sample Defect in JSON format.

The list of defects and project metadata is passed to the visualizer which will handle the rendering and visualization of the project. The process is illustrated by the figure below:

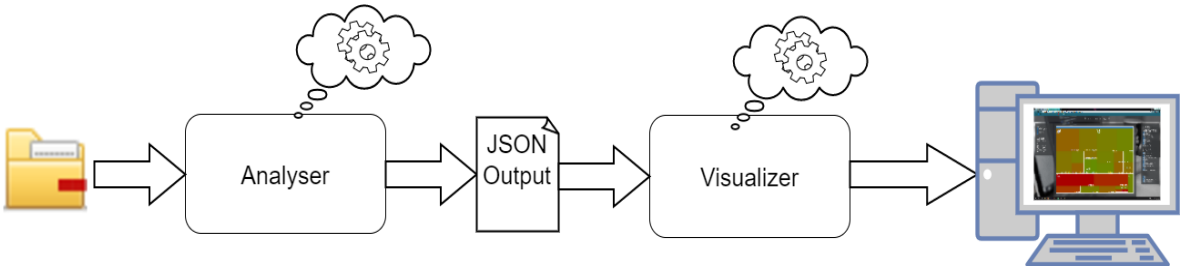


Figure 5. The process from input to visualization.

The same process is illustrated below in more detail with a sequence diagram[1]:

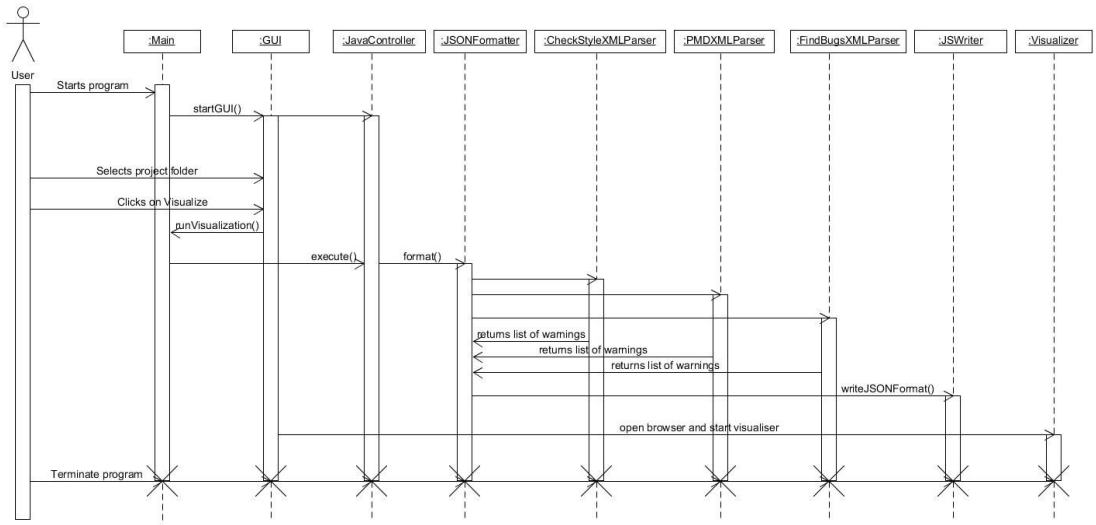


Figure 6. Sequence diagram.

[1] For a bigger diagram: link……..

2.2 Persistent data management

The system will have to deal with all kinds of output data from ASATs like *HTML*, *XML* among others. This data will be unified into one common type. We have chosen this common type to be xml because many ASATs (CheckStyle, PMD, FindBugs) already provide *XML* output.

We have decided to not implement a database to store the output files produced by the analyser for the following reasons: First of all, it might take some effort to set up a database and we don't want to waste too a lot of time setting up a database. Second of all, we think that it should be possible to run our tool without the access to the internet, so implementing and setting up a database might not worth the time and trouble.

The output files that are produced by the analyser are first converted into *JSON* format with a library and then it is written to a JavaScript file. This seems very strange, but the reason why we are doing is because *Google Chrome* and *Microsoft Edge* does not allow JavaScript to access files that are stored on the local machine. We could simply ask the user to disable the security of their browser, we don't want the users to run into security problems later on (as some people might forget to turn the security back on).

2.3 Choice of programming language

The analyser will be written in Java. Mostly because the team has the most experience in Java and are most familiar with its frameworks for basic user interfaces.

The visualizer will be written in JavaScript. JavaScript was chosen for the visualizer because it has very good frameworks to build upon. The treemap visualization framework for example is a solid groundwork to build our visualizer upon. Java does not have these kinds of visualization frameworks.

3. Glossary

ASAT
Automated Static Analysis Tool. A program that takes source code as input and produces as output some measurement of quality of code. These might be metrics like average lines per method, or these might be warnings of possible bugs.

Checkstyle
A static analysis tool mainly used for ensuring a good and homogenous coding style between programmers.

PMD
A static analysis tool which mainly checks for redundant, wrong or suboptimal code. It for example highlights code where bugs may possibly exist or code which will never be reached.

HTML
HyperText Markup Language. It is mainly used as a markup language for webpages.

XML
Extensible Markup Language. It is a markup language for encoding documents to make them both easily readable by humans and machines.

Java
A general purpose programming language that is designed to be easily runnable on different machines through the use of Java virtual machines.

JavaScript
A programming language generally used for scripting and enabling interactivity on web pages and web applications. Features many different kinds of frameworks for these purposes.

JSON
JavaScript Object Notation. Specific format to encode an object in. It is used to transfer data between different languages.

Google Chrome
An internet browser developed by *Google.*

Microsoft Edge
An internet browser developed by *Microsoft* (Maybe an improved version of *Internet Explorer*?)

4. References

[1] General Defect Classifications
http://www.st.ewi.tudelft.nl/~zaidman/publications/bellerSANER2016.pdf