Final Report[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)

Table of contents

Acknowledgements

We want to say thanks to our software engineering teaching assistant (SE TA), Bastiaan Reijm. Thanks for all the feedback for the sprints, and for the suggestions and help that you have given us when we encountered a problem.

- Team BlueTurtle

1. Introduction

The problem to be solved consisted of a software engineering tool to aid software developers and researchers. There exist many different automated static analysis tools which all have their own names for many different kinds of warnings. The naming of the warnings between two different ASATs is usually different.

Grouping these warnings together in some coherent way and presenting this in a clear visual way to a software developer, is a good way to help give this software developer insight into their project and where the tools report which kinds of warnings. It can also help a software developer with the following: getting an idea on how they should spend their time fixing the warnings that were generated by the ASATs. Some kinds of warnings are more important for some software developers than others and therefore these kinds of warnings have a higher priority for a fix.

On the other hand the tool should help researchers of ASATs see which kinds of warnings the ASATs report and where these might overlap in a project.

The end user's requirements of our project consist of the following:

- The user should be able to interact with a visualization of their project.
- The user should be able to see how many warnings exist in each component in their project.
- The users should be able to see which ASATs these warnings originate from.
- The user should be able to see which kind of warnings there are.
- The user should be able to see in which packages and classes the warnings exist.

The following topics will be discussed in the rest of this document: In chapter 2, an overview will be given for the developed and implemented software product. In chapter 3, a reflection will be given on the product and process from a software engineering perspective. In chapter 4, a description will be given for the developed functionalities. In chapter 5, a description will be given for HCI module that was realised for the user interaction with the developed solution. In chapter 6, an evaluation will be given for the functional modules and the product in its entirety, including the failure analysis. And finally in the last chapter 7, an outlook will be given.

2. Overview of the developed and implemented software product.

This section gives an overview of the developed and implemented software product. The product consists of two main components; the analyser and the visualiser. The users provide the analyser with the folder of their project through the use of the *GUI*. The analyser gathers all the necessary information from the output files of the ASATs that are generated by Maven, and produces an output. The visualiser will then use the output of the analyser to create a visualisation. Figure 1 provides an overview.

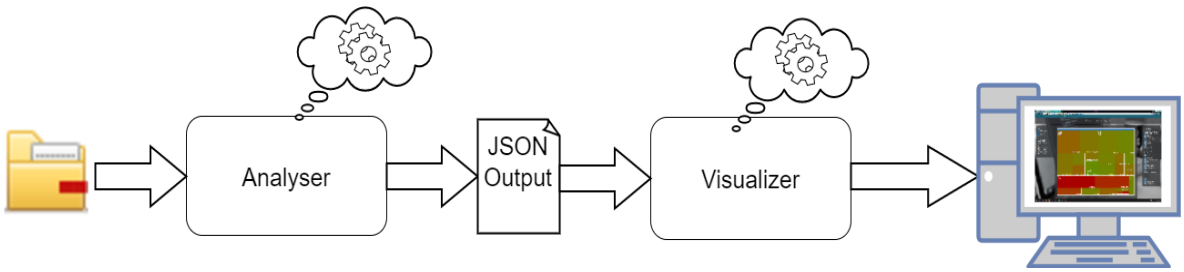


Figure 1. The process from input to visualization.

The same process described above is illustrated in more details in a sequence diagram, which can be found in Appendix A, Figure 2.

In the following sections, a more detailed description will be given for the analyser and the visualiser.

2.1 Analyser

The analyser consists mostly of different parsers. They are in the same package, and are used to parse the output files of the different ASATs (produced by Maven), such as CheckStyle, PMD etc. And there is another important parser, the *GDCParser* which is needed for parsing the *HTML* file, that contains the information of the the General Defect Classifications(*GDC*) (Bellet et al, 2015) . The information of the *GDC* is needed for classifying the defects into different categories, such as functional defects, maintainability defects, etc. The rest of the analyser consists of the following:

- Classes which are used for summarising the warnings for a specific class or package.
- Classes which are used for finding all the necessary information of the project that was given as input.
- Classes which represent warnings of the ASATs
- Classes which are used for grouping the warnings by their corresponding packages or classes.
- Class which is used for generating the output for the visualizer.

For high-level overview of the composition of the analyser, see Appendix A, Figure 3.

2.2 Visualiser

The visualiser uses the export of all the warnings from the analyser, it then gives an overview of the warnings and their locations in the project. As the information regarding the GDC is parsed in the analyser, they can be filtered by different categories. They can also be filtered by different ASATs. So the user can see the warnings generated by different ASATs in their project and can also compare them.

It is also possible for the user to view the source code and the exact lines where warnings exist. This helps the user to locate where the problems are in their own IDEs. For high-level overview of the composition of the visualiser, see Appendix A, Figure 4.

3. Reflection on the product and process from a software engineering perspective.

The product has been written in Java and Javascript. The parser and analyser are written in Java. The visualisation part running on the browser has been written in JavaScript, HTML and CSS.

The team has adopted an agile work style. The product was developed in 8 iterations, each of which lasting a week. Each iteration was planned according to two lightweight SCRUM documents. The first being a sprint plan created at the start of each sprint outlining the tasks to be performed in that sprint, along with the responsible team member and estimated time. The second being a sprint reflection at the end of each sprint, documenting which features have been implemented and which tasks are still open. Each sprint reflection was taken into account for the next sprint plan. There were weekly meeting with the client were the team gave a demo of the latest features and received feedback on what to work on for the next iteration.

The team used Github for version control. The team used an open repository. The team followed a pull-based development model. New features or bug fixes were implemented on separate branches which were merged into the master branch (or rejected) through pull requests. Each major pull request was reviewed by at least two other team members. If the code in a pull request failed to build, the pull request was rejected by default.

The team worked with continuous integration using Travis-CI in conjunction with Maven as the automated building tool. The automated static analysis tools CheckStyle, FindBugs and PMD were used to ensure high code quality. jUnit test framework was used to test the code. The team started out with Cobertura as the code coverage utility tool. Later the team switched to using JaCoCo for checking code coverage as Cobertura has problems with Java 1.8 and lambdas.

Every week the team received feedback on the code structure and how this could be improved from Bastiaan. The team carefully considered each suggestion and evaluated how it might improve the code. An example of such feedback which the team ended up implementing is turning the *PackageNameFinder* class into a singleton class.

There were two major problems the team encountered throughout the project. The first problem was trying to build a tool which ran on non-Maven projects. This meant letting the tool run PMD, FindBugs and CheckStyle separately through the command-line. There were several problems with having the tool run the ASATs:

- The ASATs would somehow need to be included as runnables in the tool. Directly including the software into the project bloated the size of the tool considerably (to about half a gigabyte).
- The version of each ASAT would become outdated after a while. Adapting the tool to work with any version the user wants would introduce maintainability problems where ASATs might change their command-line interface in the future.

- ASATs have many options and configurations. Letting the user configure how they want the tool to run the ASATs would require setting up a fairly complicated settings feature. This would end up taking the team a considerable amount of time.

Using Maven solved all these problems and it also saved the team a lot of development time and effort. Considering the development time and effort saved and that Maven is also widely used, the team chose to put this restriction on the projects that the tool supports.

The second problem consisted of a miscommunication between the team and the client. The client wanted a tool to visually compare different ASATs on a project. However, after the first meeting the team was under the impression the tool was meant to visualise code and warnings, not necessarily which ASAT the warnings originated from. The weekly client meetings really shined when it came to dealing with this problem. From receiving the feedback, the team was quickly able to adjust their vision of the product and what the client wanted. In the end, this problem only meant some misdirection at the start of the project and was solved reasonably quickly.

4. Description of the developed functionalities

The tool consists of two parts. The first part is a simple user-interface which starts as an executable program on the user's computer. In this user-interface the user can select their project folder and click on the "*Visualize*" button. This opens a page on the user's web browser with the visualisation of their project, which is the second part of the developed product. The visualisation consists of three parts. A left-hand menu column, a block with the actual interactable visualisation, and a right hand menu column.

The left-hand menu column gives the user the several options:

- The user can enable or disable each ASAT present in the visualisation.

- The user can select which of the three different kinds of visualisation modes they want to see. There are three modes:
    - A view which shows the composition of each class based on which ASAT the warnings are from.
    - A view which shows the composition of each class based on which main defect category the warnings belong to, which are Functionality, Maintainability and Other.
    - The default view which assigns each package or class a gradient color from red to green based on how many warnings exist in that particular package or class.

- The user can toggle the absolute/relative view of the project. The absolute view bases the intensity of a class's (or package's) gradient color on an absolute scale, where one warning per line of code is pure red, and zero warnings per defect line is pure green. The relative view maps pure red to the highest ratio of warnings to lines of code which exist in the project, and pure green to the lowest ratio.

The right-hand menu column allows the user to enable or disable specific kinds of warnings. These warnings are grouped into three main categories: *functional*, *maintainability* and *other*. Within for example *functional defects* exist subcategories like *logic* or *concurrency*.

The middle visualisation block is the main part of the program. It is a treemap of the user's project, where every block symbolises a package or class. The highest level is a view of the packages which make up the project. The user can click on a package and the visualisation zooms in to show the classes which make up the package. The user can click on a class and the visualisation changes to the lowest level view, a view of the code of that class. The code is highlighted in places where warnings exist. The user can at any time move up a level by clicking the back button, allowing the user to explore his project as they see fit. Hovering the mouse over a class or package will also display the amount of warnings per ASAT in that class or package.

5. Development of the HCI module.

In order to see whether the tool that the team has developed is understandable and intuitive, testing is required. The team has therefore come up with a scenario which was tested. The following scenario describes a typical use case:

Bob is a software developer and he works for a software company. He is currently a part of the developer team. The team got an assignment to develop a program for a client. The team started directly after talking and getting the requirement from the client; they set up the repository on GitHub, set up Travis-CI and set up Maven for the project. They also decided to use ASATs (Automated Static Analysis Tools) for their project, so that they can see if there are warnings in their code.

After a few weeks of development, in the weekly meeting with the team, Bob has been assigned with the task to check all the warnings that were generated by the ASATs, and he needs to fix them. To Bob's surprise the ASATs report a list of over a thousand warnings. Bob has no idea where he should start. Bob decided to check some of the warnings from each of the ASATs, and he noticed that some of the ASATs gave the same kinds of warnings, but with a different name for the warning. Bob also noticed that he couldn't really see how the warnings are distributed across the whole system that they are developing. Bob decided to first fix the warnings that really need to fixed, for example warnings for coding style does not really need to be fixed very quickly. But Bob thinks that going through the whole list of warning one by one will take quite some time. He also think that he will lose some significant amount of time that he needs to fix the relevant warnings.
Bob was asking himself: Is there no software tool available where I can get information on the following things:

- How many warnings there are for each of the ASATs
- An overview of how the warnings are distributed across the system.
- An overview that shows if there are overlap in the warnings that are generated by the ASATs. And if they are, can they be grouped to the same category.

The team has decided to evaluate the usability of the developed tool. Ten second year computer science students were sat down in front of a computer with the tool, a list of questions, and a short explanation of the purpose of the tool. The testers were allowed to interact with the tool while answering the questions. Below is the list of questions:

1. Which ASATs produced most warnings in the entire project?
2. Which package has the most warnings?
3. How many warnings are there for the category Functional Defects?
4. How many warnings are there for PMD in the package Brain?
5. What's the definition of functional defects?
6. Which category has the most warnings in package junitTest?
7. How many warnings in the project are about Code Structure?
8. Which ASATs generate most warnings in LoadPlayersController class in default package?
9. When a package is red in the normal color scale. This means that the package...?
10. Is there anything unclear to you? Feedback and suggestions are also welcome.

For the in-depth results of the usability evaluation, see appendix A. Each question was answered correctly by at least 70% of respondents.

Below are some suggestions which, among others, ended up getting included in the project:

- *"When turning toggles on and off, everything seems to hang. Which is a bit annoying. Maybe a loading bar or something would be nice."* Having the tool feel responsive is very important for a good user experience. Because of this suggestion the team increased the responsiveness of the visualisation to where toggling buttons on and off is practically instant.
- *"I think displaying the statistics next to the cursor is a bit confusing. I'd rather it be in a separate static box next to the visualization."* The statistics were moved from being next to the cursor to a box on the left-hand side of the visualisation.
- "*The back button on the top looks like you can go back to a specific folder instead of the previous folder."* Initially the tool had a back button which allowed the user to go one level back up in his project visualisation. On this back button, the path to the file that the user was currently looking at, was displayed. This confused some respondents. They thought each component of the path itself was clickable to instantly go back to any level desired. The tool was changed to have this functionality which indeed is much more intuitive.

Though feedback were given for things that were not very intuitive to the user, it seems that most of the testers (potential users) understood the goal of the tool.

6. Evaluation of the functional modules, the product in its entirety, and the failure analysis.

6.1 Product

The team wrote a requirements document in the first sprint. The MoSCoW method (Clegg et al, 2014) was used to write the requirements document. This requirement document was then send to the client, so that the document can be reviewed.

At every sprint the team creates a prototype of the system with some of the requested features of the client. At the weekly meeting with the client, the team will give a demo of the prototype and the client will be able to give feedback on the prototype. Based on the feedback of the client, the team can derive new features/requirements. The team will then go through the feedback of the client and team make an ordered list of new features to implement (first one on the list has the highest priority). A new prototype will then be developed for the next meeting and the same process repeats again.

Besides getting feedback from the client, the team also used the feedback that was gathered from the usability evaluation (as mentioned in chapter 5), in order to get more feedback on the tool. The team then used the feedback to develop a new prototype that the team will use to discuss with the client.

The developed tool has all the must-have and should-have features from the functional requirements. The team decided to not implement any of the could-haves, because these did not have a high priority. Also all the non-functional requirements have been met.

6.2 Failure analysis

As explained in chapter 3, the original plan was to run the ASATs via the commandline. Due to the issues that the team has ran into, the restriction to Maven projects was added. This was discussed with our client and an approval was given for this decision.

It was pointed out by the SE TA, that it might be convenient if the analyser can run Maven for the user instead of letting the user run Maven by themselves. This is because the analyser needs the output files of the ASATs, that are generated by Maven. Some research was done for this issue, and the final conclusion of this research was to leave it to the user to run Maven. For more information about the research, please read this document.

7. Outlook.

There are four main areas where the team feels that the tool can be developed further.

First of all, support for multiple languages: There are several aspects where the product can be improved. The first option is support for multiple languages. Early on in the project, this was regarded as a could-have feature; a feature that, if the team had the time, could be implemented as an extra. The team soon realised building a functional tool for just one language was already a big enough task. The tool is easily adaptable to support more languages thanks to the independence between the visualiser component and analyser component. Support for e.g. Javascript would require a component which runs some Javascript ASATs (such as JSL or JSHint) on a code base and passes the list of general defects to the visualiser. The visualizer's main functionality remains the same, and would only need some small adjustments to work with the project structure of the new language.

This is only true for Javascript however as the GDC has already been defined for some Javascript ASATs. For any other language, the GDC would have to be expanded first.

Second of all, support for more ASATs: This tool can be expanded to support more ASATs for Java. However, since the GDC does not include any Java ASATs which aren't already supported, it would once again have to be expanded first.

Third of all, support for non-Maven projects: Currently the tool only supports maven projects. It relies on maven to generate the output files for CheckStyle, FindBugs and PMD. If these files are not found, the visualizer will display the project, but with no warnings. It is possible in theory for the user to run the ASATs themselves, write the output to a file and put the files where Maven would put them. This is of course very tedious however. The tool could be expanded to not rely on Maven for generating the ASAT output. In fact, this is what the team originally set out to accomplish. After a couple of weeks however it became apparent this was too much for a 9 week timeframe and the team chose to use Maven to save a lot of work.

And finally, history based project evaluation: Another feature which was on the team's could-have list for a while was a history based project evaluation feature. Eventually the team chose to focus on other functionality and features. A history based project evaluation would remember each evaluation and be able to compare them over time. This would enable the user to see how their project has evolved in for example which warnings went up/down and where.

8. References

Beller M & Bholanath R. & McIntosh S. & Zaidman A. *Analyzing the State of Static Analysis: A Large-Scale Evaluation in Open Source Software.* 2016. *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), 470-481. doi: 10.1109/SANER.2016.105*

Clegg D. & Barker R. (2004-11-09). *Case Method Fast-Track: A RAD Approach*. Addison-Wesley.

Appendix A: Graphs for overviews

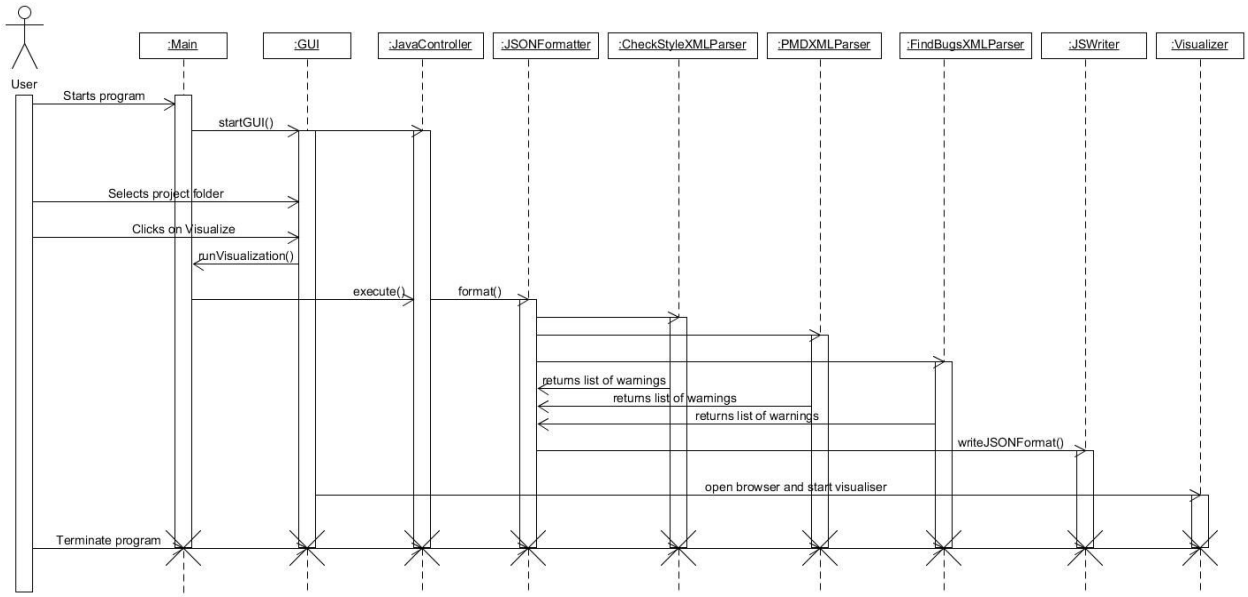


Figure 2. Sequence diagram.

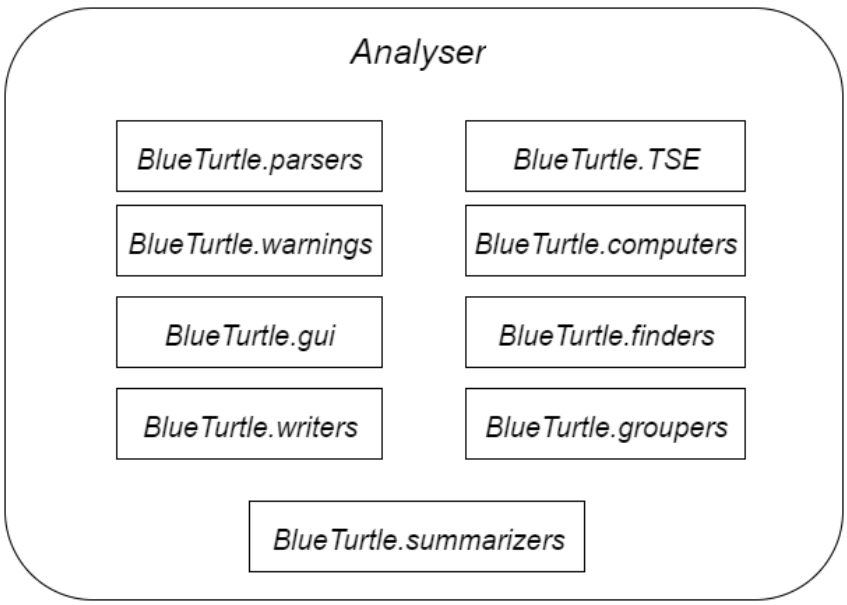


Figure 3. High-level overview of the composition of the analyser

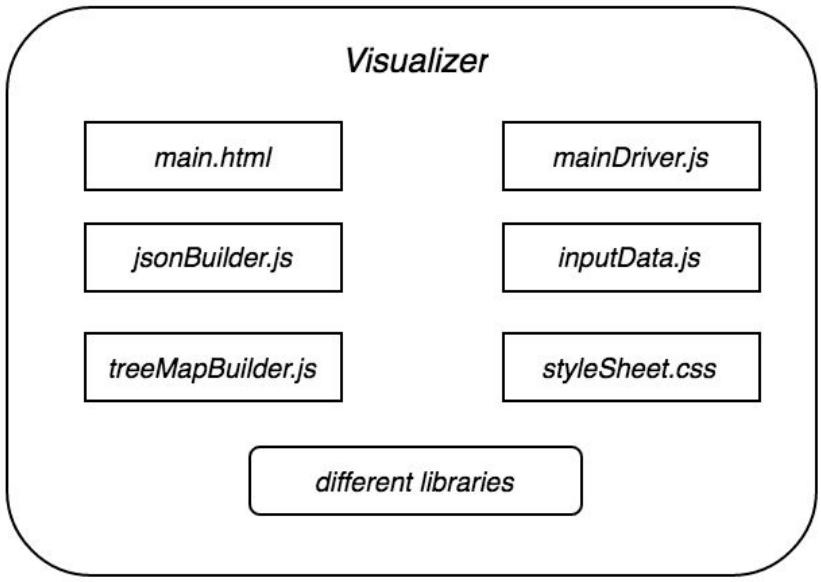
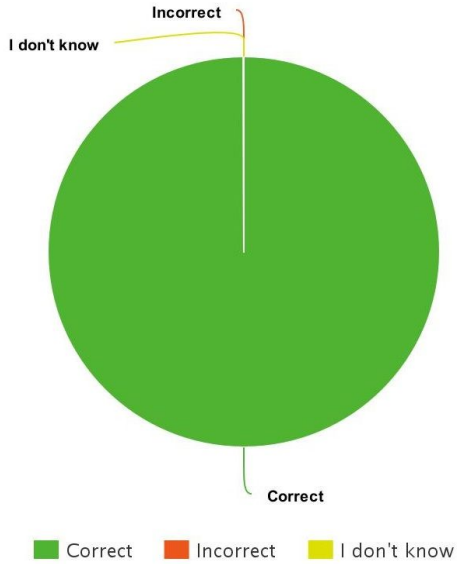


Figure 4. High-level overview of the composition of the visualizer

Appendix B: Results of the usability evaluation.

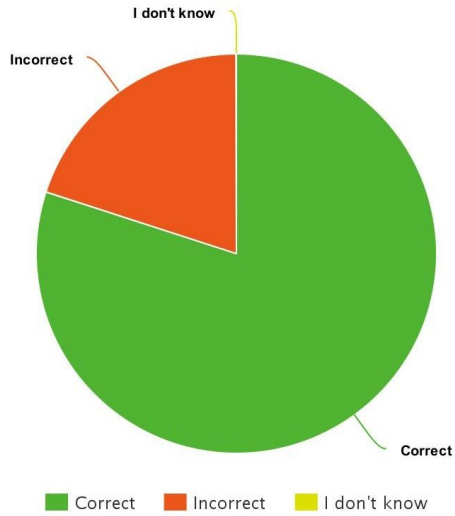
1. Which ASATs produced most warnings in the entire project?



Possible answers were (bolded is the right answer):

- **Checkstyle**
- PMD
- FindBugs
- I don't know

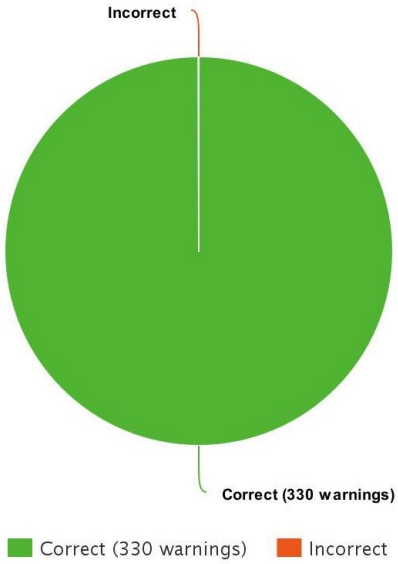
2. Which package has the most warnings?



Possible answers were:

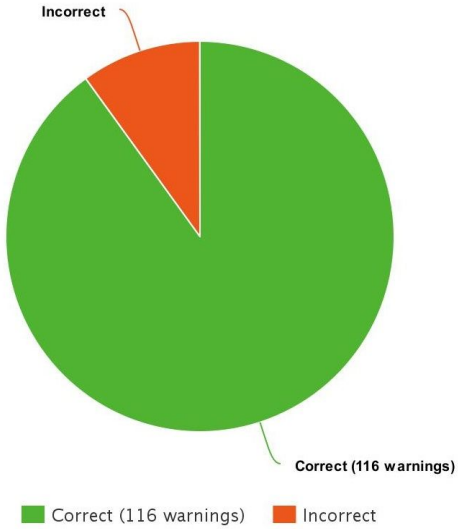
- **jUnitTests**
- Default
- Brain
- Project

3. How many warnings are there for the category Functional Defects?



Open question.

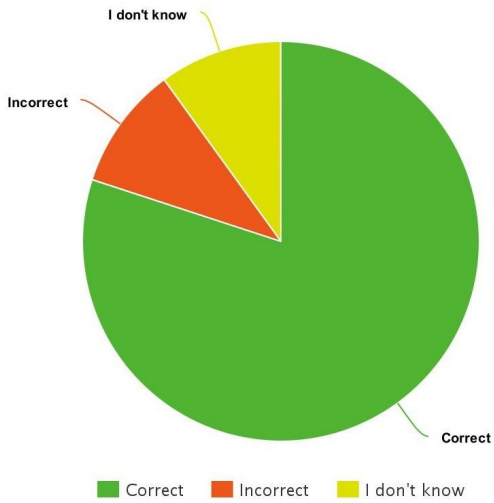
4. How many warnings are there for PMD in the package Brain?



Open question.

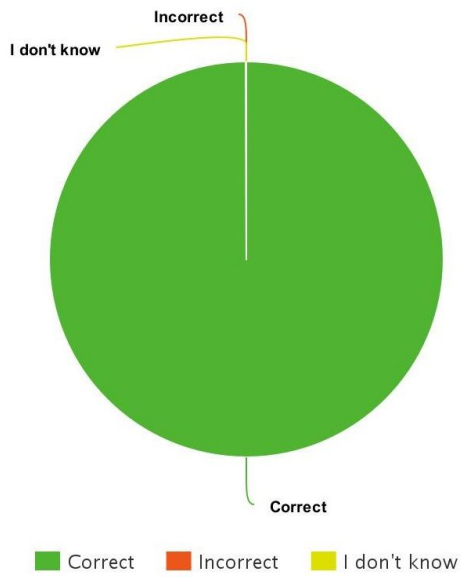
5. What's the definition of functional defects?

6. Which category has the most warnings in package junitTest?



Possible answers were:

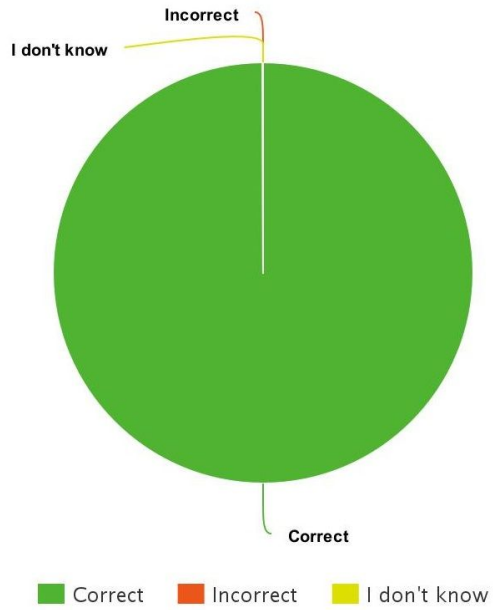
- Rules which are caused by different functions
- **Rules that check for defects that affect the behavior of the program**
- Rules that check that errors and exceptions are properly handled
- I don't know



Possible answers were:

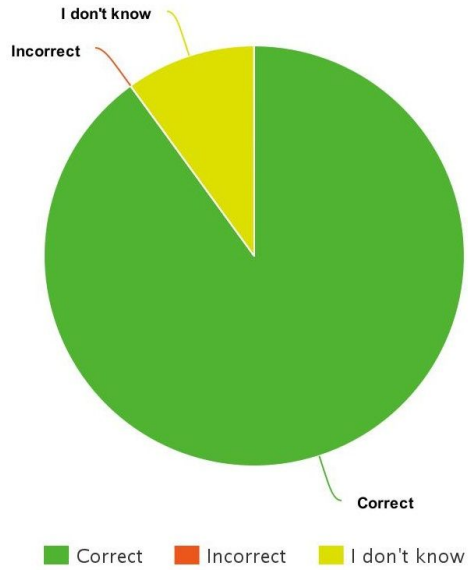
- **Maintainability Defects**
- Functional Defects
- Reusable Defects
- I don't know

7. How many warnings in the project are about Code Structure?



Open question.

8. Which ASATs generate most warnings in LoadPlayersController class in default package?



Possible answers were:

- PMD
- **CheckStyle**
- FindBugs
- Cobertura
- I don't know

Question 9 and answers: *When a package is red in the normal color scale. This means that the package...?*

- Has lots of warnings
- Has the most warnings
- has a lot of warnings
- Has some warnings
- Contains relatively more problems
- Relatively many warnings
- Number of issues
- Has more errors relative to the others

Question 10 and answers: *Is there anything unclear to you? Feedback and suggestions are also welcome.*

- When turning toggles on and off, everything seems to hang. Which is a bit annoying. Maybe a loading bar or something would be nice.
- Klikken op test project moet gaan naar root ipv 1 map omhoog.
- Sometimes it is almost impossible to read the names of the files. Also, some classes have a name on the page while others don't seem to have a name in white while hovering over them does give the class name. Also, when you click on default in the tree you expect to go to that part of the tree but you go one up, which feels strange compared to behaviour of programs in the field.
- I think displaying the statistics next to the cursor is a bit confusing. I'd rather it be in a separate static box next to the visualization. Abbreviations like ASAT is a bit confusing when you don't know what it means. The back button on the top looks like you can go back to a specific folder instead of the previous folder.
- I would set the ASAT color setting default
- It is not very clear whether I am in a project, package or file. The text next to the mouse gets difficult to read when on top of other text