CRANFIELD UNIVERSITY

João Pedro Rodrigues de Almeida

Visualising defects in source code

School of Engineering
MSc in Computational Software Techniques in Engineering

MSc
Academic Year: 2011 - 2012

Supervisor: Stuart Barnes
August/2012

CRANFIELD UNIVERSITY

School of Engineering
MSc in Computational Software Techniques in Engineering

MSc

Academic Year 2011 - 2012

João Pedro Rodrigues de Almeida

Visualising defects in source code

Supervisor: Stuart Barnes
August/2012

This thesis is submitted in partial fulfilment of the requirements for the degree of MSc

Abstract

Debugging and testing phases are usually the tasks that consume most of the resources and time of a software project. Though defect detection can be fully automated, defect localization cannot, as a software developer is needed to solve most encountered defects. This thesis project aims to ease the work of software developers in these tasks by creating a defect visualization framework to automate part of the process of solving defects in source code and allow software developers to save time and resources in their projects.

Keywords: visualization, defects, source code

Acknowledgements

I would like to express my thanks to Dr. Stuart Barnes, whom guidance truly helped me through this master thesis, to Mrs. Pauline Buck, for always supporting her students and making them feel at home, and to my parents, who were and always will be there for me.

Table of Contents

List of figures

1 Introduction

This report is my master thesis for the conclusion of the Computational Software Techniques in Engineering Master Course at the School of Engineering, in Cranfield University. It aims to create a framework that will allow software engineers to visualize defects in source code, in order to help them test and debug their software. As a software developer, I end up losing too much time either in the testing phase or in the debugging phase of my software projects. If there is any way of decreasing the amount of time of these two phases of any project by automating some of the testing or debugging tasks not only the software project will be done in less time and with more quality but also these tasks will be less monotonous of doing.

I chose this subject for my thesis because I believe there is not much done in this area and that my work can help in improving the quality of future software projects and easing the work of software developers in two of the most important phases of any software project: testing and debugging.

Within this chapter I will define and explain some of the subjects of this thesis project as an overview and introduction to the problem and its specifics.

1.1 Defects

Defect is a failure, a fault or a deviation from quality on a software system [1] and is many times referred as error, failure, bug or fault [2]. As a failure, a defect will allow the software's source code to be compiled, though it will generate problems during its execution; as a fault, a compiler won't even be able to compile the source code and, finally, as a deviation from quality, the defect will allow to run the software and will not be noticed unless the software developer is keen to find it [3].

As a software developer, a distinction between the different types of defects has to be made. It is of highly importance to actually take notice that different types of defects must be dealt with in different ways.

Defects should be distinguished in two categories: functional and evolvability [1]. Functional defects affect actual functionalities of the software system while evolvability defects affect the non-functional requirements of the software system during its life time.

The functional defects are the ones we see as a failure or a fault on compilation or execution, and even though they sometimes are not easy to find, a great portion of them are easy to detect [1]. Evolvability defects are hard to detect during the development phase of the software project [4,5]. Though they can be detected after, most of them will not be, since the users of the software normally are not experienced in software development and do not know the development of their software to an extent that would allow them to detect those kind of defects. Even so, although evolvability defects are harder to detect and represent the most part of the defect that are not encountered during the development phase of the project, no one can say a software is free of functional defects either [1,6], since there is always the chance something has escaped the grasp of the software developer, and only in very few rare cases there is not.

1.2 Code Review

No software developer will ever say they will create a software in one go, without code reviewing. Code review is an important task of any project and its process has a direct effect on the final quality of the software [1], though it is many times not so well performed because it is a tedious task and software developers tend to lose focus quickly on such tasks [7,8]. It can be said that a fault will always result from an human error, being that error in the source code or in the lack of it [9].

Code review is always a good instrument to be used by a software developer, a skill they must try to get better and better, so they can prevent as many defects as possible [1]. Nevertheless there are a few approaches and software that can be used in order to ease that amount of work in this particular task.

Defect detection and correction phases of a project are always time consuming [8-12] and sometimes prevent projects from finishing within the time line available, either with manual code reviewing or with debugging and testing tools. Every software developer does code reviewing in their specific way and even so there is no full proof software or method known, it is a matter of minimizing liabilities and work out their set of skill as software developers that can make a difference.

1.2.1 Defect Detection

Finding a defect in the source code is a task that cannot be taken lightly and it should be done by a software developer that really understands what the software is supposed to do at that given function [3,13], that is the reason why normally it is the software developer the responsible for detecting and, more importantly, correct every possible defect.

Also, software developers can ask their peers to do a revision for them after, to an specific fault or to the entire software system, because sometimes a new view over the software can bring new valuable insight and it can help in defect detection [14].

That is not the case of big corporations or big projects where the software developer that created the software is many times not the person that will review its code and the software that will be used is normally project independent [15]. In the maintenance of the code during the life time of the software there is the same issue: How to be sure we are maintaining the functional and non-functional requirements of the software we are reviewing? Even the documentation sometimes does not help or because it is outdated or because it simply does not state every design decision [16].

One more problem to add is that although manual code review is many times done by the software developer, that task usually only detects functional defects [1], and it is very difficult to assure that the non-functional requirements will all be met when making changes to the source code of the software.

Defect detection tools and methods are often used to decrease the amount of defects in the source code as well as ease the work of the software developer [7]. It is important to grab as much as information as possible about the software requirements and at least try to prevent certain aspects of the software to turn worse, like runtime execution for example.

The cost of software defects grows exponentially in the life cycle of a software project [15] so the better the testing and debugging tasks in a development phase of the software project are performed, the less the software project will cost. Those two tasks should also not be performed only once, since they are not immune to the creation of new defects [14,17].

1.2.2 Testing

A programmer will almost always follow some predefined steps in order to correct defects in their software: check test failure, try to figure out the defect, correct the defect and restore the software in order to test it again [3,8]. These tasks are estimated to consume more than a half of the time a programmer is working on the project in most cases [9-12]. So, there are tools to help the software developer, not only creating an extensible test suite based in source code but also giving a feedback of whether the software is giving the correct answers to the given inputs.

Although it is a great help to be sure the functionality of the software is achieved, there are always some concerns about the kind of tests created, that can not contain all system requirements, and also about the feeling that a full passed test suite means that the system has no defects, which is also not true. A test is conducted under specific conditions [2] which means that most of the possible conditions are not tested at all, we just assume those do not need to be tested. A testing suite is always a good start point for defect detection but it is not a standalone solution for software validation and verification.

1.2.3 Debugging

We would assume that every software developer does debugging to their software with an automated tool before releasing it, which is actually not true

[15]. Debugging is all about defect detection and localization [18] and one of the most important tools of a software development. It helps to assure that a given piece of software is doing everything it is supposed to do every step of the way, instead of assuming that by the outputs and inputs, like in testing.

Each software developer has a way of debugging their programs, task that needs that he/she can actually understand the meaning of each variable value in a certain point in the source code. There are many ways of debugging a program [19] and each software developer will do it his/her own way.

1.2.4 Source Code Analysis

Software analysis is a step prior to any defect visualization, but must be taken in consideration when creating a visualising tool, which will bring to the user the output of that analysis [16]. In the case of defect detection, the software analysis tool will have to output all defects of the software so that the visualization tool can properly present the information to the software developer. These analysis can be done using different methods, which will influence the output obtained.

In order to be sure the information it is given to the visualising tool is the best one to present, the software analysis tool method must be the best one for the task in hand, which can differ from project to project [6]. We cannot forget that when a defect detection software detects a defect it is the software developer that will have to actually check the faulty source code and correct it [11,20], so the visualization of those defects is as important as its localization.

In some cases it will be impossible to detect a defect in the source code, even when the software is running [21], because there are libraries that can be used that can create a problem in the operative system itself for example, so the task of any software developer is also not to rely only in the software but also in his/her knowledge of the language and the software system in development.

In the end, the aim is to increase software quality [7]. Software quality decreases with the system vulnerability to faults, either in the source code of the

software system or in the language itself. A software system must abide the language rules in which is made, and sometimes it is necessary to apply some measures in the source code to avoid later exploits of the software system [22]. As is was stated before, it is the responsibility of the software developer to use any software tools at its best capability without discarding the chance of those tools being misdirected or even wrong in some cases.

1.3 Visualization

Visualization is a viable mean of showing and analyzing large amounts of information [5] and a powerful tool that allows a better understanding on a software system by presenting the data in such a way that allows software developers to identify complex regularities and discontinuities [23]. Different techniques and methods can be used to create an appealing visualization of any system that will allow different analysis to be made by the software engineer in order to complete an specific task.

Automated tools often miss details regarding unknown factors and fail to find patterns that are not 100% logical. Automated tools combined with human expertise through visualization normally offer a better solution when analysis is needed [23].

Representing software is difficult as it has no form or tangible perception besides its source code which makes any software visualization bound to the software analysis output [23] as it cannot create a visualization for data which does not exist or was not given to the system to be visualized.

1.4 Software Quality

Although this project itself is not about the quality of the software, it sure increases the quality of any software that may use it to decrease the number of defects in its source code, so software quality should be defined within this project.

Software quality can be seen as anything that can be improved in a software to meet its requirements [2,24,25]. To have the best quality possible is

an ideal objective, though unrealistic, since it is always possible to do better and better, but it is possible to know if the next version of the source code of our project has better quality than the previous one and that is one of the goals of this project, to allow software developers to use a tool that ultimately will always improve their software quality.

Software developers that are always concerned with the quality of their software systems will not only correct known defects but also try to detect the ones that are not visible yet to both solve and prevent the release of faulty software [14].

Although it is possible to measure software quality by the number of defects encountered [24], that can turn out to be inaccurately, since we can have no information about the number of defects which are still in the source code [25]. All we can say is that if a defect is corrected or prevented the software quality increases, despite not knowing in percentage or value by how much.

An interactive process to help software developers to quickly narrow down the search space of the defect and correct it will always help the software developer in their defect detection and location task [18] thus decrease the time needed to conclude that process and increase the overall software quality.

2 Research Plan

This chapter will elucidate the reader about the research done in order to create this document and project.

2.1 Research Aim

This research aims to find the goals of a visualization of defects in source code software solution and to create the best visualization method possible for all types of defect detection software.

2.2 Research Questions

- What is the aim of a visualization of defects in source code software?
- What are the needs of a software developer in such software?

2.3 Boundaries

Although this thesis aims to create a framework for visualization of defects in source code, it will not focus on the defect detection itself, so the methods used by the defect detection software will not be taken into account, only its output, in order to focus the thesis on the visualization itself.

2.4 Research Methods

The research was done in Scopus [26], IEEE Xplore [27] and Google Scholar [28], three of the biggest search engines that contain engineering journals conference papers from many sources.

This research was conducted first into the literature advised by the advisor of this thesis and some of its references, then by searching into the keywords "fault detection", "visualization" and "software analysis" and their possible combinations, in order to create a good spectrum of results. In the end, a few more articles were researched for the different software encountered during the previous research, in order to accurately decide which one would be better for the framework that in being created and which ones had characteristics that should be used or taken into account on this project.

3 Literature Review

In this chapter it will be presented the state of art taken into account to do this project, result of all the research made prior to the work development in order to figure out the best way to visualize defects in source code.

The reader will be able to understand the content of this project, as well as analyze the problem and the different existent solutions.

3.1 Defect Detection and Localization

Any automation tool for defect detection and localization will ease the work of the software developer and save time, resources and costs [9-12], but studies have shown that a defect detection software solution can only detect about half of the existing defects in a software system [6], which creates the necessity of making different solutions work together in order to increase the percentage of the defects detected.

Another problem is that only a few percentage of software engineers actually use a defect detection tool while creating their software mostly because their time consumption in a project and their lack of integration with compilers and other software systems [15].

These problems require a good and practical solution, since defect detection and localization are of most importance for software development [5] as increases its quality. In function libraries or software frameworks the problem increases as defects become historical dependents of other software systems [4].

3.1.1 Software for Source Code Analysis

Testing and debugging consume not only time, but also resources and project budget [9,11,12] so any method that allows a faster defect detection potentially decreases the cost of a project [10].

It is very difficult to establish a mathematical model for fault detection [21], not only because the software languages allow different approaches in the

way of writing the source code to do an specific task, but also because every software system aims for a different set of requirements, which makes each one of them completely different from the others.

In order to automate defect detection in a software development there are many approaches that can be taken [6]. Which method to choose will depend on the project itself and in what information the software developer wants to visualise. The well known methods for defect detection are static analysis, machine learning analysis and suite testing analysis.

All these method contribute in a different way for the verification and validation of the source code, resulting in a better quality piece of software [6], and they can even be used simultaneous. The fact is the greater barrier to any software defect detection is the uncertainty of the way software developers write their source code as the code languages accept numerous ways of creating the same software [21].

3.1.1.1 Static analysis

Static analysis is an method of defect detection that will increase the quality of any software system, as it will make hidden defects in the source code to surface [22]. It is done by most software development software nowadays, since it is an analysis to the source code as a file, without running the program.

For example, static analysis will attempt to find buffer overflows defects in the source code, preventing future security and reliability problems in the software [15].

One of the most used static analysis is a data flow analysis, which has a much wider defect detection than its peers as it saves the change of states of the program, either as a change of values in variables or even in the source code structure. It will relate all variables from a software source code in order to detect inconsistencies, by creating a flow graph [29].

Often the need of detecting defects exists from modifications to the initial code, so a data flow analysis would be able to test only the flow which would

differ from the last version of the code, thus saving a huge amount of time, essential in a software development project [19].

3.1.1.2 Machine learning analysis

Though data flow analysis is pretty consistent for one runtime of the program, a machine learning analysis uses the combination of the analysis of all runtime executions of a software in order to detect evolvability defects [7]. But, in order to do that, the software must be trying to do always the same thing, that meaning the different versions of the code must be trying to achieve exactly the same goals.

For a code review of a software project that aims to improve their source code, this analysis should be the best choice [7]. That doesn't happen most of the times though. Since this method doesn't allow requirement or functionality changes without the change of giving false positives in the defect detection it is risky to use it in this thesis project.

3.1.1.3 Testing suite analysis

In this method the fault detection is based on the test suite to the source code, using their information to narrow down the fault whereabouts [11]. If the test suite saves data for the debugging task, that will allow a faster and most accurately fault detection and localization [9,20].

This kind of analysis allows multiple executions of the code and allows system and unit testing at the same time, without compromising requirement changes, as long as the testing suite changes accordingly. A source code with a high degree of testability will also less likely contain defects [29].

Also, it gives the control of the visualization and analysis to the software developer. That advantage is also its bigger problem: since the software developer has full control over the test suite if he/she fails to address all possible tests, a defect can stay undetected [11].

This approach also allows to test some non-functional requirements as execution time or find bottlenecks in the software by timing the actual tests and saving result for comparison with later code releases.

3.1.2 Defect Localization

To correct a defect first we need to detect and locate it, then solve it. While the first two tasks can be almost single handled by an automated process, given that process can detect and locate that specific defect, the third one must be carried by a software engineer [17]. That fact makes defect localization of key importance on a defect detection software system, since it cannot detect a defect and then do not locate it at all, it must at least give an approximately accurate location of the defect.

Many approaches have been developed to in order to partially automate defect localization, though they cannot substitute the manual approach [18], that is why defect detection software systems should be integrated within a framework that will allow the software developer to easily finish the task of the automated tool. Actually, an experienced and informed software engineer's intuition about the location of a defect is generally correct [17], so there is little an automated tool can do to try to surpass that fact.

3.2 Visualization of Defects

Visualization is a valuable tool to quickly and effectively analyze big data structures [16]. When we talk about software projects with millions of lines of source code in thousands of files it is almost impossible to prevent the software developer losing focus during the debugging task and a lot of time searching for the line of code where the defect might be. Defect detection and localization is turned into a task almost impossible to quickly achieve successfully without a visualization tool. Visualization can help to locate faulty source code by managing the visual output of the software in a way that helps the software developer fulfil the debugging task easily [10].

Most of software developers consider visualization tools important in the software development process [16] as they were created in order to allow

software developers not only to analyze a big number of files and lines of code at the same time but also to prevent loss of time in the overall software project development or in its maintenance. It is possible to use colour, brightness and even contrast in order to highlight the errors within the source code [10].

Two of the most important tasks in a visualization software system is the definition of what will be visualized and how that information will be presented [23], as it is of most importance for any visualizing tool to show the right amount of information for its viewers [3]. To show more than the necessary information can make the viewer lose focus or even miss what is important on that specific task and to show less will make the viewer lose valuable information and possibly make wrong decisions within the scope of the project.

3.2.1 Requirements for Software Visualization Tools

A survey about functional and quality requirements for software visualization tools showed that the most important quality attributes of a software visualization tool are rendering scalability, information scalability, interoperability, customizability, interactivity, usability and adoptability [30]. So, a defect detection software tool should have no scalability or performance problems, the information to be shown should always be perceptible and its views and objects should be easy to customize, use and interact with.

The same survey showed that the functional requirements required for a visualization tool are views, abstraction, search, filters, code proximity, automatic layouts, history, colour, notes, zooms, pans, delete and edit entries and save and load options [30]. In the case of a defect detection visualization tool, some of this functional requirements may depend on the defect detection system, though if possible they should exist within the visualization system.

All these requirements are important for a defect detection software tool because as it was said before, poor integration is one of the reasons why software engineers do not use defect detection tools [15].

Regarding the specifics of a defect detection tool, all defects should have a category [1,17] as they should be solved based on a priority system [17].

Categorizing defects is one of the most important tasks of a defect detection system as most of them do not affect functionalities of the system [1] and should have less priority than the ones that do [17].

3.2.2 Software Tools for Defect Detection and Visualization

Defect visualising tools are one of the most important parts of a software developer software configuration, though it is many times undervalued or present but not noticed. This kind of tools run many times under the surface of the user interface of many compilers, so software developers tend to assume they are part of the system and sometimes forget they can be improved. Actually many defect visualising tools are made only for certain purposes or goals and that makes them unsuitable for general defect detection and localization [16].

Most of defect detection tools detect defects on the source code, but that does not have to be that way. Defect detection tools can also defect defects on the software structure or even in given information about different runtime behaviours [8] and then locate the defect in the source code.

Every defect detection software solution has its defect visualization, even if not graphical at all like in the case of most compilers, that just give a statement of the defects in text. This is the case of *Eclipse* [31] and *Microsoft Visual Studio* [32] for example. There are also tools that provide to the software developer a graphical interface and present all the errors on his/her software in a view, like *Tarantula* [33] or *xSlice*, a tool within *Cleanscape TestWise* [34], and that is the kind of solution this project aims to be.

Most of defect detection software provides a non-graphical view of the errors similar to a compiler, that is the case of the ones found during this research: *Cleanscape C++ Lint* [35], *Cleanscape LintPlus for C* [36], *Cleanscape FortranLint* [37], *Coverity Scan* [38], *FADA toolkit* [39] and *Cppcheck* [40].

All this defect detection software have a few things in common like the type of information they display (type of error, line and file in which the error was

encountered and error message) and the fact that the visualization is used only for that specific defect detection tool [31-40]. That means that depending on the defect detection tool used the software developer will get a different output, fact that will make the software developer to lose a great amount of time in defect localization when using more than one defect detection tool. Another problem is that each defect detection tool is used for a single code language and that makes the visualization tool bound to that language as well.

Since this project is focused in the visualization of defects in source code and not in the defect detection itself, it is of most importance that the solution will be able to get as much information as possible from the defect detection tools. The output of each one of the defect detection tools should be analyzed in order to guarantee that the visualization tool is able to show the necessary information, which is one of the most important goals in a visualization system [3].

Tarantula and *xSlice* are test-based defect detection solutions [33,34] and they have graphical visualization methods that will be presented on the next section of this paper.

3.2.3 Visualization Methods and techniques

Not only a visualization system should allow the viewer to choose what to see, but also it should allow different types of views over the same data and different approaches to the same view [23]. Different methods and techniques have been developed in order to fulfil the need of software developers on automated tools for defect visualization.

3.2.3.1 Meta-model for Visualization Tools

A visualization tool always includes one or more graphical representations and views of one or more types of data [16]. In the case of defect detection, the object of the visualization tool is the output data of a defect detection software and not the source code or the defects themselves, because they do not have a natural representation [23].

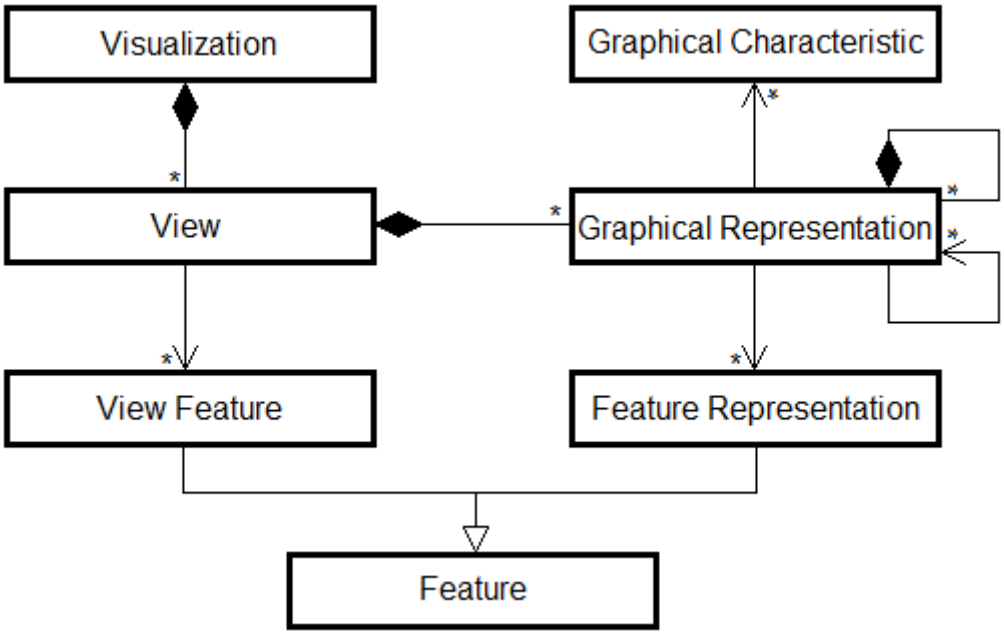

Figure 1 - Meta-model of visualization tools

Representation problems aside, when creating a visualization tool, a software developer should always aim for a generic framework rather than one that will only be useful for a specific task or a specific project [16], so it is important to define first a meta-model of the visualization to be developed.

Generically, a visualization tool links the various graphical representations the user needs to visualize by allowing the user to use exploring and acting features in a certain amount and type of views [16] that will allow the user to browse through the different representations and explore each one of them to find the necessary information.

As we can see from Figure 1, graphical representation can include other representations within, allowing different layouts and layers to be defined within the visualization tool, giving the opportunity to the viewer to choose what to see and how to see it, feature of most importance in visualizing tools [23].

3.2.3.2 Visual Representations and Interactions

There are three main properties of the software that are used in visualization: software structure, runtime behaviour and the source code itself [8]. In this project we will focus on the source code, as software defects are bound to a line of code.

The four general graphical representations of a software source code are line, pixel, summary and hierarchical representations [8]. While the first two represent each line of the source code a line or a pixel respectively, the last ones are used when the number of lines of code is greater than the size of the actual visual representation or when the software is not using any scrolling feature: a summary representation within the different files and a hierarchical representation across all files.

From Figure 2 it is easy to understand how the line representation of the source code. By preserving indentation the representation makes it easy to search for the line of code in the code itself once we go from the representation to the actual file.

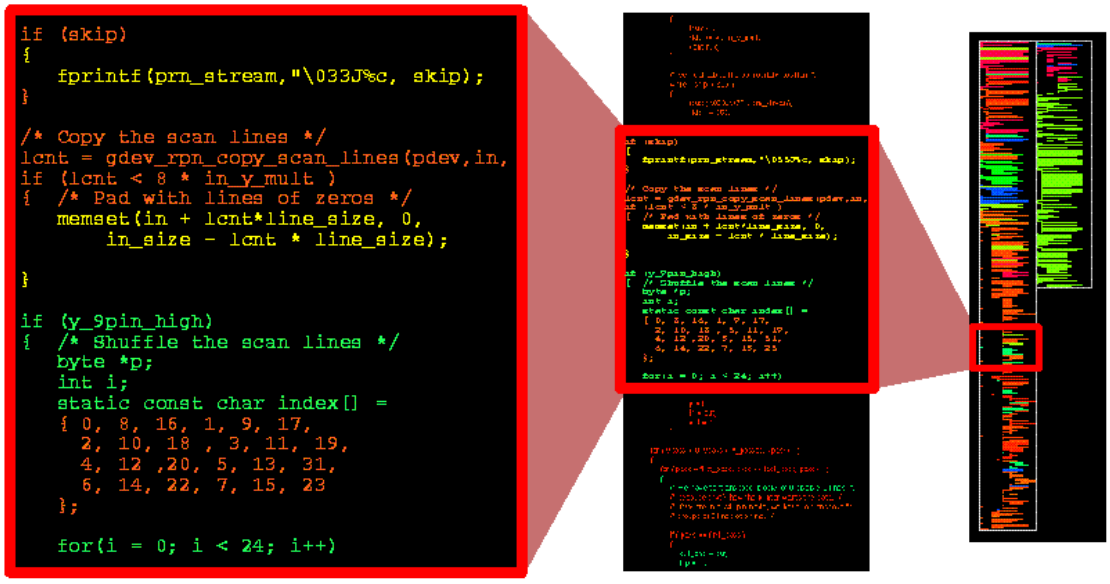

Figure 2 - Line representation of the source code

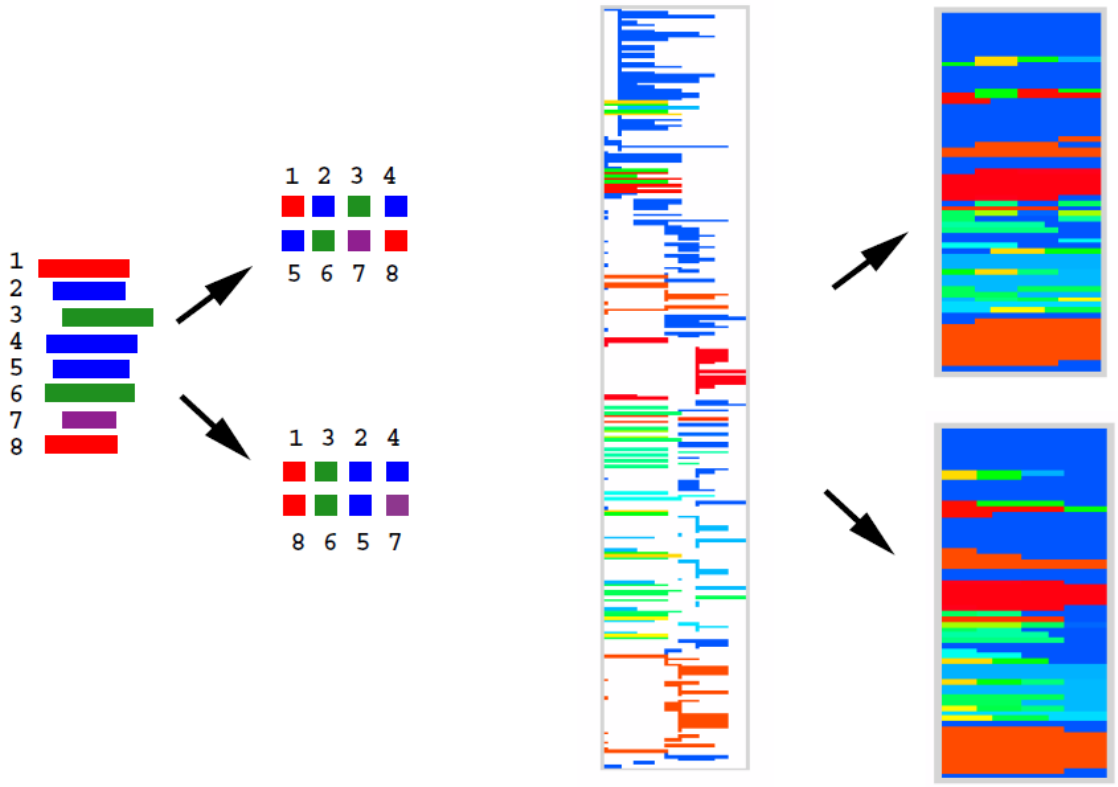

Figure 3 - Pixel and line representations of the source code

The problem with indentation is that sometimes it is useful to fill the line to both margins (as shown in Figure 3) in order to prevent the user to miss some of the lines [8]. Another problem with the indentation is that if we keep the real length ratio of the source code we may end with some lines too small or too large in the visualization.

As we can see by Figure 3, the pixel representation of the code will allow the visualization of much larger data sets without the use of scrollbars, since we can represent more in less space [8]. Another advantage is that we can represent the lines in another order, as in the line representation without indentation, since they are no longer bound to the perception of the source code by the user. Pixel representation will also ease the user need to find patterns on the code, as although a pixel is small, its colour is easy perceivable [8].

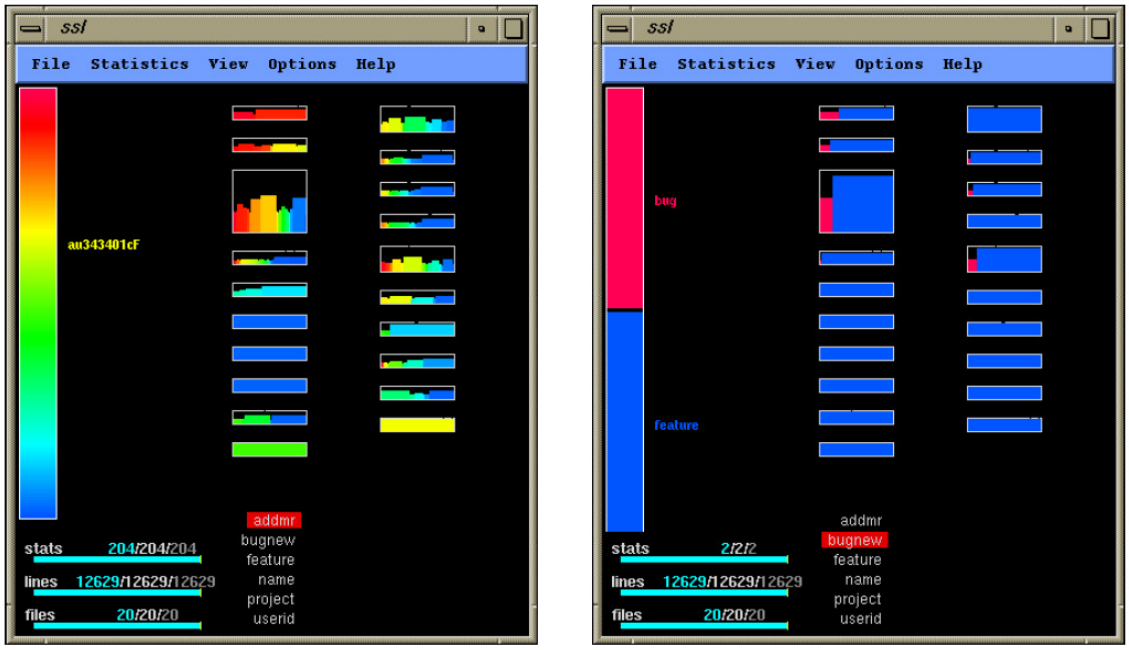

Figure 4 - Summary representation of the source code

As proved by Figure 4, the summary representation is handy for quick analysis of the code, where details are not important and the user rather wants to see the whole source code within the same view. An hierarchical approach (see Figure 5) can be taken to link the summary representation and a more detailed one (like a pixel or line representation) to allow a visualization tool to be fully capable of giving the information the user requires at certain moment.

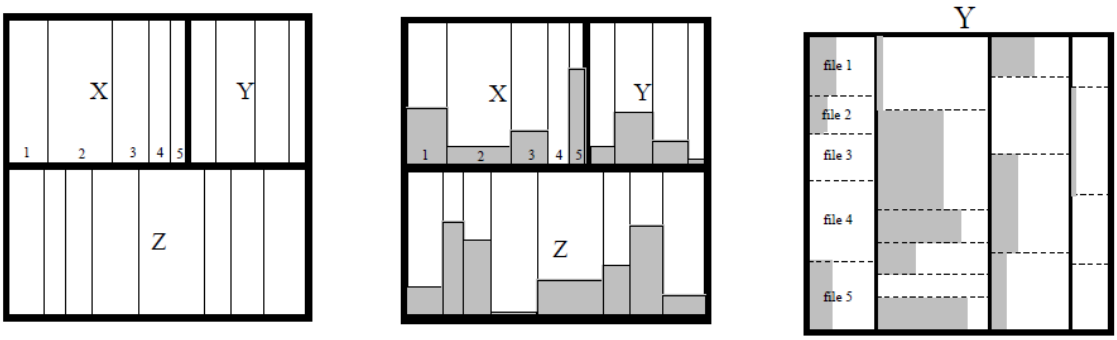

Figure 5 - Hierarchical representation of the source code

3.2.3.3 System Radiography and Bug Watch

System Radiography and Bug Watch are two software defect visualization that complement each other, as System Radiography is suited for an overall view of the system and bug distribution and Bug Watcher for understanding the phase transitions and the specifics of a single defect [4,5].

These two visualizations are time-based, meaning they require a time-frame [5], so they are good tools for the detection of evolutionary defects and for the cross-reference of system versions and defect analysis [4]. That means that these two visualization are more useful in system maintenance rather than in the development phase of the software system, in which the time frame is much shorter and many times does not carry any meaning related to the defect.

In a System Radiography components are grouped by products and assigned to a line and its colour varies with the number of detected defects, in order to be easy to perceive where the defects are concentrated [5]. As we can see from the example in Figure 6, there are 5 components where the number of detected defects is higher, so in this case those components should be the first ones to be debugged.

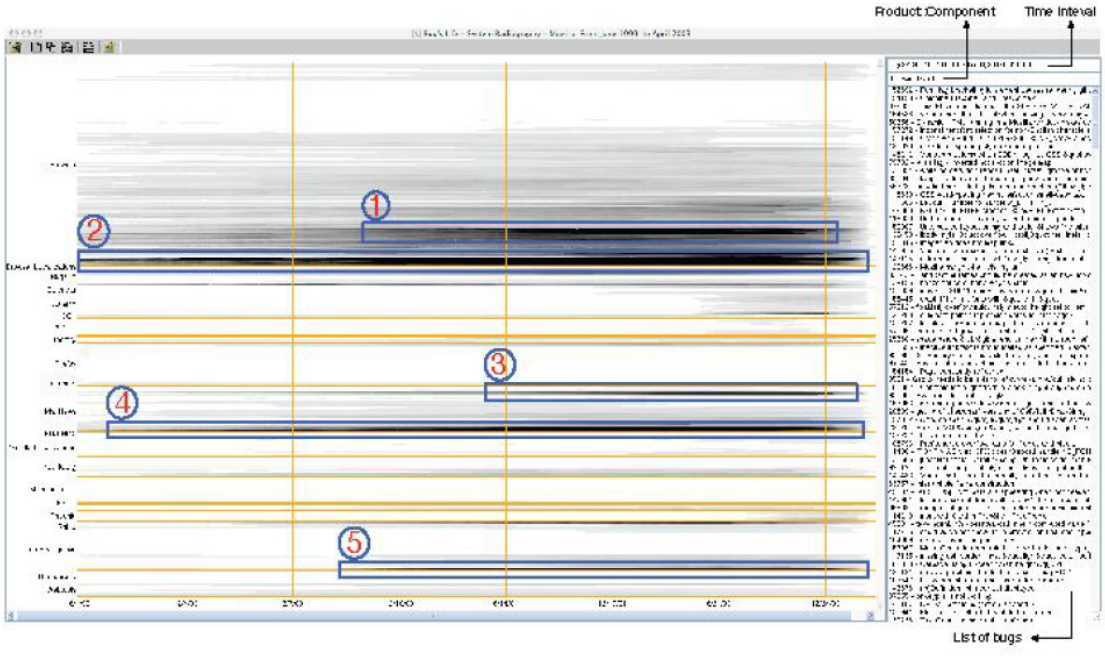

Figure 6 - System Radiography

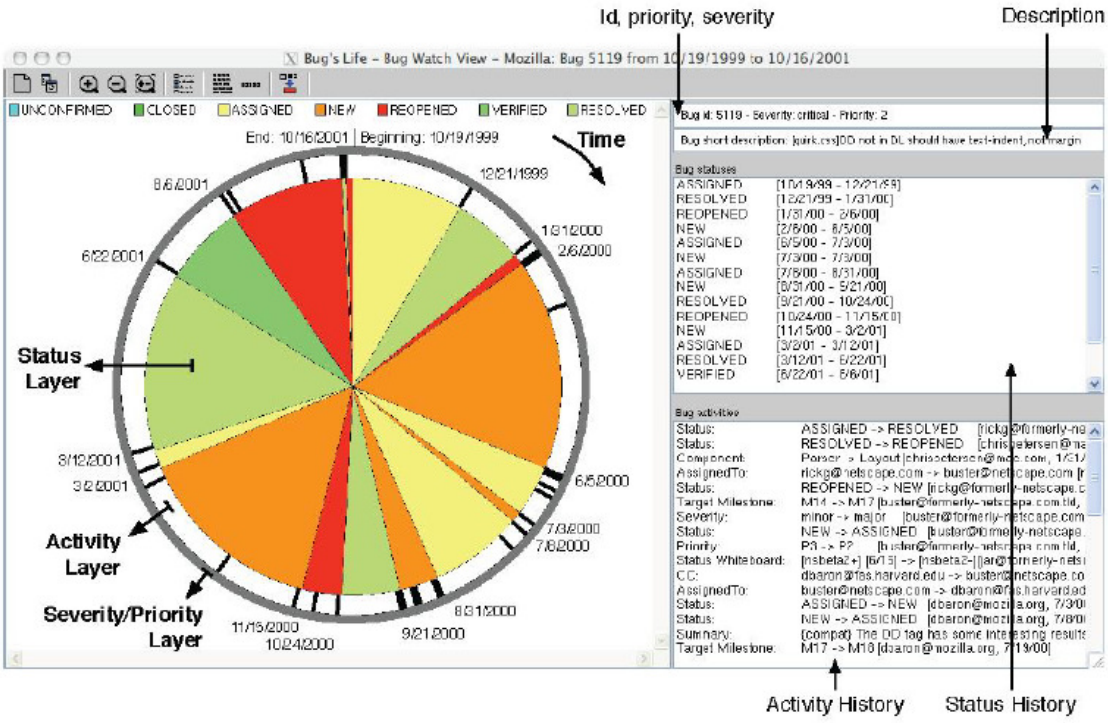

Figure 7 - Bug Watcher

A Bug Watch visualization aims to give the maximum information about an specific defect and helps the viewer to understand the many defect status transitions [4,5], as it can be seen in Figure 7. It is a type of visualization that works well with a single defect, but it fails to be a visualization tool for many defects at once, as the defect status transitions and defect information can become imperceptible to the viewer [5].

In the end, the ideal is to use System Radiography for the system overview and the Bug Watcher for the visualization of the defects [5].

3.2.3.4 Seesoft

Seesoft is a line oriented visualization tool oriented for software statistics [41] and can be used to visualize defects in source code. It is based in four key ideas: reduced representation, colouring by statistic, direct manipulation and capability to read the actual code [41].

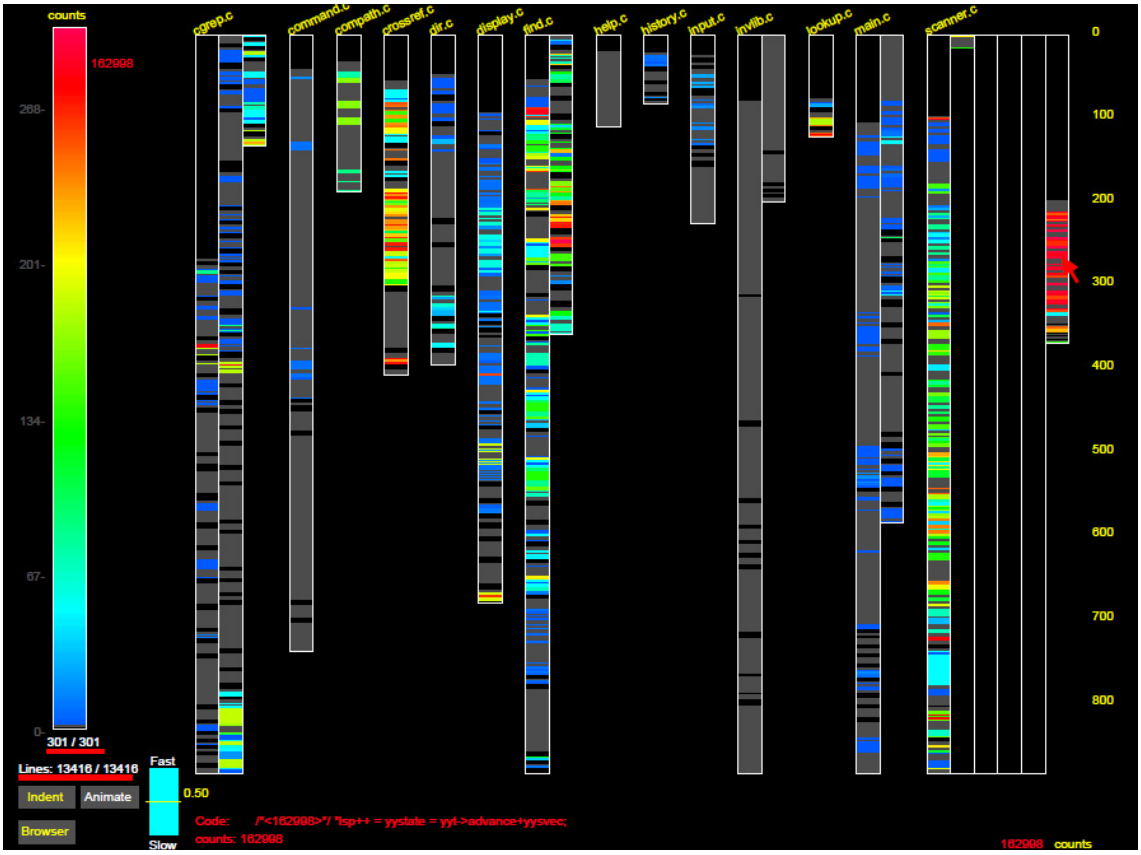

Figure 8 - Seesoft

As it is shown in Figure 8, *Seesoft* displays files as columns and lines as thin rows, providing a qualitative view of the distribution of the code characteristics [41]. For a defect visualization tool, this method is intuitive to use and it allows a quick localization of a defect.

Seesoft is useful to discover patterns in small projects, but not suitable for bigger ones. As it shows the entire project in one visualization, the bigger the project or the files, the smaller the lines will be, which will make them imperceptible [41]. This can be solved with visualization features though, like filtering and zooming, with a search engine or even with a hierarchical representation of the source code, creating more layers to the view.

3.2.3.5 Tarantula

Tarantula is a visualization system that is cross-references the lines of code with a test suite and colours the lines based on the amounts of passed and failed tests that crossed that line of code [33].

Though it is has an entirely different design, it is based in the *Seesoft* visualization system [3,10], some of the differences being the detailed information and code outside the visualization view to solve the scalability problem referred in the previous section, the mapping of the source code lines by brightness and not only by colour and the possibility to change to different types of visualization on the same data to maximize its usability and perception. As we can see from Figure 9, it is now possible to address much more lines of code in a single visualization.

Tarantula only shows one file at a time and has some usability problems like scrolling both vertically and horizontally, is still in its beta state at this point [33], so a further analysis of this method and software will only be possible after its release.

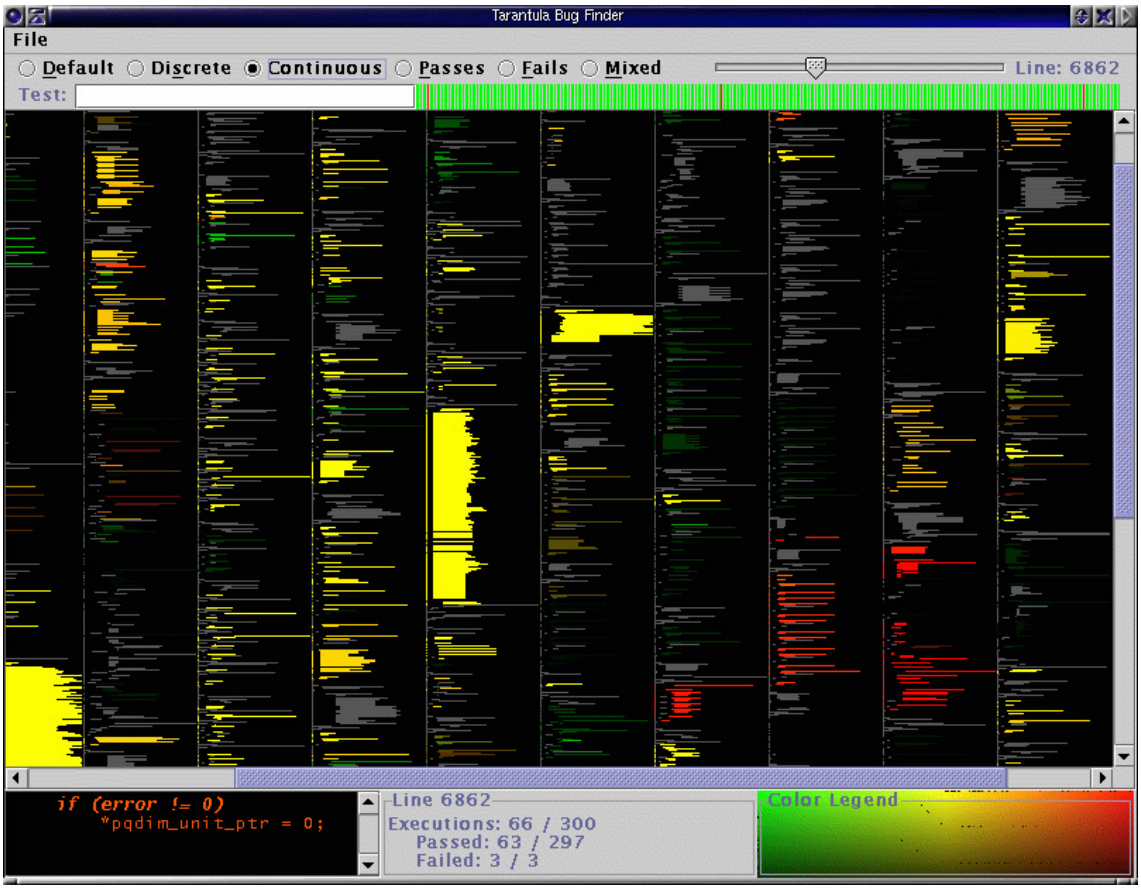

Figure 9 - Tarantula

3.2.3.6 xSlice

xSlice is a program slicing debugger based on software testing [34]. The main difference between *xSlice* and its peers is that it bases its analysis on software slices and statements, and not on the whole source code and its lines of code.

A slice is a set of statements from the source code. In this case a slice represents the statements affected by a test case [13]. *xSlice* applies a colour mapping to each slice of the software, allowing the viewer to visualize the faulty statements, the statements in that particular slice and all other statements in different colours [10], as it can be seen by Figure 10.

Further analysis on the methods of *xSlice* will require the purchase of this solution, as it is a commercial one.

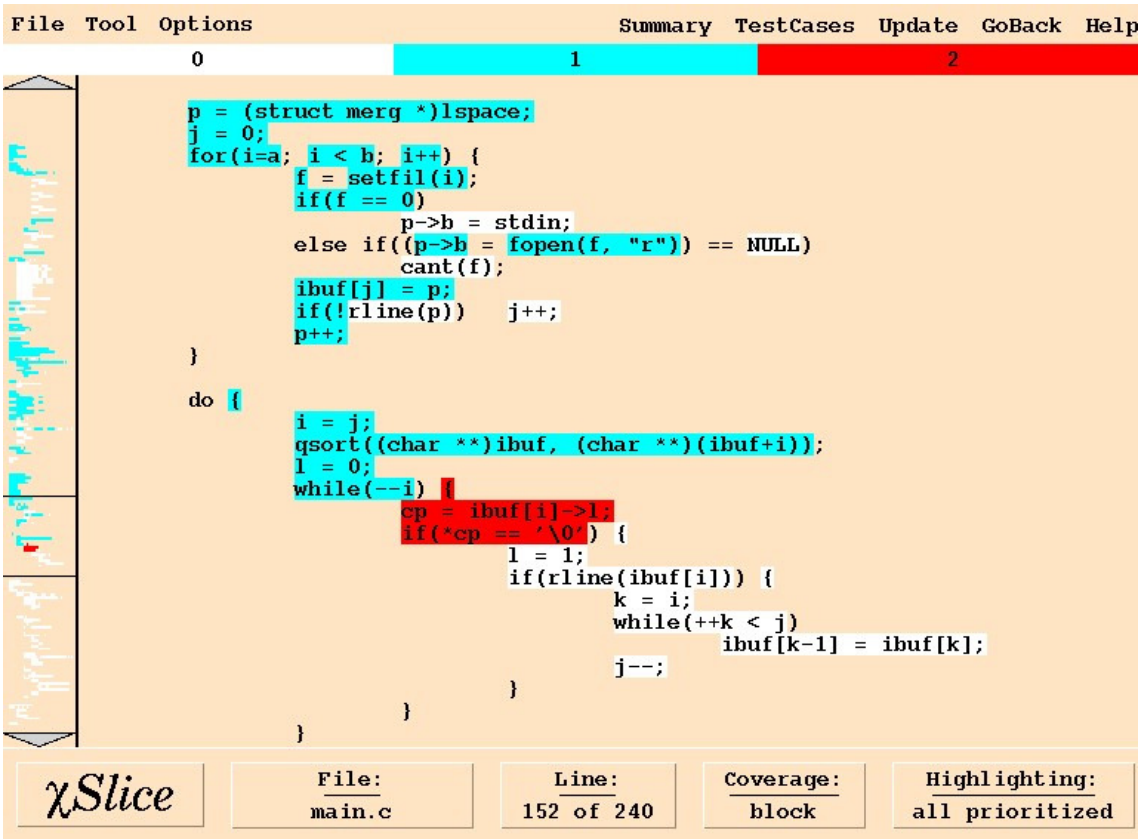

Figure 10 - xSlice

4 Project

The project of this thesis is a defect visualizing tool that was developed based on a methodology to be explained later on this paper. An analysis and discussion of the results will also be done further on, after the overview of all the requirements gathered during the research for this thesis project.

4.1 Requirements: Problems and Solutions

The requirements, problems and solutions encountered during this research should be enunciated for later explanation of the specifics of the methodology used in this thesis product. To better understand the types of problems normally encountered and to structure its presentation, they were separated in six types of quality requirements: scalability, interactivity, customizability, interoperability and usability.

4.1.1 Scalability

Scalability is a requirement any software system should be able to meet. In the case of defect visualization, if scalability is not properly met as a requirement, the use of the visualization system will be restricted to small projects. The bigger the software system, the bigger the necessity of using visualization for defect detection and localization, so any visualization system should be able to scale up as maximum as possible.

In the case of defect visualization, there are two main problems regarding scalability: the rendering speed and the scalability of the information to be seen on the view of the visualization system.

4.1.1.1 Rendering Speed

One of the reasons most pointed by software engineers not to use defect detection software is their time consumption, therefore it is of extreme importance that the visualization tool do not consume unnecessary time. Discarding the time usage of the defect detection tool, which is not the object of this thesis project, there are a number of effective ways of reducing the rendering speed of a visualization tool.

Each view can be computed either in the beginning of the visualization process or in a demand system where the views are computed as they are requested by the viewer. Another problem is the amount of information to be computed, which should never be more than the necessary, not to create bottlenecks in the rendering of each view. Though less information normally means faster computation, it is not good to have less information than the necessary either, as the software engineer may miss some important aspect of the software or the defect if not presented with all the information needed.

The choices of libraries and the objects for the visualization tool should also be taken carefully, as the more complex they are, the more time the computer will need to render the views.

4.1.1.2 Information Scalability

The scalability of a visualization tool is not only about performance, as each view has graphical aspects that must be taken into account when scaling up or down. Every graphical aspect of a view is composed by pixels, though a pixel can be seen as the limit of any scaling system.

The objects chosen for a visualization tool should not only be simple, but also easily scalable without losing its properties. In a screen, a square will always be more scalable a circle for example, because it can be represented as a pixel and a circle cannot without losing its properties, as we can see from the example shown in Figure 11. If more than one type of objects is represented in the same view, one should always aim to choose forms and aspects of the objects which will not disappear when scaling the view.

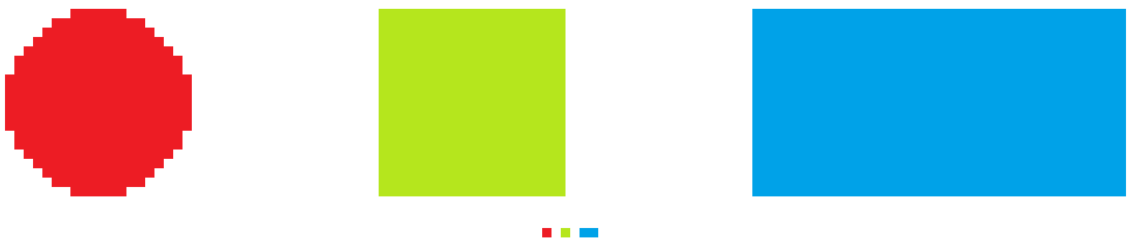

Figure 11 - Results of scaling different objects in a view

The more scalable a view is, the more objects it can represent within its area, though there is always a maximum number of possible representations, as a screen as a limited number of pixels (and not necessarily equal to the number of pixels as a pixel can represent more than one object). As we can see from Figure 11, using a square instead of using a rectangle in a representation of an object in a view for example will also increase the amount of objects the view can support.

The ability to represent lines of source code as rectangles or pixels is not a novelty and as it was already said in this thesis paper, a visualization tool can and should be able to represent in different ways the same data.

4.1.2 Interactivity

A software system must be interactive. That means simply that the way to navigate through the system must be easily perceptible and should not chance as the user navigates.

In a visualization system, it is important to keep the panels always in the same place in the view and create tools that can be used in all views, not just in some of them.

4.1.2.1 View Navigation

View Navigation is of extreme importance for the user, as it is the way he/she will interact with the system to obtain the necessary information. Each view should always have navigation tools that must work in any possible case scenario. If a view cannot be navigated in all of its extension, the user will lose information that cannot be recuperated.

A view should also be a fair abstraction of its object, to be easily understood by the user. Like it was said before in this thesis paper, the more approaches to view that can be taken the better.

In the case of defect visualization, the four basic approaches to the source code view should always be taken into consideration in the build of a

defect visualization system: pixel, line, summary and hierarchical representations.

4.1.2.2 View Integration

A visualization system normally is composed by more than one view which creates several layers of visualization. Those layers must be connected logically and it should be easy to navigate between them.

There are several problems that must be avoided while creating different layers in a visualization system. Each lower layer of the view must be fully integrated to the previous one, so that every view on the system becomes accessible. Information that needs to be passed between different layers and views must be saved by the system in order to ensure the user is opening the desired view.

4.1.3 Customizability

Even the most specific task or software system can be made in different ways by different people, so customizability becomes important when creating a software tool that aims to be used by many different people.

For a visualization system, not only the visualization but also the way the information is gathered and rendered should always be presented to the user as something that can be changed in some sort or by options or by different approaches to the software.

4.1.3.1 Input Files

In a visualization system the information must first be gathered and it is normally contained in a file. That file should be easy to write, read and understand, not only to prevent bad usage and following miss representation in the visualization system, but also to allow any defect detection tool to easily output the desirable file for visualization.

The format of the file is also important, as it should not be protected by the system and should also be readable in any operative system by other

software than the visualization tool (for debugging and researching tasks of the defect detection tool).

4.1.3.2 View Options

View Options are very important in a visualization system, as the aim of the visualization is not only to show the data but also to allow the viewer to work with the visual output.

Options like zoom, panes, edit, delete, save, load, filter and search should be present any time it is possible, as it will improve the usability of the software system as it will prevent users from not using the visualization system because of lacking of options or visualization tools.

Options to the view itself are also desirable, as different users will have different hardware where they use the visualization and not always the same visualization is well presented in every piece of hardware. Another common problem is the inability of a view to show the data properly in different monitor resolutions, and that is something it needs to be though about when creating a visualization tool.

In the case of defect detection tools, the proper tools for the solution of the defect and the proper view options are of key importance, to be sure the software developer can not only easily solve the defects that are shown in the visualization view but also see all the defects that the software actually have.

4.1.3.3 Defect Prioritization

Regarding specifically defect visualization tools, there is a common problem that must be taken into account when creating a visualization of defects. Defects have, besides their id/name, location and description, a category/priority that must be treated in a special way.

When debugging a software, the software developer follows a priority system, which means we will solve first the defect with certain characteristics. A software developer must be able to filter the data by defect category or priority, either with filtering or searching options, in order to prevent a great loss of time

usage in searching the defects with the same priority level before solving them before passing to the next priority level.

4.1.4 Usability

As a tool, a software system must not have any usability issues as it will require more time for the user not only to overcome them but also to work around them. The user interface, the software assistance and, in the case of any visualization tool, the view usage must be as good as possible.

4.1.4.1 User interface

The general user interface of any system is important as it is what it feels the gap between the software system and its user and it allows that user to work with the system.

In a defect visualization tool, the user interface must be able to allow its user to quickly refresh the view with new information, edit the file with the defect, find the next defect to correct and get a global summary of the project. Other options are also important as the navigation and utilities of the view, but these four elements are the key for the debugging task in hand that must be present for the software developer to make use of the system.

4.1.4.2 View Usage

It was said before in this paper that view options are very important in a visualization system to allow the viewer to work with the visual output and that each view should have as many options as possible. Another problem arises when the options exist but are not perceptible or the place they exist are not conducive to its use.

The view usage must always be intuitive and easy to percept or, if not, the software should have a quick introduction to the tools and how to use them, like a tutorial for example.

4.1.5 Interoperability

When a software system is working within a framework, its interoperability is essential to the aim of that framework, whatever it might be. The input and output of each system must be well defined and its characteristics should be as open to change as possible, as long as do not corrupt the entire framework.

In the present case, the defect visualization tool is a bridge between the defect detection tool and the file editor, as the software developer uses a defect detection tool to detect the defect, the defect visualization tool to locate it and the file editor to solve it.

4.1.5.1 Defect Detection Tool Integration

Defect Detection tools output must be consider to define an input system for the visualization tool. Not only the type of information but the actual composition of the file must be entirely defined regarding not only different tools and systems as well as different technologies.

The visualization tool should be able to read the input file regardless of the defect detection tool that created it, as long as the defect detection tool follows the rules for that file imposed by the visualization system.

The creation of those rules must also take into consideration not only the vital information for rendering the view but also the information that may be necessary for the software developer to locate and solve every possible defect encountered.

4.1.5.2 Editor Integration

The visualization tool could allow a file edition within its views, but in the case of defect visualization that is not the best solution.

The debugging task is not free of new defects and any compiler can easily discover many of the errors mostly common done by software developers during the development phase of the project, so the ideal solution is to open those compilers or a file editor with programming attributes that will allow the

software developer not only to change the software in a known environment but also will prevent the creation of many more defects.

As it is not bound to any code language, the user of the visualization tool must be able to change the editor in which it pretends to solve the defect.

4.1.5.3 Time Consumption

Time Consumption is one of the most important reason why software developers do not use defect detection tools, so integration between those tools and the file editors through visualization is key for the successful accomplish of this thesis project, as it translates in a major time saving.

That integration should be extended as far as possible, as long as it not constitutes a threat to any other goal of the visualization tool.

4.1.6 Adaptability

Adaptability is one of the most difficult aspects of the software system to be taken care of, since there is no way of telling beforehand what will be expected from the software system during its life cycle. In this precise case, no one can tell what will be chances in the requirements for defect visualization in the next development of a defect detection tool.

What it should be guaranteed is that the visualization system will be able to adapt to new situations and can be easily improved without jeopardizing previous work.

The system must be able to get information from different defect detection tools, regardless of their methods and systems. That means that the connection between the visualization and the defect detection software must be done in a way that can be easily extended to any defect detection tool.

The system should also be able to visualize defects in all types of software in any source code language, so it cannot depend on specifics of any type.

At last, the visualization tool must be able to be used in most of operative systems, so that it can be inserted in the same system as the defect detection tool if necessary.

4.2 Methodology

The methodology used in the defect visualization tool created during this thesis project is based on the literature review presented so far. This software aims to be used within a defect detection framework that will now be explained.

4.2.1 Visualization tool

This visualization tool was developed in a Java Environment, as its executable file is not bound to any platform and it can executed in almost all current platforms known. Its purpose is to become an interface to be used by defect detection tools to communicate with file editors and compilers, as in Figure 12. In order to allow the software developer to save time in its debugging task, he/she can use this visualization to locate the defects in the source code and then use the file editor to solve them.

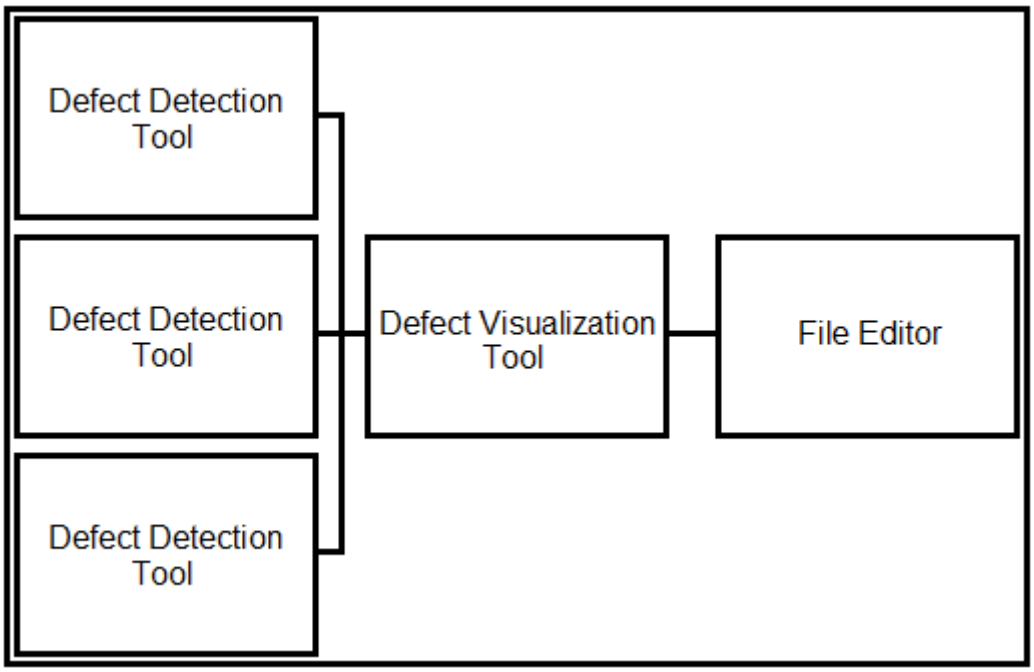

Figure 12 - Defect Visualization Tool Framework

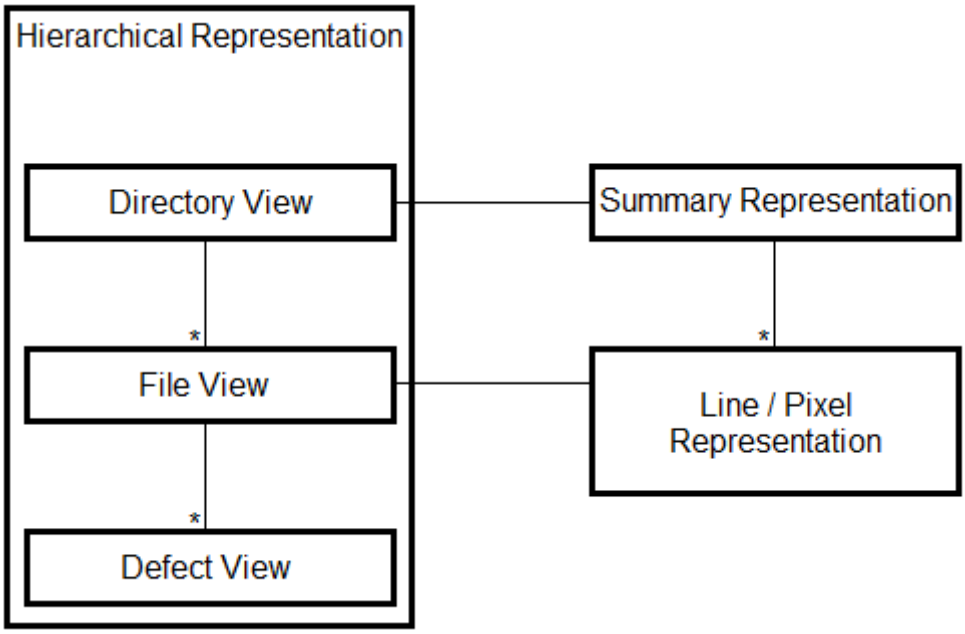

Figure 13 - Defect Visualization Tool Views and Representations

Within the defect detection tool, three visualization layers were created, each one with its view, represented in Figure 13, in order to provide the user the four graphical types of visualization views referred in the literature review.

This approach was taken also to ensure a faster localization by the software developer of the defect he/she wants to correct. While the directory view presents all the files within the project (not only a directory in the directory file system of the operating system, but all files presented in the input file, which will be referred further in this thesis paper), the file view allows the user to see all the defects within a file, and finally the defect view specifies a chosen error to be corrected.

The creation of these three layers were necessary for different reasons and serve many purposes as it will be discussed in the continuation of this chapter.

4.2.2 Framework Integration

The integration of the defect visualization tool is made by an input file and command lines. The input file is a XML file that should be created by the

defect detection tool, so that every defect detection tool will be able to work with the defect visualization tool. The file in XML should have all the input information needed for the visualization tool to be able to show it. An example schema of the XML file one should create to work with this tool can be found in the Appendix A of this paper.

If a defect detection tool cannot provide the necessary XML file as the visualization tool needs it is necessary to include a parser in the framework, before the defect visualization tool, to ensure the information required is in the defect visualization tool XML input file. The same process can be applied to quickly merge the information provided by several defect detection tools in order to create a single input file with all the information.

The fact that different defect detection tools treat their information differently makes it impossible for the defect visualization tool to be able to search for the rest of the information if not all provided in the input file, so the input file must complete and trustful information (the only exception could be the actual line of code, but it would be a great performance lost in making the visualization tool to search for such information when the defect detection tool has it as it had to analyse it, so the input file should always contain it, with all the information required).

Currently there are no defect detection tools that have the necessary output for this defect visualization tool, though some of them already include the option of outputting the information as a XML file, as it is the case of *Cppcheck* [40]. A full example of how the defect visualization tool currently works within a framework is presented in the end of this section of the thesis paper, after the description of the defect visualization tool.

The integration with the defect detection tool and the file editor is made in the options of the visualization tool, to ensure that the software developer doesn't need to manually change between tools.

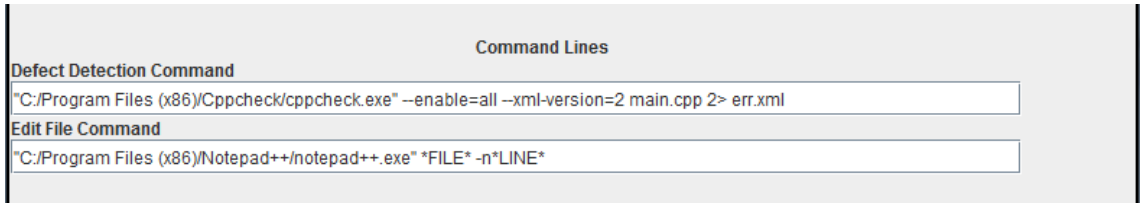

Figure 14 - Command Lines Options

Figure 14 is an example of how the tool can be used with *Cppcheck* [40] and *Notepad++* [42] (the *Cppcheck* command will work but it will not produce the input desired for the visualization tool as it is, as the XML schema differs, to use *Cppcheck* a batch file would be needed as after executing *Cppcheck* it would be necessary to execute a tool to parse the file to the desired schema).

To fully integrate any possible editor, the user can use the tokens *FILE* and *LINE* to define where is the command those should be, and the defect visualization tool will change them to the desired defect attributes in the debugging process.

Using command lines and XML files, the defect visualization tool makes it possible to be used with any compiler, file editor and defect detection tool. It makes it possible as well to work simultaneous with more than one defect detection tool, using batch files.

4.2.3 Directory View

The directory view presents to the user all the files within a project in a simple representation, an abstraction of a directory file to allow the user a simple recognition of what the view is (not an actual directory file on the hard drive, but rather a list of all the files within a project).

Figure 15 is a directory view example with 4 files. The name of the files and the colours are defined by the XML input file, so the defect detection can prioritize or categorize files at will. Plus, each file will show a tooltip with the number of errors encountered.

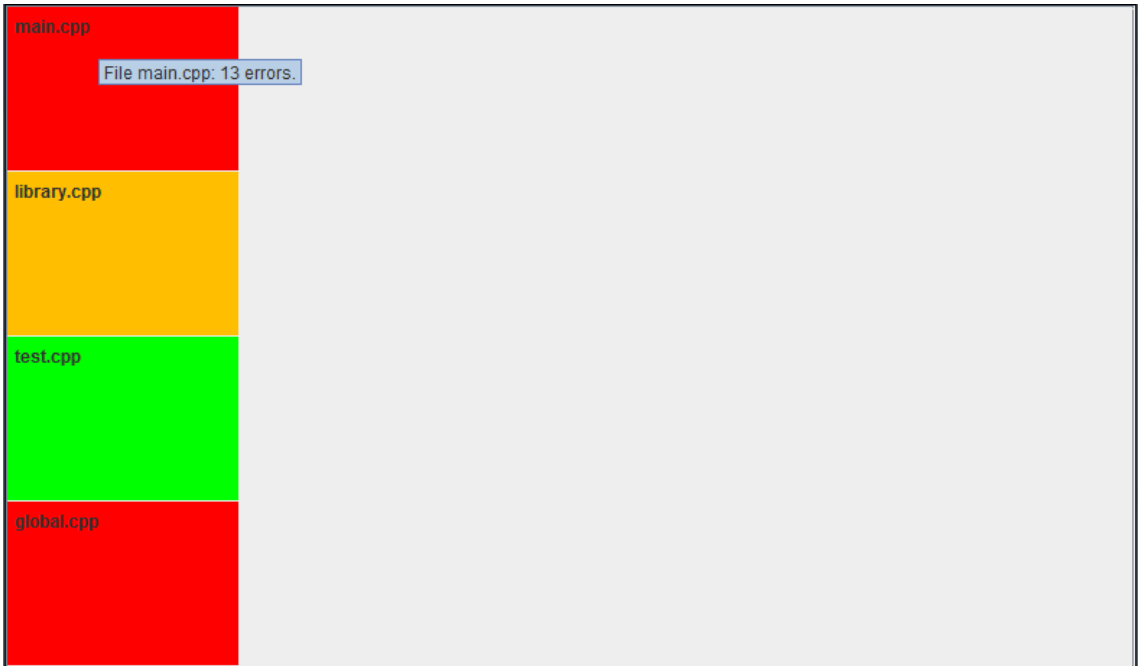

Figure 15 - Directory View

A good example of prioritizing (and shown in Figure 15) is the colouring of the file by the same colour as its defect with most priority, but other colouring techniques may apply. Depending on the defect detection tool capability it is possible to define files by package, folder or class within the software or even categorize the files per software developer in charge of them (if there is a team of 5 people the defect detection tool will be able to assign files or packages to each individual).

The scalability of the directory view is assured by a horizontal scroll, which will appear automatically if necessary, and the width and the amount of rows in the view can be easily changed, as shown by figure 16.

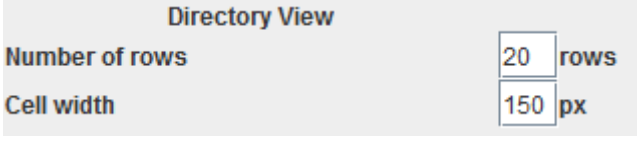

Figure 16 - Directory View Options

4.2.4 File View

The file view presents to the user the lines of code which have or have not defects. Based on the *Seesoft* [41] visualization, it presents the lines of code as full bars in a range of colours representing the presence or not of a certain type of error. As directory view, scalability is assured by an horizontal scrolling bar when the screen cannot support the entire view at once.

By Figure 17 it is possible to see that the view does not present the indentation of the code as the *Tarantula* system, as by doing so some lines of the code could become imperceptible, and because it is not an essential or many times even desirable feature. Besides its line representation, with the possibility of changing the number of rows and the width of the "lines of code", as shown in Figure 18, makes it possible to create a pixel representation of the source code as well.

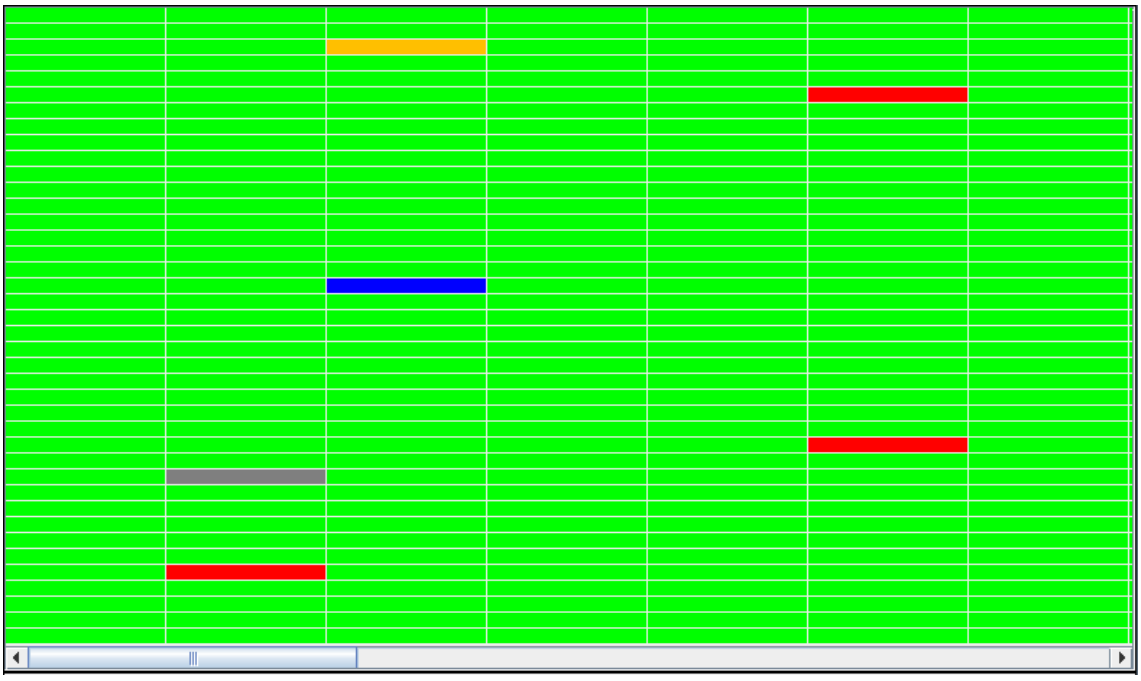

Figure 17 - File view

Figure 18 - File View Options

To allow not only the categorization but and the prioritization of the defects in the debugging process, the view can be easily filtered in the filter menu, shown in Figure 19.

Since the defect visualization tool is decoupled from the defect detection tool, it is possible to use any method for colouring the defects, as the colours are defined in the input file. Using more than one defect detection tool at the same, the only thing that the software developer must assure, is that the different methods and approaches used by the defect detection tools do not neutralize or disturb encountered defects of each others.

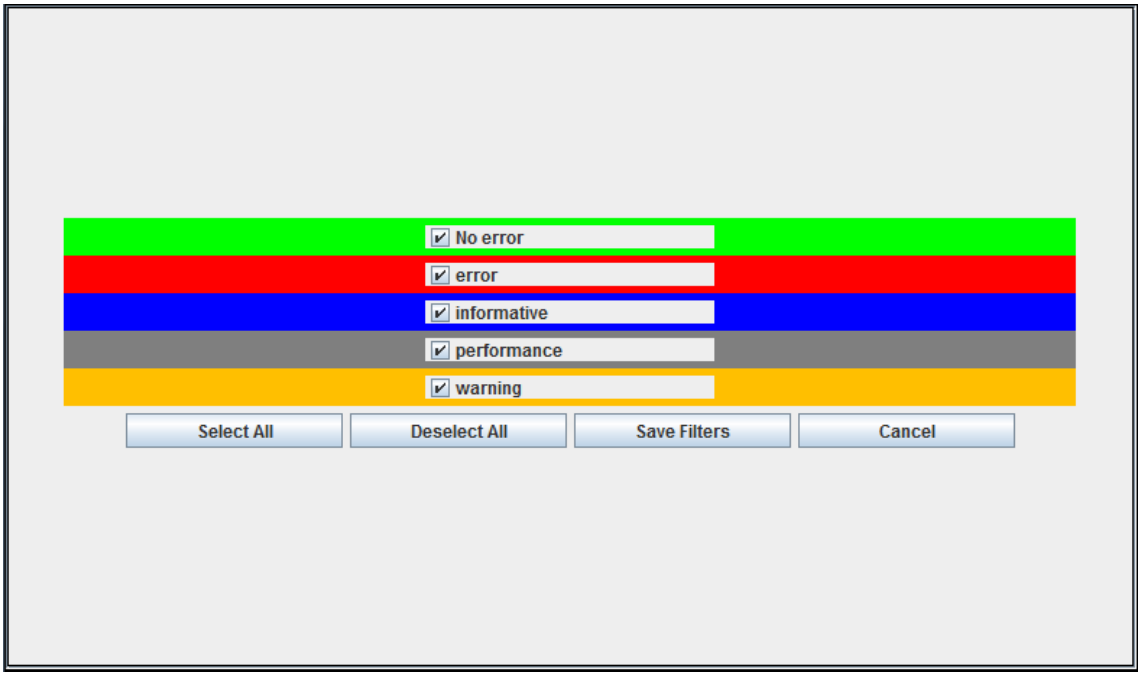

Figure 19 - Filter Options

This way it is possible for the software developer to choose what to see, when to see it, which is of most importance not to be overpowered with useless information during the most critical phase of the debugging process.

4.2.5 Defect View

The file view has a bottom panel that will show the information about the current defect. That was designed in order to allow the file view scalability to be increased. Putting the information in another panel, it is assured that the information will always be readable and easily accessed even when the view is scaled to the maximum possible.

The defect view is composed by the defect data and a tool to open the desired file in the line where the error was encountered, as shown by Figure 20.

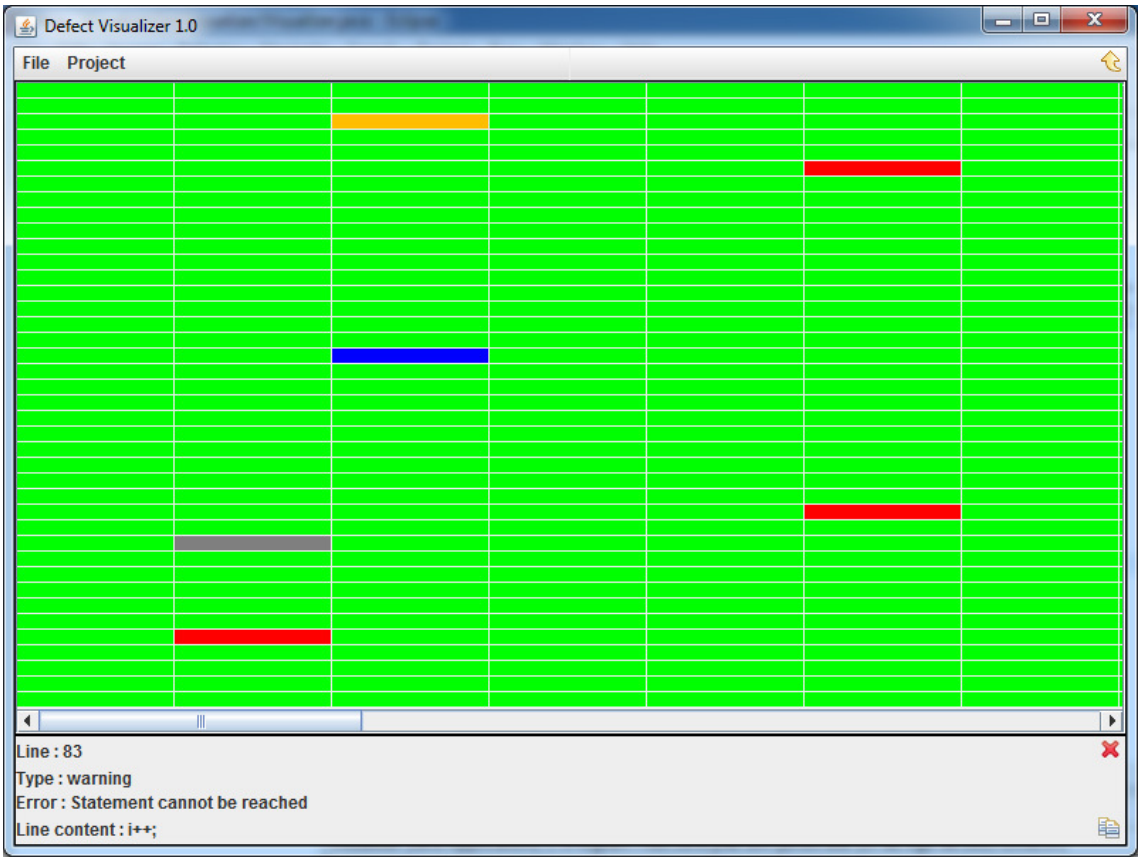

Figure 20 - Defect View

4.2.6 Statistics View

Until now hierarchical, line and pixel representations of the source code were presented in this paper, which leave us with the summary representation. In the case of defect visualization software, the summary should always present the size of the project and the number of encountered defects that it contains. For this project a statistics view was created to be shown in the bottom panel (until a defect is selected by an user and after the user closes the defect view), represented in Figure 21.

The statistic view was created with the same height as the defect view, so that the view does not lose eyesight of the necessary information when changing from the directory view to the file view or the other way around.

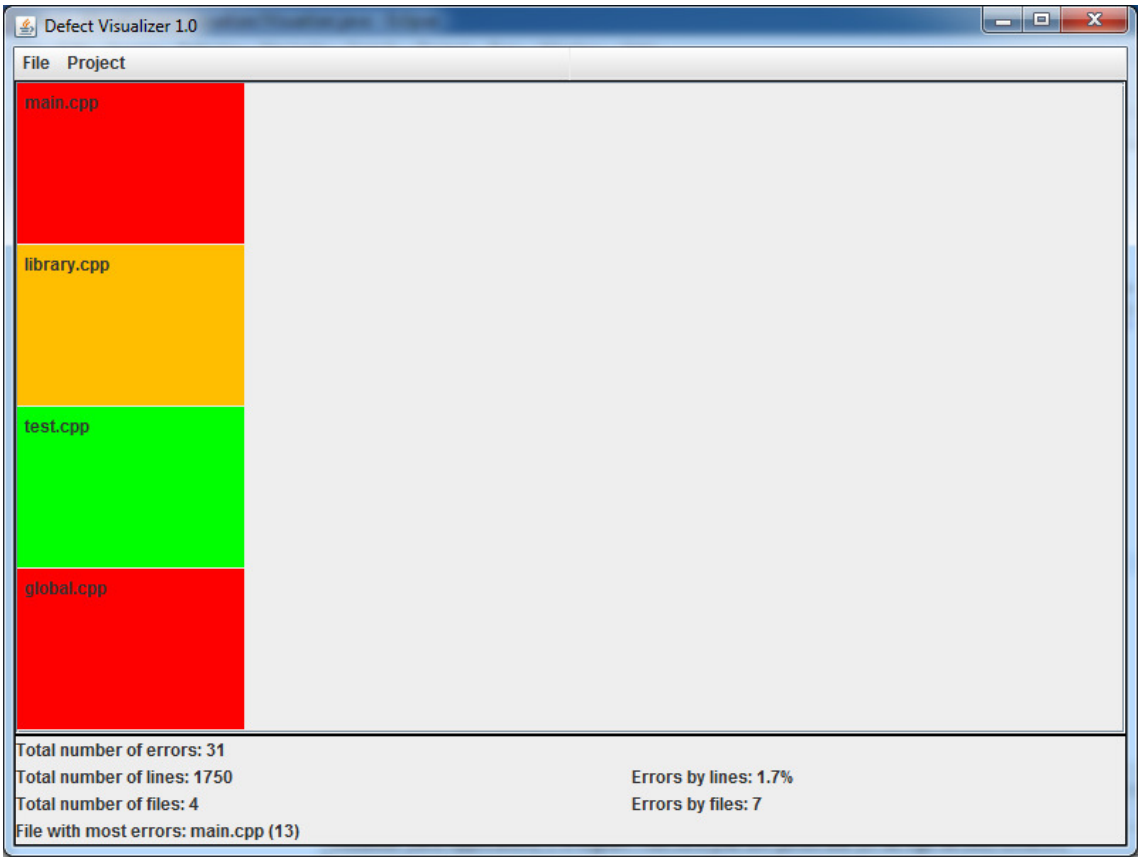

Figure 21 - Statistics View

4.2.7 Framework Example

Now that the reader understands how the views work and interact, an example of a specific fully integrated system will be presented. In this example all the steps to setup the framework and correct the first defect will be taken and explained in detail for fully understanding of the process.

For this example an executable file was prepared to execute the *Cppcheck* [40] tool and parse its output file to create the proper input file for the defect visualization tool. The File Editor used was the Notepad++ [42]. The final framework of this example is represented by Figure 22.

In this example only one defect detection tool is being used to make it of simple reading, but more defect detection tools could be used and then with a parser merge all the results into one file to serve as input for the defect visualization tool. In the future defect detection tools can have outputs that will be easily treated, but that does not happen in the present moment, so a parser is always needed for now.

For the defect visualization tool to work properly, the command lines options must be well defined. As it can be seen by Figure 23, the defect detection tool command is set for the created parser executable file and the edit file command set for Notepad++ with the proper nomenclature for file path and line number.

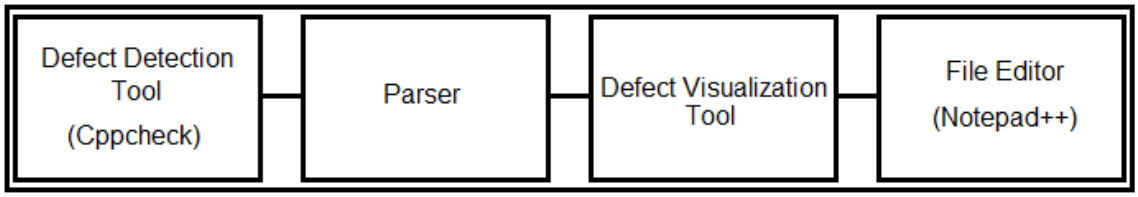

Figure 22 - Framework Example

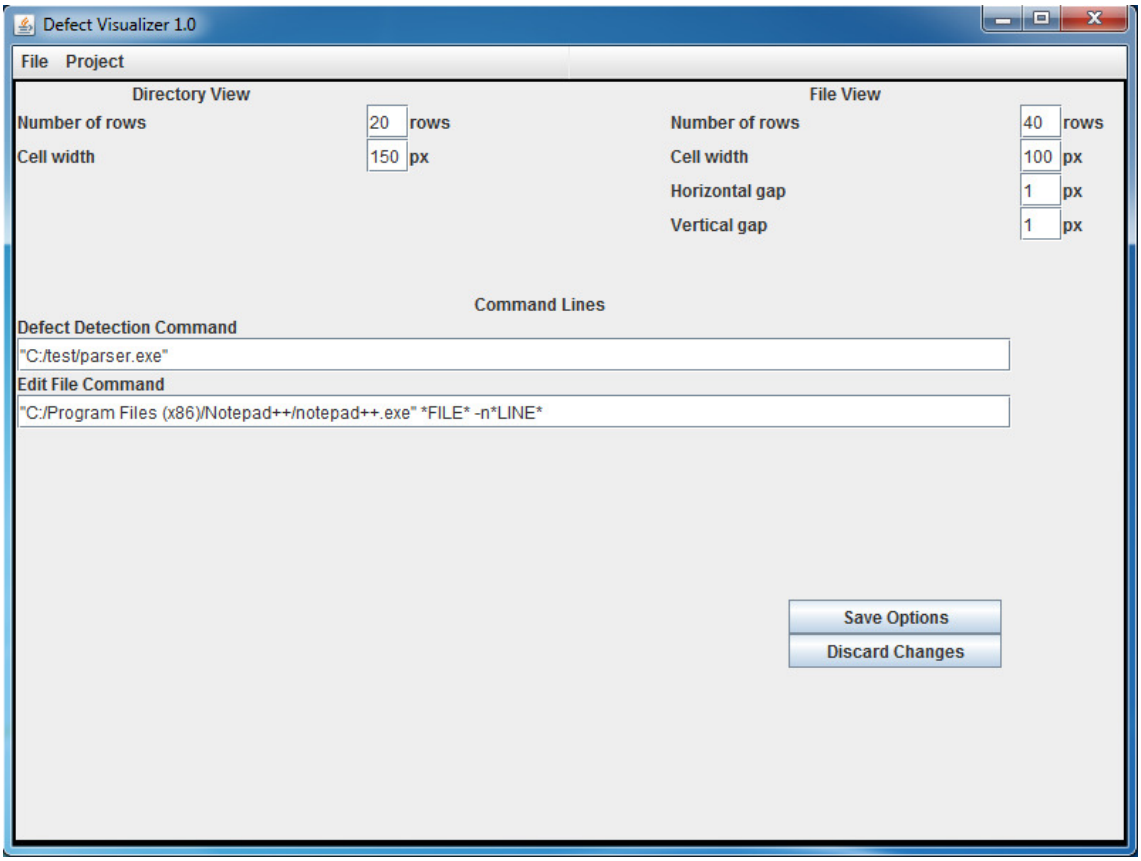

Figure 23 - Framework Example - Setting up options

The *FILE* and *LINE* values will be changed according to the defect view choice by the user, so that the file editor opens the file in the proper line of code. This is important as it saves the software developer the time of being looking for the line of code where the error occurred, as it will be possible to confirm later on.

To be able to visualize the output of the defect detection tool the user must refresh the view (this command can be seen in Figure 24 and it will always execute the defect detection command defined within the defect visualization tool options and update the views) and then open the desired file that should be defined within the open file menu, as shown in Figure 25 (that option is necessary only for the first refresh command, or when the user wants to change the input file, so that the defect visualization tool knows where the input file is).

Figure 24 - Framework Example - Creating input file

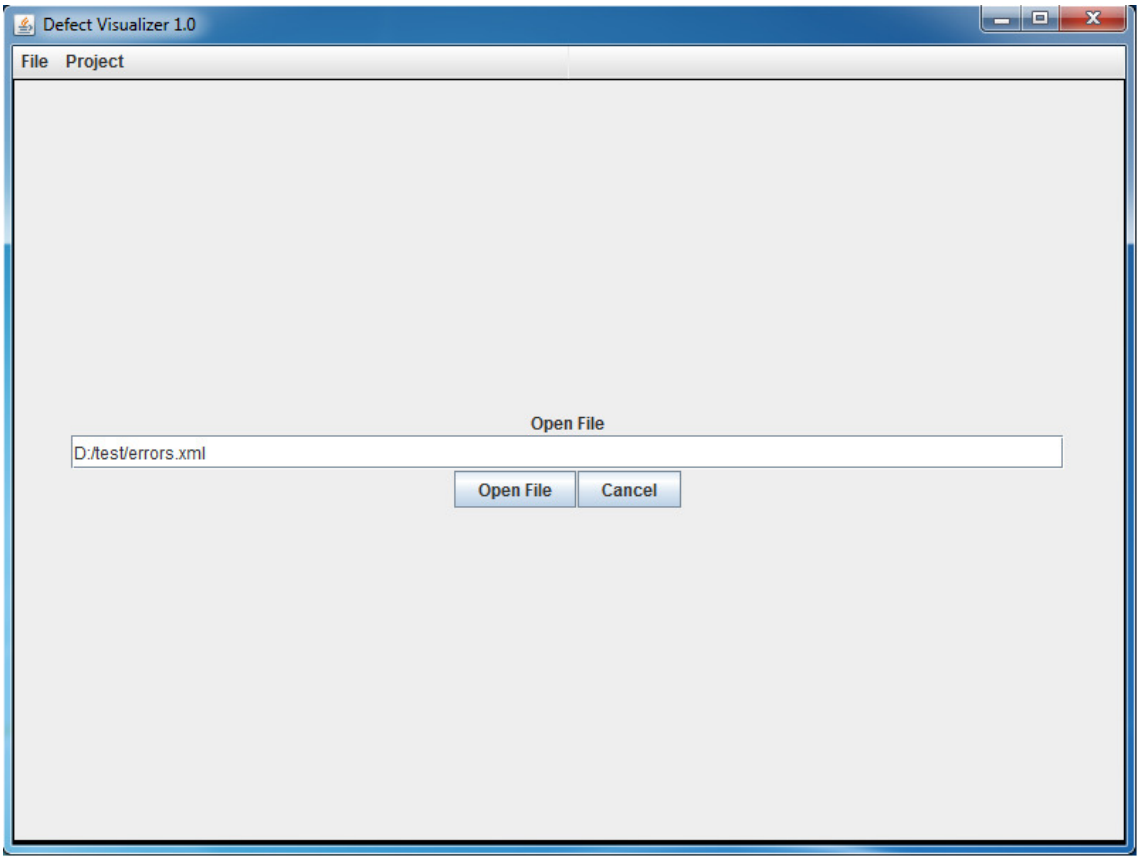

Figure 25 - Framework Example - Opening input file

As soon as all the information is saved, the defect visualization tool will allow the user to visualize the directory and the statistic views or the project. In this example the project contains four files, in two different folders, as its visualization can be seen in Figure 26.

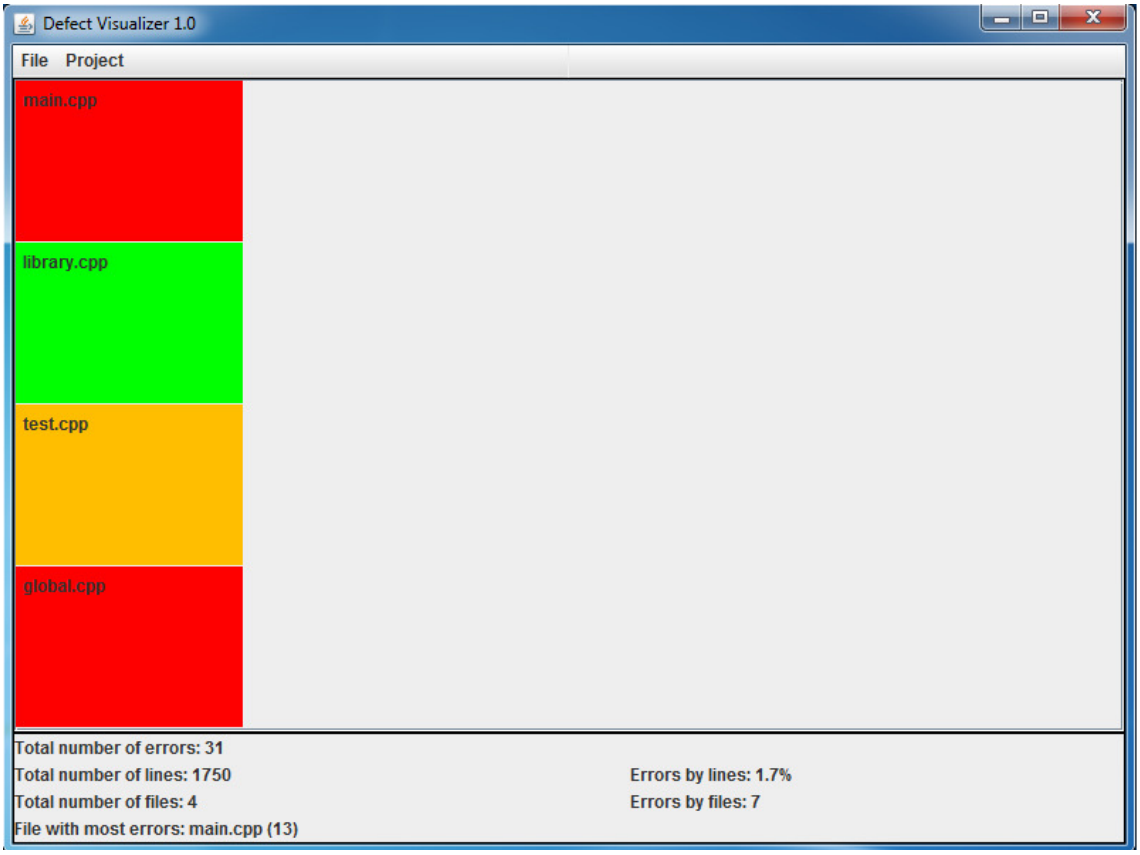

Figure 26 - Framework Example - Directory View

In this case the files are coloured by defect priority (main.cpp and global.cpp contain errors, test.cpp contain no errors but contain warnings and library.cpp contains no errors of any priority). This colouring was obtained by the analysis of the data within the output of *Cppcheck* [40] by the created parser as *Cppcheck* does not create priorities for its defects or files, only categories.

To solve the desired defect, the file global.cpp was opened, resulting in the file view represented by Figure 27. In this case the defects appear within the first 280 lines of code, so the view does not need to be scrolled or scaled (280 lines of code represent a view of 7 columns by 40 rows, which were created based on the defined options shown in the beginning of this section).

Though by prioritizing errors over warnings, one of the three presented errors in red should be dealt with first, in this example the warning will be solved first.

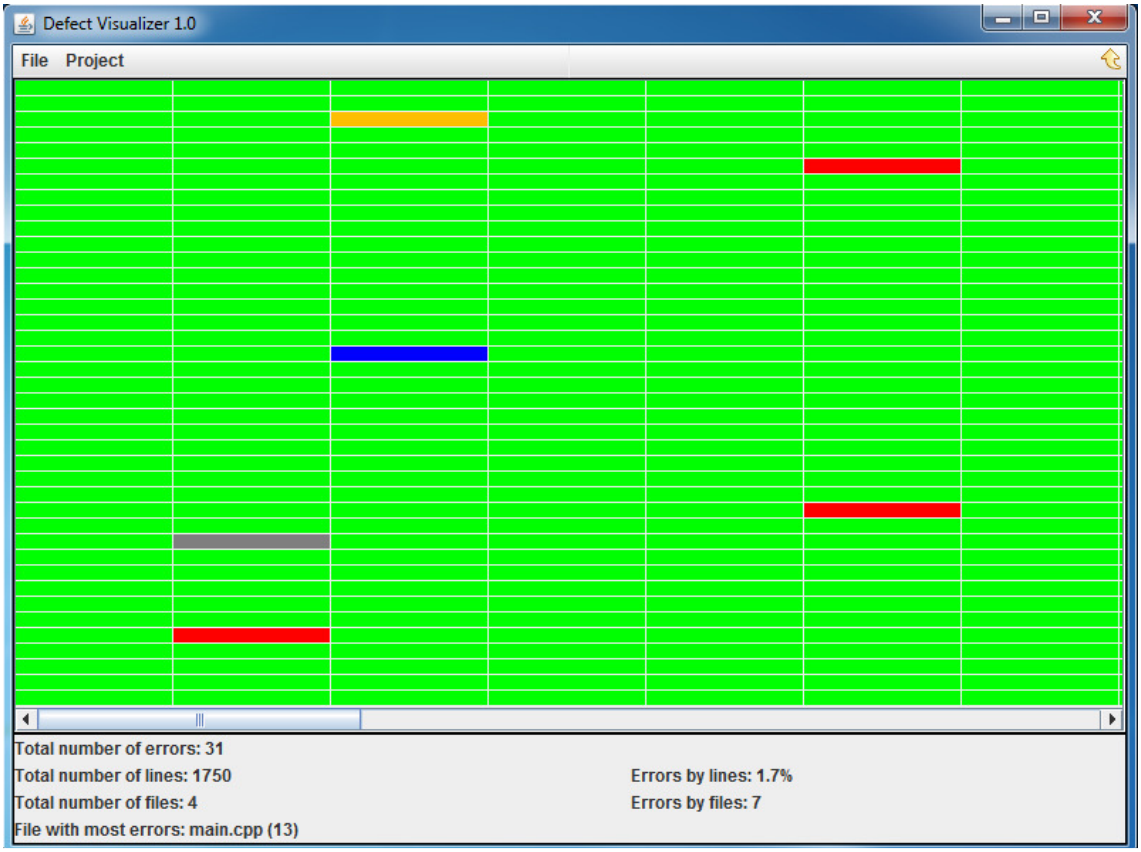

Figure 27 - Framework Example - File View

By clicking on the yellow box, representing the warning to solve, the bottom panel no longer shows the statistic view, rather changes its content to the defect view, as it shown by Figure 28.

The defect view will always show the line of the file in which the defect was encountered, the type of defect, the defect statement or explanation and the actual line of code in the source code file. This information is the basic information given by all the defect detection tools encountered during the literature review on this thesis project. As it was discussed before, it is the essential information for the software developer to quickly solve the defect .

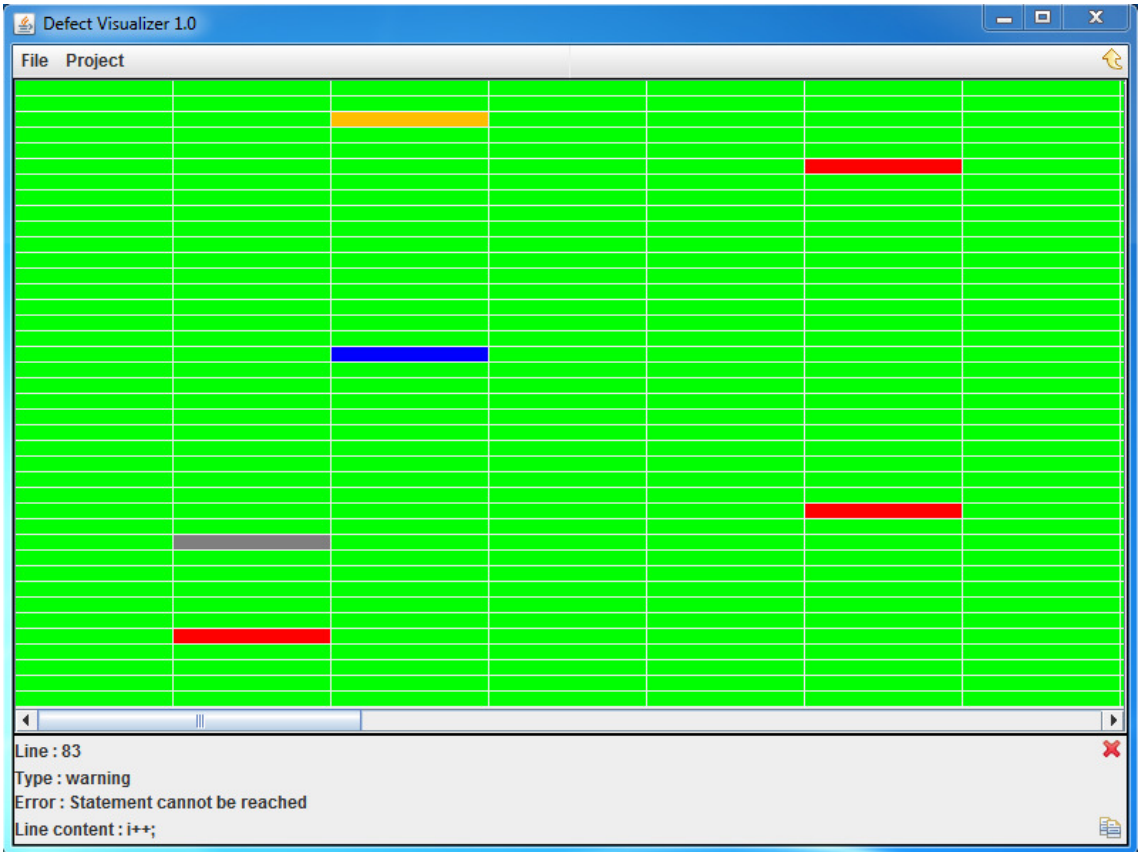

Figure 28 - Framework Example - Defect View

As it can be easily seen by Figure 28, the defect view has two buttons, one on the top, represented by a cross, is used to close the defect view and return to the statistic one, and another on the bottom, represented by a paper, which is used to open the file editor with the information provided by the defect and the file editor command in the options. When the second one is pressed, the result is, as presented in Figure 29, the opening of the source code file in the line where the defect was encountered so that the software developer can easily solve the defect.

This way the defect detection software not only communicates with the defect detection tool but also with the file editor, preventing the software developer to be lost within changes in the software and defect localization within the source code files, searching for the correct line of code.

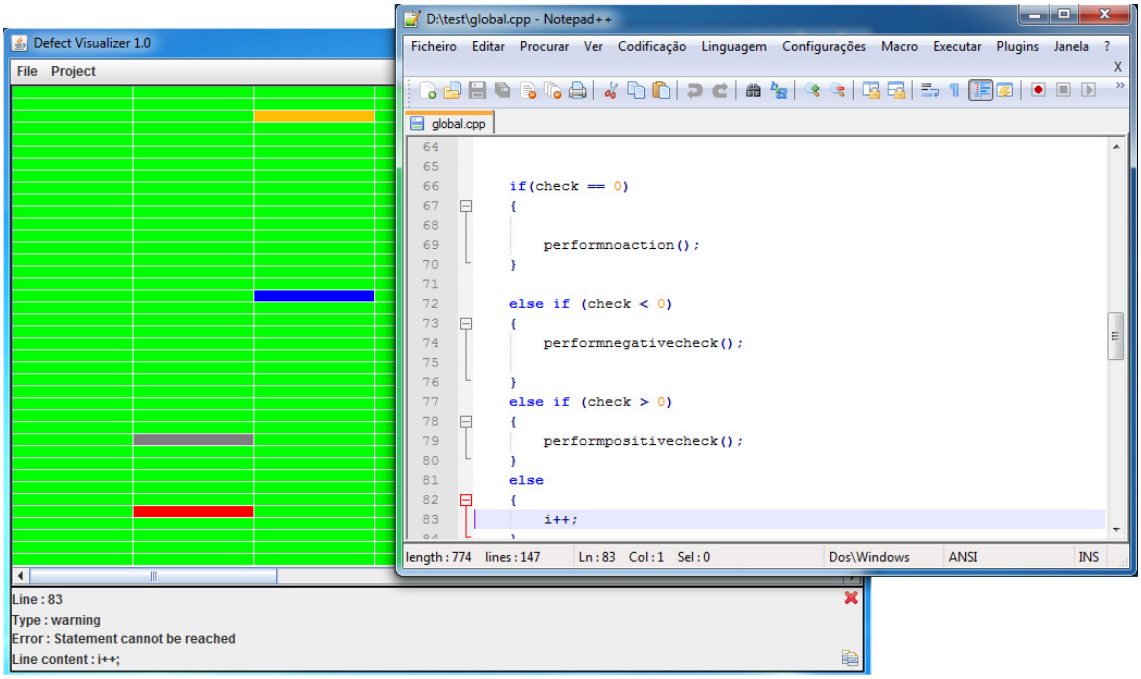

Figure 29 - Framework Example - File Editor

To continue to solve the rest of the encountered defect the software developer would only have to click in another defect and again on the edit file button until no defect is left to solve. At any time the software developer can update the visualization by clicking in the refresh menu, so that it can check if the previous defects were corrected or if new defects arose from the correction of the solved ones.

4.3 Results and Discussion

The most important thing in a visualization system, and one that it was the first priority of this visualization method, is scalability. The visualization method proved to be greatly scalable. It allows the user to scale up to one line of source code per pixel and on top of that it allows filtering within the view, which reduces the amount of information necessary to the visualization and thus also increases the scalability of the system.

Although in terms of scalability the method proved to be as good as possible for any size project, the fact is there is a few details that should be improved. The first one is a vertical scroll that appears when an user wants to create more rows than the existent in a view, the necessity of knowing the

actual height of the view to maximize its capability is not something that should be necessary. Another problem is that when viewing lines as pixels, clicking on the actual defect may become a harder task that it should be. A tooltip or a local zoom option within the mouse range should solve that problem.

The most important problem that needs a real and fast solution is when a line of code has more than one defect, as they can have different priorities and characteristics. A possible solution to that problem would be to create a "fake line", that meaning there would be more than one object in the visualization representing the same line but different defects.

The current solution is that the defect visualization tool should choose which defect to present, since the defect visualization tools knows the types of defects but not its priority level.

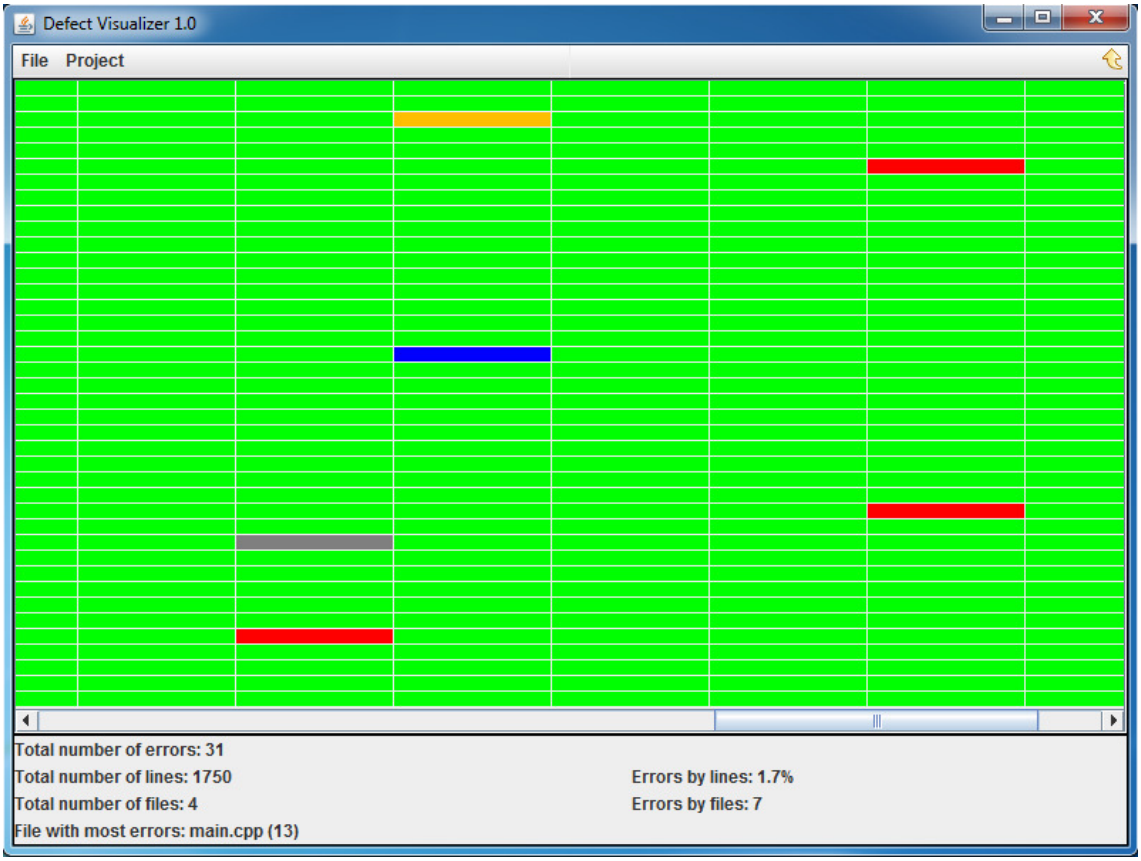

Figure 30 - File view with many lines and few errors

Regarding scalability there is still one last minor problem: in the directory view one must be careful with the length of the names in the files, as it can be as big as one wants to and that will make a file harder to find (a search option in the directory view is necessary and should be implemented).

Within a file view, if a file is great in number of lines and short in defects, finding one may become a task hard to complete in a few time. As it can be seen by Figures 30 and 31, in that situation the user will have to manually scroll the view until the desired location, scale down the view in order to view all the file at once or, if not possible, will to have to scroll the minimum required. The best option in this case is to resort to the filters, as it is shown in Figure 31, that will allow the user to see only the defects disregarding all lines of code which have none.

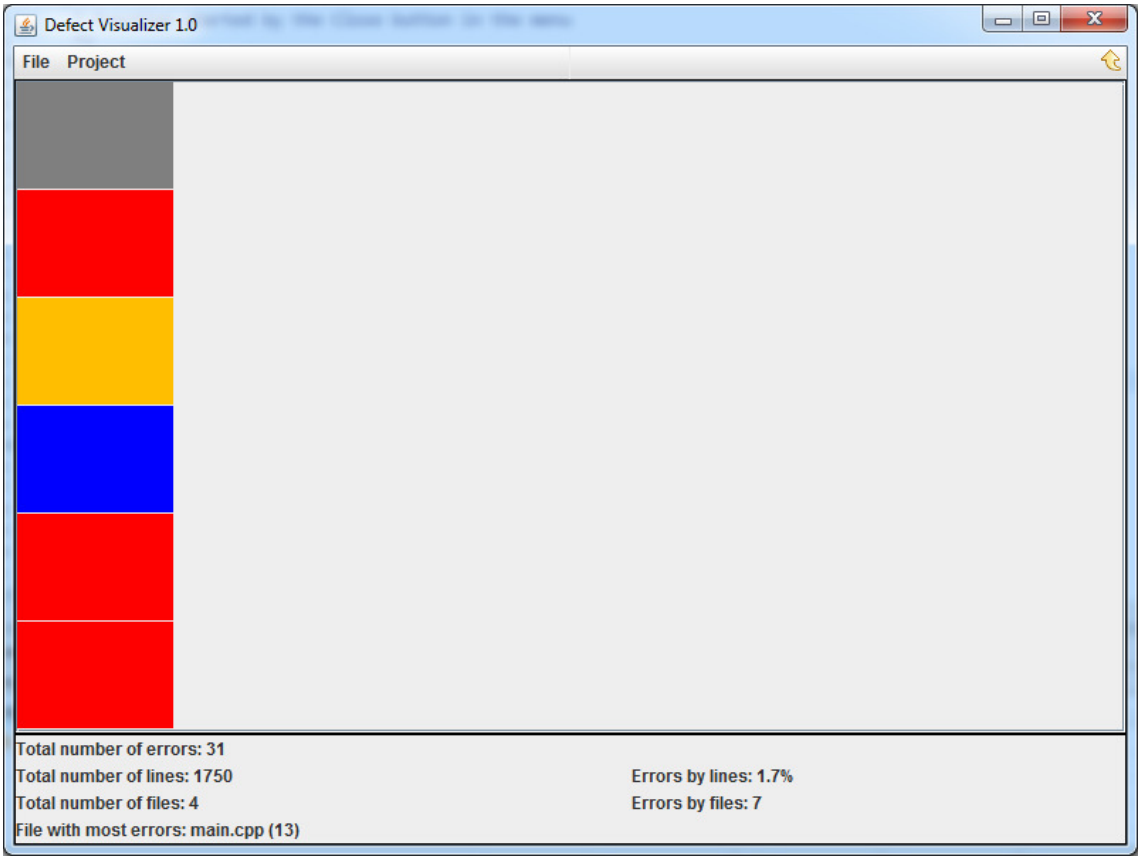

Figure 31 - File view with the use of filters

It is also possible that the user wants to find an specific defect among many others, and for that there must be a search option for the file view as well, for the software developer be able not only to find defect by type but also by content.

There is a good general abstraction used by this method and the visualization layers are well defined, but further analysis on the information presented itself may be necessary as defect detection tools are always using more and more types of information to find defects in source code and that information may need to be passed to the defect visualization tool.

The integration within the framework is key for the success of this type of software and the way this project does it is also remarkable, but its setup may be a too troublesome for small projects, unless the software developer uses the same defect detection many time, as there is no standard for defect representation and one will have to create a parser for each defect detection with which desires to use the visualization tool. There should be a major gain of time once it is done though.

This visualization system can be used in almost any platform and works with any tool that has a workable output and any compiler or file editor that has command line options, but the system has an handicap, as in software projects with more than one source code language, or with files with many different characteristics, the software developer may need the files to have different options depending on their localization, extension or any other characteristic and the system does not offer that.

5 Conclusion

Any defect visualization tool must aim to be able to be fully integrated in a framework and to use a method that ultimately will always save time to its user, as these two are the most common reasons why software developers do not use defect detection tools.

Though further work and analysis must be done, it is a good first step towards the definition of defect visualizing tools as a standalone software capable of working with more than one defect detection tool and a file editor at the same time, as it was seen in the last chapter.

5.1 Method Overview

As a defect visualization tool, scalability is the most important feature of its method, and as it is the method corresponds to a highly capable scaling visualization. It is not only possible to reduce its views components to a minimum without losing information or perception of the object of the visualization but also to filter the results in order to diminish the amount of unnecessary information to be visualized.

As an interactive process, this method is still not fully capable of being operated without concern, as it still fails to attend some important aspects necessary for a defect visualization method, like the fact of not addressing two errors to the same line of code.

That problem could be solved by approaching the view as *xSlice* [34] and instead of creating an object per line of code rather doing it by statement in the code, but that would turn the defect visualization tool not compatible with most of defect detection tools available. The creation of a "fake line" seems the best solution for that problem.

5.2 Product Analysis

As a prototype, the defect visualizing tool behaves well, though it still lacks functionalities like search and zoom options.

The language and communication within the framework chosen was good and there are numerous possibilities of usage of this software, as presented in the previous chapter. It is a simple software and easy to navigate and has the capacity of being used in most current system platforms.

Its categorization and prioritization system allows a quick decision in which defect to correct next and the options given, refresh of the defects (as soon as the defect detection ends its analysis) and a clear and simple usage of the file editor or compiler for its user to correct the defect.

The decision on how to use the visualization tool is always on the side of the defect detection tool, which is good in a way that will not restring its use, but at the same time if defect detection tools do not work towards a common use of output solutions, it can increase the cost of creating similar frameworks to the one in the example on the previous chapter.

One last problem with this defect visualization tool is that it doesn't allow two or more files or defects to be opened at the same time, so a software developer cannot compare two different defects or files if needed. Further analysis to the system is necessary and user feedback would also improve its quality.

5.3 Future Work

From a method perspective, it is necessary to address the problem of two defects in the same line of code. An extension to the phase of software maintenance rather than only be used for the debugging phase of the software development would also be desirable, as it would constitute a clear step towards a fully integrated method for defect visualization (and not only a method for software development).

For that to happen, further analysis and research would be necessary as the method would have to approach defects time life, status and status transitions, as the System Radiography and the Bug Watcher methods. The two systems cannot co-exist in the same visualization though, as the creation on an extra layer in not necessary for the visualization of defects in the development

phase of the project (and it would only result in more time consumption) it is absolutely essential for software maintenance.

As a defect visualization tool, it lacks functionality. As it was said above, file search, defect search and zoom options would be a major update to the system. Many other details can be improved and many of them will only be noticeable as its users need extra functionality, so user feedback should also be a priority. Another functionality that makes some projects incapable of using this tool is the options system, as it should be possible to change the options of only one file without changing for the entire project. In projects with source code with more than one language or with different characteristics within the files the software developer cannot be losing time is changing the options every time he wants to visualize another file. Files with different sizes or that need a different file editor can also create a similar problem.

At last, a software developer should be able to compare files and defects, which this tool does not allow him/her to do.

6 References

[1] Mantyla, M.V.; Lassenius, C. (2009), "What Types of Defects Are Really Discovered in Code Reviews?", *IEEE Transactions on Software Engineering*, pp.430-448.

[2] IEEE Computer Society (1990), "IEEE Standard Glossary of Software Engineering Terminology".

[3] Eagan, J.; Harrold, M.J.; Jones, J.A.; Stasko, J. (2001), "Technical note: visually encoding program test information to find faults in software", *IEEE Symposium on Information Visualization, INFOVIS 2001*, 22-23 October 2001, San Diego, California, USA, pp.33-36.

[4] D'Ambros, M. (2008), "Supporting software evolution analysis with historical dependencies and defect information", *IEEE International Conference on Software Maintenance, ICSM 2008*, 28 September - 4 October 2008, Beijing, China, pp.412-415.

[5] D'Ambros, M. (2007), ""A Bug's Life" Visualizing a Bug Database", *4th IEEE International Workshop on Visualizing Software for Understanding and Analysis, VISSOFT 2007*, 24-25 June 2007, Banff, Ontario, Canada, pp.113-120.

[6] Runeson, P.; Andersson, C.; Thelin, T.; Andrews, A.; Berling, T. (2006) "What do we know about defect detection methods?", *Software*, Vol.23, no.3, pp.82-90.

[7] Axelsson, S.; Baca, D.; Feldt, R.; Sidlauskas, D.; Kacan, D. (2009), "Detecting defects with an interactive code review tool based on visualisation and machine learning", *21st International Conference on Software Engineering and Knowledge Engineering, SEKE 2009*, 1-3 July 2009, Boston, Massachusetts, USA, pp. 412-417.

[8] Ball, T.; Eick, S.G. (1996), "Software visualization in the large", *Computer*, Vol.29, no.4, pp.33-43.

[9] Chaim, M.L. (2003), "A debugging strategy based on requirements of testing", *7th European Conference on Software Maintenance and Reengineering*, 26-28 March 2003, Benevento, Italy, pp.160-169.

[10] Jones, J.A.; Harrold, M.J.; Stasko, J. (2002), "Visualization of test information to assist fault localization", *24rd International Conference on Software Engineering, ICSE 2002*, 19-25 May 2002, Penang, Malaysia, pp.467-477.

[11] Yu, Y.; Jones, J.A.; Harrold, M.J. (2008), "An empirical study of the effects of test-suite reduction on fault localization", *ACM/IEEE 30th International Conference on Software Engineering, ICSE 2008*, 10-18 May 2008, Leipzig, Germany, pp.201-210.

[12] Jones, J.A.; Harrold M.J.; Stasko, J. (2001), "Visualization for Fault Localization", *23rd International Conference on Software Engineering, ICSE 2001*, 12-19 May 2001, Toronto, Ontario, Canada, pp.71-75.

[13] Agrawal, H.; Horgan, J.R.; London, S.; Wong, W.E. (1995), "Fault localization using execution slices and dataflow tests", *6th International Symposium on Software Reliability Engineering*, 24-27 October 1995, Toulouse, France, pp.143-151.

[14] Tam, K. (2011), "Debugging Debugging", *IEEE 35th Annual Computer Software and Applications Conference Workshops, COMPSACW 2011*, 18-22 July 2011, Munich, Germany, pp.512-515.

[15] Kleidermacher, D.N. (2008), "Integrating Static Analysis into a Secure Software Development Process", *IEEE Conference on Technologies for Homeland Security*, 12-13 May 2008, Waltham, Massachusetts, USA, pp.367-371.

[16] Sfayhi, A.; Sahraoui, H. (2011), "What You See is What You Asked for: An Effort-Based Transformation of Code Analysis Tasks into Interactive Visualization Scenarios", *11th IEEE International Working Conference on*

Source Code Analysis and Manipulation, SCAM 2011, 25-26 September 2011, Williamsburg, Virginia, USA, pp.195-203.

[17] Wong, W.E.; Debroy, V.; Golden, R.; Xiaofeng X.; Thuraisingham, B. (2012), "Effective Software Fault Localization Using an RBF Neural Network", *IEEE Transactions on Reliability*, Vol.61, Issue 1, pp.149-169.

[18] Hao, D.; Zhang, L.; Mei, H.; Sun, J. (2006), "Towards Interactive Fault Localization Using Test Information", *13th Asia Pacific Software Engineering Conference, APSEC 2006*, 6-8 December 2006, Bangalore, India, pp.277-284.

[19] Gupta, R.; Soffa, M.L. (1994), "A framework for partial data flow analysis*", International Conference on Software Maintenance, ICSM 2004*, 19-23 September 1994, Victoria, British Columbia, Canada, pp.4-13.

[20] Gonzalez-Sanchez, A.; Piel, E.; Gross, H.-G.; van Gemund, A.J.C. (2010), "Prioritizing Tests for Software Fault Localization", *10th International Conference on Quality Software, QSIC 2010*, 14-15 July 2010, Zhangjiajie, China, pp.42-51.

[21] Hayashi, S.; Asakura, T.; Zhang, S. (2002), "Study of machine fault diagnosis system using neural networks", *International Joint Conference onNeural Networks, IJCNN 2002*, Vol.1, 12-17 May 2002, Honolulu, Hawaii, USA, pp.956-961.

[22] Li, P. (2010), "A comparative study on soft1ware vulnerability static analysis techniques and tools", *IEEE International Conference on Information Theory and Information Security, ICITIS 2010*, 17-19 December 2010, Beijing, China, pp.521-524.

[23] Langelier, G.; Sahraoui, H.; Poulin, P. (2005), "Visualization-based analysis of quality for large-scale software systems", *20th IEEE/ACM International Conference on Automated Software Engineering, ASE 2005*, 07-11 November 2005, Long Beach, California, USA, pp.214-223.

[24] Kitchenham, B.; Pfleeger, S.L. (1996), "Software quality: the elusive target", *Software*, Vol.13, no.1, pp.12-21.

[25] Weerahandi, S.; Hausman, R.E. (1994), "Software quality measurement based on fault-detection data", *IEEE Transactions on Software Engineering*, Vol.20, Issue 9, pp.665-676.

[26] Elsevier B.V (2012), *Scopus - Document Search*, available at: http://www.scopus.com/ (accessed April-May 2012).

[27] IEEE (2012), *IEEE Xplore*, available at: http://ieeexplore.ieee.org/ (accessed April-May 2012).

[28] Google (2012), *Google Scholar*, available at: http://scholar.google.com/ (accessed April-May 2012).

[29] Yeh, P.-L.; Lin, J.-C. (1998), "Software testability measurements derived from data flow analysis", *2nd Euromicro Conference on Software Maintenance and Reengineering*, 8-11 March 1998, Florence, Italy, pp.96-102.

[30] Kienle, H.M.; Muller, H.A. (2007), "Requirements of Software Visualization Tools: A Literature Survey", *4th IEEE International Workshop on Visualizing Software for Understanding and Analysis, VISSOFT 2007*, 24-25 June 2007, Banff, Ontario, Canada, pp.2-9.

[31] The Eclipse Foundation (2012), *Eclipse - The Eclipse Foundation open source community website*, available at: http://www.eclipse.org/ (accessed June 2012).

[32] Microsoft Corporation (2012), *Microsoft Visual Studio,* available at: http://www.microsoft.com/visualstudio/en-us (accessed June 2012).

[33] Georgia Tech College of Computing (2005-2011), *Tarantula - Fault Localization via Visualization*, available at: http://pleuma.cc.gatech.edu/aristotle/Tools/tarantula/index.html (accessed June 2012).

[34] Cleanscape Software International (2006-2011), *Cleanscape Software Visualization and Analysis Toolset: xSlice dynamic debugging tool*, available at:

http://legacy.cleanscape.net/products/testwise/tools_xslice.html (accessed June 2012).

[35] Cleanscape Software International (2006-2011), *Cleanscape C++ Lint*, available at: http://legacy.cleanscape.net/products/cpp/index.html (accessed June 2012).

[36] Cleanscape Software International (2006-2011), *Cleanscape LintPlus C Source Code Analysis Tool*, available at: http://legacy.cleanscape.net/products/lintplus/index.html (accessed June 2012).

[37] Cleanscape Software International (2006-2011), *Cleanscape FortranLint Fortran source code analysis tool*, available at: http://legacy.cleanscape.net/products/fortranlint/index.html (accessed June 2012).

[38] Coverity (2012), *Coverity Scan Site: Accelerating Open Source Software Integrity*, available at: http://scan.coverity.com/ (accessed June 2012).

[39] UVSQ and INRIA (2010), *FADA toolkit*, available at: http://www.prism.uvsq.fr/~bem/fadalib/index.html (accessed June 2012).

[40] Geeknet, Inc. (2012), *Cppcheck - A tool for static C/C++ code analysis*, available at: http://cppcheck.sourceforge.net/ (accessed June 2012).

[41] Eick, S.C.; Steffen, J.L.; Sumner Jr., E.E. (1992), "Seesoft - A tool for visualizing line oriented software statistics", *IEEE Transactions on Software Engineering*, Vol.18, Issue 11, pp.957-968.

[42] Don Ho (2011), *Notepad++*, available at: http://notepad-plus-plus.org/ (accessed August 2012).

Appendix A

A.1 XML Input File Schema

```
<?xml version="1.0" encoding="UTF-8" ?>

<errors>

        <type id="" r="" g="" b="" />

        <type id="" r="" g="" b="" />

        <file name="" path="" lines="" r="" g="" b="">

                <error line="" type="" msg="" content="" />

                <error line="" type="" msg="" content="" />

        </file>

        <file name="" path="" lines="" r="" g="" b="">

        </file>

</errors>
```

A.2 Defects

A defect is always defined within a file and has a type, defined in the top on the XML file. Besides its type, a defect is composed by a line that represents the line in the source code file where the error was encountered, a message that represents the error itself and a content representing the actual content of the line of code in the file.

A.3 Files

A file can have none or more defects. It has a path that represents the actual path for the file, a name representing the name of the file, the number of lines which the file has and a colour, represented by its red, green and blue components.

A.4 Types

All defects must have a valid type, composed by an id, which represents its name, and a colour, represented by its red, green and blue components. The type with id="noerror" defines the colour of the files which have no defect at all.

A.5 Example

```
<?xml version="1.0" encoding="UTF-8" ?>

<errors>

    <type id="noerror" r="0" g="255" b="0" />

    <type id="error" r="255" g="0" b="0" />

    <file name="main.c" path="C:\test\main.c" lines="518" r="255" g="0" b="0">

        <error line="82" type="error" msg="The local variable i may not have been initialized" content="if ( i > 0 )" />

    </file>

    <file name="library.h" path=" C:\test\ library.h" lines="423" r="0" g="255" b="0">

    </file>

</errors>
```