Code Inspection Supported by Stepwise Abstraction and Visualization
An Experimental Study

Anderson Belgamo[1,2], Elis Montoro Hernandes[1,3], Augusto Zamboni[1], Rafael Rovina[3] and Sandra Fabbri[1]
[1]*LaPES - Software Engineering Research Lab, Federal University of São Carlos, UFSCar, São Carlos, SP, Brazil*
[2]*IFSP - São Paulo Federal Institute of Education, Science and Technology, Piracicaba, SP, Brazil*
[3]*IFSP - São Paulo Federal Institute of Education, Science and Technology, São Carlos, SP, Brazil*

Keywords: Code Reading, Reading Technique, Software Inspection, Experiments, Empirical Software Engineering, Experimental Software Engineering.

Abstract: Background: In order to inspect source code effectively and efficiently, in a previous work the use of visualization for supporting the reading technique Stepwise Abstraction was proposed and implemented in the CRISTA tool. Visualization aids code comprehension, which is an essential task for a successful inspection. Goal: The objective of this paper is to evaluate the effectiveness and efficiency of using stepwise abstraction supported by visualization for defects detection, in comparison to an ad-hoc approach. Method: A controlled experiment was conducted with two groups of undergraduate students. One group inspected the Java source code of the Paint software using the approach implemented in CRISTA and the other group inspected the code using an ad-hoc approach. Results: The general performance of the subjects who used Stepwise Abstraction supported by visualization was better than that of the subjects who used the ad-hoc approach. Besides, the subjects' experience in inspection and Java did not influence the identification of defects. Conclusion: the results reveal that the use of Stepwise Abstraction and visualization promotes better performance in detecting defects than the ad-hoc approach. In future work, other approaches are being investigated as well as the support of the approaches for different types of defects.

1 INTRODUCTION

The software inspection process was created in 1972 by Fagan, for IBM, with the objective of improving software quality and increasing programmers' productivity. It is a static analysis method used for verifying whether products generated during the software development process satisfy users (Fagan, 1976, 1986).

The premise of inspection is that as soon as defects are detected, less time is spent in reworking, ensuring that the software can answer the user's requirements and be delivered on time and in accordance with the budget.

The inspection process can be used for inspecting every kind of artifact, for example, requirements, documentation, and test case plans, with the objective of improving the final product quality. Inspection does not replace testing; they are processes that must be combined (Russel, 1991; Elberzhager et al., 2012).

In addition, inspection is a way of improving the software maintainability that allows the detection of types of defects that are not detectable by other techniques (Siy and Votta, 2001). Examples are the evolvability defects, which are related to functional defects that would never be detected if inspection were not applied (Mantyla and Lassenius, 2009).

Aiming to evaluate the software inspection activity, some experimental studies have been executed and the artifacts most frequently inspected are the requirement document (Basili et al., 1996a; Travassos et al., 2002; Marucci et al., 2002; Belgamo et al., 2005) and source code (Dunsmore et al., 2003; Laitenberger and DeBaud, 1997; Kelly and Sheppard, 2000, 2002; Almeida et al., 2003).

In general, the inspection activity is supported by

Belgamo A., Hernandes E., Zamboni A., Rovina R. and Fabbri S..
Code Inspection Supported by Stepwise Abstraction and Visualization - An Experimental Study.
DOI: 10.5220/0004886000390048
In *Proceedings of the 16th International Conference on Enterprise Information Systems* (ICEIS-2014), pages 39-48
ISBN: 978-989-758-028-4
Copyright © 2014 SCITEPRESS (Science and Technology Publications, Lda.)

reading techniques, which aim to facilitate the task and guide inspectors in finding defects. Particularly, taking into account the source code, there are studies highlighting the need for a systematic way to comprehend it such that all type of defects can be found. Besides, the systematic comprehension of code can be aided by visualization techniques, which were classified by Caserta and Zendra (2011).

The objectives of this paper are (a) to present an inspection approach implemented through the reading technique Stepwise Abstraction with the support of visualization and (b) to present the results of an experimental study that evaluated the effectiveness and efficiency for defect identification in source code, comparing the use of Stepwise Abstraction supported by visualization, implemented in the CRISTA (*Code Reading Implemented with STepwise Abstraction*) tool (Porto et al., 2009b), with the ad-hoc approach.

The results of the experiment give insights that Stepwise Abstraction supported by visualization helps the inspector to identify defects and makes the activity shorter compared with the ad-hoc approach.

This paper is organized as follows: Section 2 presents concepts related to code inspection supported by visualization and Stepwise Abstraction. Section 3 presents the experiment carried out and the results obtained. Finally, Section 4 presents our conclusions and future work.

2 CODE INSPECTION SUPPORTED BY STEPWISE ABSTRACTION AND VISUALIZATION

Code comprehension is the starting point of code inspection. In relation to code comprehension, a usual approach for systematically comprehending code is to analyze its structure and to construct a high level representation for it. Some techniques attempt to standardize abstractions derived from program comments and variable names. Other methods attempt to understand the program by applying transformation rules for deriving abstract concepts which represent parts of the code (Vinz and Etzlorn, 2006). However, these approaches are not used for code inspection, which is conducted through reading techniques.

Aiming to identify the reading techniques used for code inspection, a systematic mapping was conducted and it was found that the following techniques were used: ad-hoc (Aurum et al., 2002),checklist (Gilb and Graham, 1993; Humphrey, 1989; Laitenberger and DeBaud, 1997; Dunsmore et al., 2003), Stepwise Abstraction (Linger et al., 1979; McMeekin et al., 2009), use case (Dunsmore, 2003), Perspective-Based Reading, PBR (Basili et al., 1996a; Basili et al., 1996b; Basili et al., 1998; Laitenberger and DeBaud, 1997), Task-Directed Inspection, TDI (Kelly and Sheppard, 2000; Kelly and Sheppard, 2002), and the comparison-based approach (Li, 1995).

The Stepwise Abstraction technique (Linger et al., 1979) helps to comprehend the program functionality through the functional abstraction generated from the source code. The inspector must read the code from the internal to the external structures and write a specification for the software according to his or her comprehension.

As the inspector should read and abstract all parts of the code, he or she obtains a good knowledge of the whole code. On one hand this is a positive point, but on the other hand this technique requires great effort and time, since even the simplest parts of code, as a variable increment, must be abstracted. Hence, the use of an ad-hoc technique (nonsystematic) can be most productive.

Hence, Porto et al. (2009b) developed the CRISTA tool to help in code comprehension and code inspection, adopting the Stepwise Abstraction technique supported by visualization. According to Knight and Munro (1999, 2001) and Mayrhauser (1998), visualization aims to acquire enough knowledge about software through the comprehension of the artifacts produced along the software development process and the relationship among them.

Code visualization is a kind of software visualization that is frequently used for comprehension purposes. In the context of this work code visualization is used for comprehending the code and also to support the application of the Stepwise Abstraction technique.

The next section presents the CRISTA tool and the visualization support implemented with the treemap technique.

2.1 The CRISTA Tool

The motivation for implementing the tool CRISTA was to support code inspection through the reading technique Stepwise Abstraction and the visual metaphor named treemap (Johnson and Shneiderman, 1991). It provides support for the whole process (reading the code, registering discrepancies, joining discrepancy lists, etc.), unlike other tools that just support some activities.

The visual metaphor treemap visually represents the hierarchical blocks of the code, offering a simple way to look at the code structure. Code blocks are separated and represented as nested rectangles, according to the code hierarchy. Initially, the rectangles start in red to indicate that no block was abstracted. In addition, by changing the rectangle colors from red to green, the reader obtains a visual feedback of the code analysis process according to the evolution of the blocks abstraction.

Figure 1 presents the main screen of the tool, which has three main areas: 1) the visual metaphor that corresponds to the hierarchical structure of the code being analyzed; 2) the code being analyzed, which is logically linked to the visual metaphor; any rectangle clicked on the metaphor highlights the corresponding source code and vice-versa; and 3) the documentation area, where the reader can enter a comment, as a free text, explaining what the selected code block executes. This comment can have a later use to produce code documentation or pseudo-code, since it will be physically associated to the selected block. Once a comment has been inserted, the rectangle corresponding to this code block changes from red to green. Thus, the reader can easily follow the progress of the code abstraction process.

CRISTA provides two options for abstracting code: (i) following Stepwise Abstraction strictly, in which the inspector can document an outer block only if all the internal blocks were already documented; and (ii) disabling the use of Stepwise Abstraction, allowing any code block can be abstracted without any constraint.

Besides, considering the diversity of program languages, CRISTA was designed to be easily instantiated for different languages (Porto et al., 2009b). Currently, it accepts Java, C, C++, and Cobol85 codes.

Some experimental studies were already carried out to evaluate the use of visualization and the support of CRISTA for conducting coding inspection. In these studies (Porto et al., 2009a), CRISTA was evaluated with a group of students (41 students in total). The studies indicate that the tool is easy to use and can systematize code comprehension and documentation.

Currently there are some tools for inspection, particularly for supporting the software inspection process, as mentioned by Hernandes et al. (2013).

For code inspection the following tools were identified by the authors through a systematic literature mapping: Team Tracks (DeLine et al., 2005), ReviewClipse – RC (Bernhart et al., 2010), SCRUB (Holzmann, 2009), ICICLE (Brothers et al., 1990), Codestriker and ReviewPro (Remillard, 2005).

Although these tools can be used for code inspection, they do not use visualization and they accept only one programming language: C or Java. They do not allow legacy code as Cobol programs to be visualized through a visual metaphor.

3 EXPERIMENTAL STUDY

The objective of the experimental study was to evaluate the effectiveness and efficiency for detecting source code defects through two approaches: Stepwise Abstraction supported by visualization and Ad-hoc.

The Ad-hoc technique was selected for the control group for two reasons: firstly, the authors intend to identify code inspection patterns by evaluating the way the participants have conducted the inspection, since their actions were recorded. Secondly, in general, checklists are adjusted for a specific environment, such that they can abstract the characteristics of the team for producing better results.

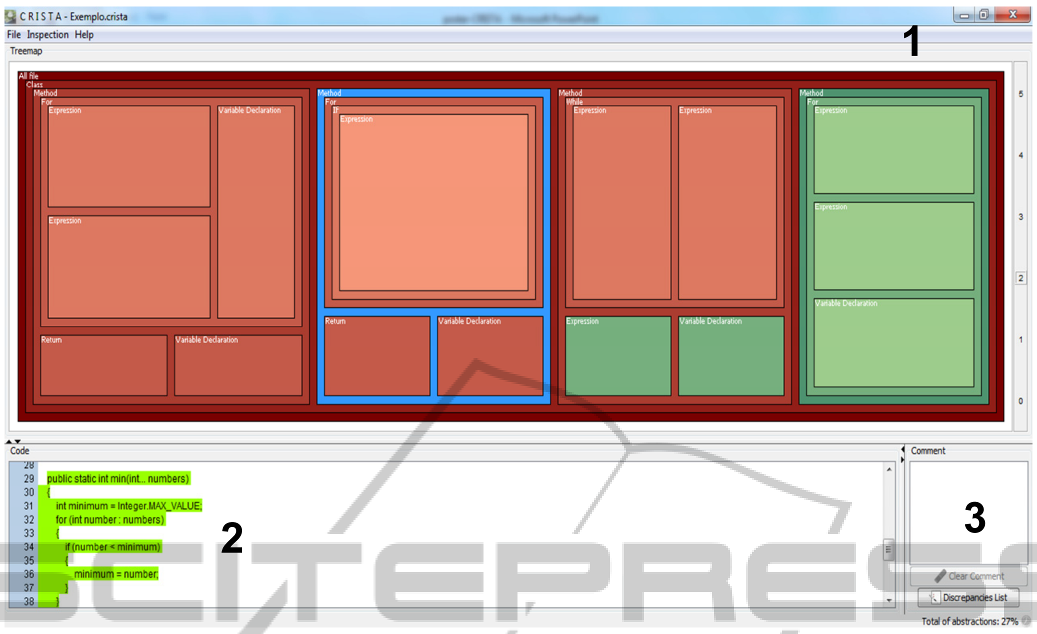
The participants inspected an object-oriented application with four known defects. The defect descriptions were previously disclosed for the participants to them so that they could mark the necessary time to identify each of the defects.

Below, the sections describe the main topics of the experiment, according to Wohlin et al. (2000).

3.1 Experiment Definition

The goal of the experiment was defined as shown in the following template:

Analyze	The use of Stepwise Abstraction supported by visualization
for the purpose of	evaluation
with respect to	effectiveness and efficiency
from the point of view of the	researchers of the code inspection area
in the context of	students of the Bachelor of Computing Science and Computing Engineering courses of the Federal University of São Carlos



```
public static int min(int... numbers)
{
   int minimum = Integer.MAX_VALUE;
   for (int number : numbers)
   {
      if (number < minimum)
      {
         minimum = number;
      }
   }
```

Figure 1: CRISTA tool.

3.2 Context Selection

The experiment was a task during a Software Engineering module in the context of the Bachelor of Computing Science and Computing Engineering courses at the University of São Carlos. The participants received a grade only for their participation and not for their performance. They were aware of collaboration in the research and were willing to participate

The context of the experiment can be characterized according to four dimensions (Wohlin et al., 2000): i) off-line: the software was not developed by industry; ii) student: the subjects were undergraduate students; iii) toy: the problem to be solved was not a real problem, although the selected application has all the features of the object-oriented paradigm; iv) specific: the results cannot be generalized to other contexts.

3.3 Variables Selection

For this experiment we considered the independent variable, called "UsedTechnique", which represents the use or non-use of the reading technique Stepwise Abstraction supported by visualization and implemented in the CRISTA tool (Porto et al., 2009b). This independent variable has two treatments which characterize the way the code inspection activity was done:

- StepVis: represents the use of Stepwise Abstraction and visualization, that is, the use of the CRISTA tool.
- Ad-Hoc: represents the use of an ad-hoc approach.

Besides, there were also the following independent variables:

- Experience in Java: this variable was used to investigate whether experience in Java language would have any influence on the results.
- Experience in code inspection: this variable was used to investigate whether experience in code inspection would have an influence on the results.

Two dependent variables were considered:

- Effectiveness: defined as the amount of real defects identified by the code inspector. It was measured as the sum of discrepancies that were correspondent to the list of four known defects;
- Efficiency: defined as the time spent by the inspector. It was measured as "the number of real defects identified/time".

It is important to notice that the inspector can identify discrepancies which cannot be classified as "real defects". These discrepancies are called false positives and are not used to calculate the effectiveness and efficiency variables.

3.4 Selection of Participants

The selection of participants was based on convenience since the Software Engineering program was offered by one of the authors. The topic "Software Inspection" was part of this program and most of the students (experiment participants) had no previous knowledge of this topic.

3.5 Experimental Design

The experimental design was based on the independent variable "UsedTechnique" such that two reading techniques were compared: StepVis and Ad-Hoc. Each subject used one of the techniques and the subjects were divided into two groups (G1, which used the StepVis alternative, and G2, which used the Ad-Hoc alternative), consisting of 30 and 29 subjects, respectively. It is important to notice that the participants were divided into two groups according the characteristics collected by the profile questionnaire.

3.6 Instrumentation

The materials used during the experiment execution were the consent form, defect and time report, feedback report, training material, source code of the Paint software, and CRISTA tool to support the Stepwise Abstraction technique and visualization.

The Paint software is a simple figure editor written in Java language. It allows users to draw, erase, clear, and undo colored strokes (using RGB pattern) on a white canvas. This software has been used in other studies (Ko et al., 2006; Robbilard et al., 2004).

There is no documentation about Paint and the code has no comment. Paint was implemented with nine Java classes through nine source files that have an average of 73 lines of code. The four defects reported were not artificial, but emerged during the creation of Paint (Ko et al., 2006):

- Defect 1: Users cannot select yellow color.
- Defect 2: The button "undo my last stroke" does not work properly.
- Defect 3: The button "undo my last stroke" is enabled without any action being taken.
- Defect 4: There is an option to draw a line but it does not work.

3.7 Hypotheses Formulation

The following hypotheses were formulated:

- *Hypothesis 1*:

$H_{0,1}$: There is no difference in the number of defects (*effectiveness*) identified when using the StepVis or the Ad-Hoc technique.

$H_{1,1}$: There is a difference in the number of defects (*effectiveness*) identified when using the StepVis or the Ad-Hoc technique.

- *Hypothesis 2*:

$H_{0,2}$: There is no difference in the time spent (*efficiency*) when using the StepVis or the Ad-Hoc technique.

$H_{1,2}$: There is a difference in the time spent (*efficiency*) when using the StepVis or the Ad-Hoc technique.

- *Hypothesis 3*:

$H_{0,3}$: The subject's experience in code inspection does not affect the number of defects identified when using the StepVis or the Ad-Hoc technique.

$H_{1,3}$: The subject's experience in code inspection affects the number of defects identified when using the StepVis or the Ad-Hoc technique.

- *Hypothesis 4*:

$H_{0,4}$: The subject's experience in the Java programming language does not affect the number of defects identified when using the StepVis or the Ad-Hoc technique.

$H_{1,4}$: The subject's experience in the Java programming language affects the number of defects identified when using the StepVis or the Ad-Hoc technique.

3.8 Threats to Validity

There are some levels of validity to consider, which are explained below:

- Internal validity: concerns external factors that could affect the dependent variable.
  a) Interference in subjects' performance due to the grade associated with the task. However, as the grade was assigned just for participation, it is considered that the performance risk was mitigated.
  b) Mistake when recording the final time of technique application: this risk could not be mitigated because each subject was responsible for marking his or her time.
- External validity: concerns the degree to which the results of the study can be generalized to a broader context.
  a) The results cannot be generalized to a professional context. This threat was not mitigated because only students participated in the experimental study.
  b) The Paint software, although used in other experimental studies (Ko et al., 2006; Robbilard et al., 2004), is not representative of industrial products in terms of either size or complexity.

- Conclusion validity: concerns the relationship between the treatment and the outcome.
  a) When data normality could not be assumed, we performed a statistical analysis using non-parametric tests. The data normality was evaluated using the Kolmogorov-Smirnov test. Thus, this kind of risk was mitigated.
  b) The subjects of both groups (StepVis and Ad-Hoc) were trained on and applied just one technique, mitigating possible interference by treatment combination.

3.9 Preparation and Execution

Each subject received the necessary material to execute the following tasks:

- Task 1: filling out a profile questionnaire, which included personal and technical details.
- Task 2: the subjects were trained as follows: G1was trained in code inspection using "stepwise abstraction + visualization", that is, using the CRISTA tool, and G2 was trained only in code inspection, that is, the Ad-Hoc technique. The training was performed using a sample application. This task was performed one week before the execution of the experiment.
- Task 3: inspection of the Paint software according to the assigned technique. The subjects worked under examination conditions and were not allowed to talk to each other or to ask the supervisor.

3.10 Data Collection

Data collection occurred through questionnaires that should be completed by each subject. The group that used the CRISTA tool also answered questions related to the use of the tool.

3.11 Analysis and Interpretation

The analyses were carried out by means of the MiniTab statistical tool and the results are shown in the next sections.

3.11.1 Descriptive Statistics

Figure 2 shows the percentages of subjects who found the defects mentioned previously.

From Figure 2, it can be seen that all subjects who applied StepVis found Defect 1. Almost 100% of these subjects also found Defect 2. The greatest difference occurred for Defect 4, which was found by approximately 93% of subjects who applied StepVis and about 41% those who applied Ad-Hoc. In addition, in relation to Defect 3, subjects who applied StepVis performed slightly worse than those who applied Ad-Hoc. Therefore, the kind of defect, its complexity, and its localization are being investigated in another experiment. Nevertheless some comments can be made:

- Defect 1: This defect is easy to locate in the source code since it involves a single class where it can be observed that only two colors of the RGB pattern are mentioned (red and green). Apparently, the influence of the technique on finding this defect only affected the time spent for its identification, as shown in Figure 3(a).
- Defect 2: This defect is considered complex because its identification is not trivial through static analysis. Regardless of this characteristic, both techniques reached good effectiveness, as shown in Figure 2.
- Defect 3: This defect depends on a deeper comprehension of the source code because it is necessary to comprehend and inspect two different Java classes. The identification of defects that involve more than one class is probably more complex.
- Defect 4: To identify this defect it is necessary to comprehend and inspect three Java classes that are associated through inheritance and dependency. According to Figure 2, the use of StepVis provided better effectiveness than Ad-Hoc.

  Thus, the technique probably influenced the identification of this defect.

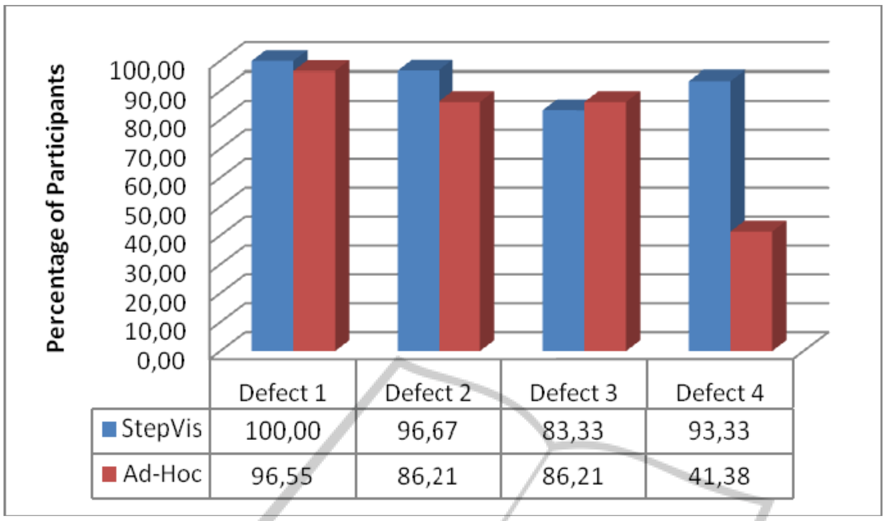
Figure 3 presents the box-plots for each defect. They present the time spent to identify the defect and the technique applied.

The box-plot analysis reveals that the subjects who applied StepVis identified defects in almost the same time interval.

This fact can be observed, for example, in Figure 3(a), where the variability related to StepVis is less than that related to Ad-Hoc. Although it is easy to find Defect 1, with StepVis, 100% of the subjects found it in a short time interval. Almost all subjects who applied Ad-Hoc also identified Defect 1, but they spent more time. Hence, it is interesting to investigate whether there is some relationship between the defect identification and the effectiveness of the technique.

3.11.2 Hypothesis Test

Based on the hypotheses previously presented, the data analysis used for answering them is presented.



	Defect 1	Defect 2	Defect 3	Defect 4
🟦 StepVis	100,00	96,67	83,33	93,33
🟥 Ad-Hoc	96,55	86,21	86,21	41,38

Figure 2: Percentage of participants who found defects.

The null hypothesis is written again to facilitate reading.

$H_{0,1}$: There is no difference in the number of defects (*effectiveness*) identified when using the StepVis or the Ad-Hoc technique.

For this hypothesis the chi-square test was applied using all defects found by all subjects. The p-value was 0.239, meaning that the null hypothesis cannot be rejected; that is, the technique used does not interfere with the number of defects found. However, Defect 4 was found by 93.33% of subjects who used StepVis and by 41.38% of those who used Ad-Hoc. In another experiment, whose data are under analysis, we are investigating different types of defects. Hence, different types of defects were injected in the source code in order to explore the effectiveness of the StepVis technique. This procedure is suggested by some authors whenever the effectiveness of a technique is to be investigated (Mäntylä and Lassenius, 2009; Ko et al., 2006).

$H_{0,2}$: There is no difference in the time spent (*efficiency*) when using the StepVis or the Ad-Hoc technique.

As the data did not present a normal distribution, the non-parametric Mann-Whitney test was used. Considering a significance level of 95%, the p-value was equal to 0.0247, meaning that the null hypothesis can be rejected. This means that the time spent detecting defects depends on the technique used. Hence, using StepVis through CRISTA impacts on the time spent.

$H_{0,3}$: The subject's experience of code inspection does not affect the number of defects identified when using the StepVis or the Ad-Hoc technique.

For this hypothesis the statistical analysis was done using the Pearson correlation. The result shows that experience of inspection did not impact on the number of defects found for either technique. The p-value for the group which applied StepVis was equal to 0.073 and that for the group which applied Ad-Hoc was 0.0935.

Figure 4 shows the software inspection experience for both groups of subjects.

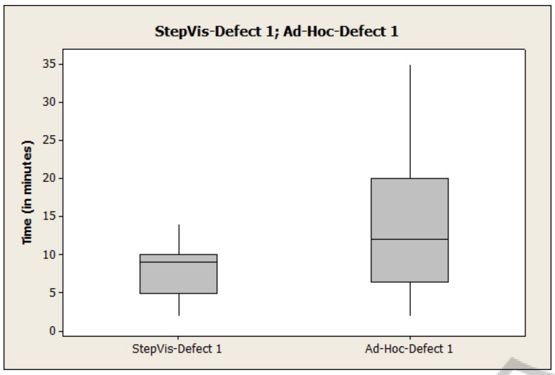
$H_{0,4}$: The subject's experience in the Java programming language does not affect the number of defects identified when using the StepVis or the Ad-Hoc technique.

In order to evaluate this hypothesis the statistical analysis was done using the Pearson correlation. Irrespective of the technique used, the null hypothesis could not be rejected. This means that experience in Java language had no influence on the number of defects found. The p-values for the two groups, StepVis and Ad-Hoc, were 0.285 and 0.475, respectively.

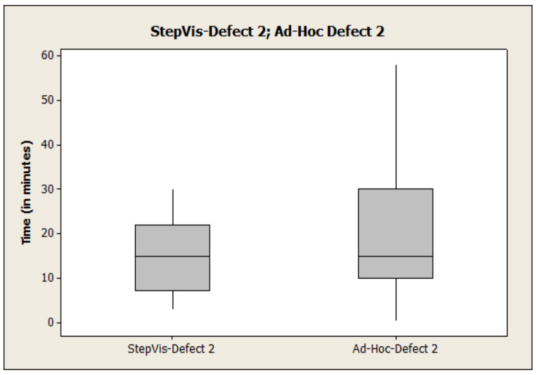
Figure 5 presents the Java experience for both groups.

4 CONCLUSION

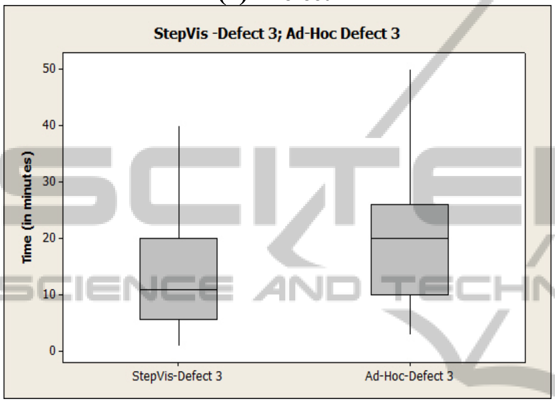
According to the experimental results and analysis, the use of the StepVis technique aided in defect identification. In addition, considering the context of the experiment, the participant's experience of inspection and the programming language had no



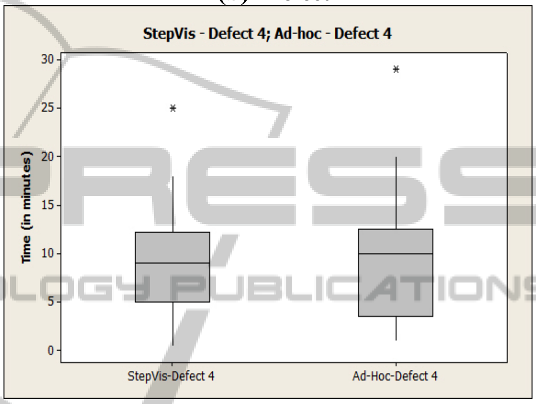
(a) Defect 1



(b) Defect 2

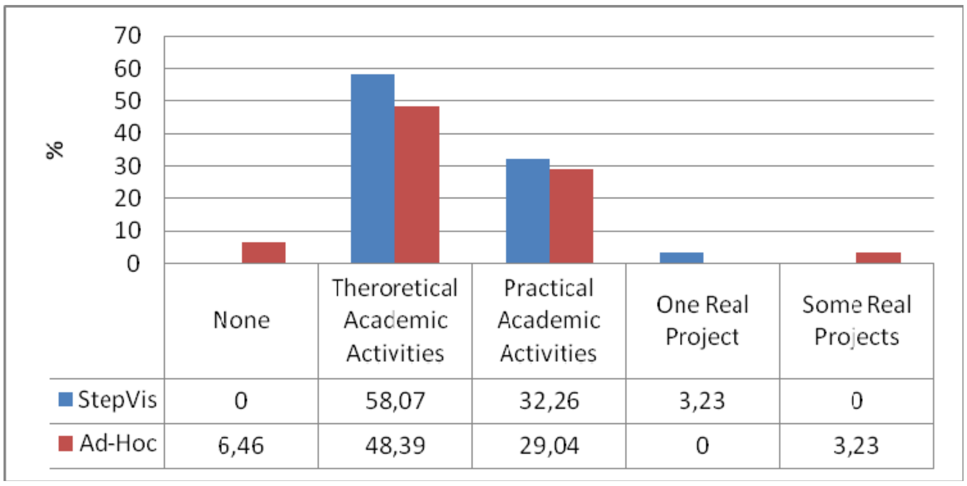


(c) Defect 3



(d) Defect 4

Figure 3: Time spent inspecting the code.



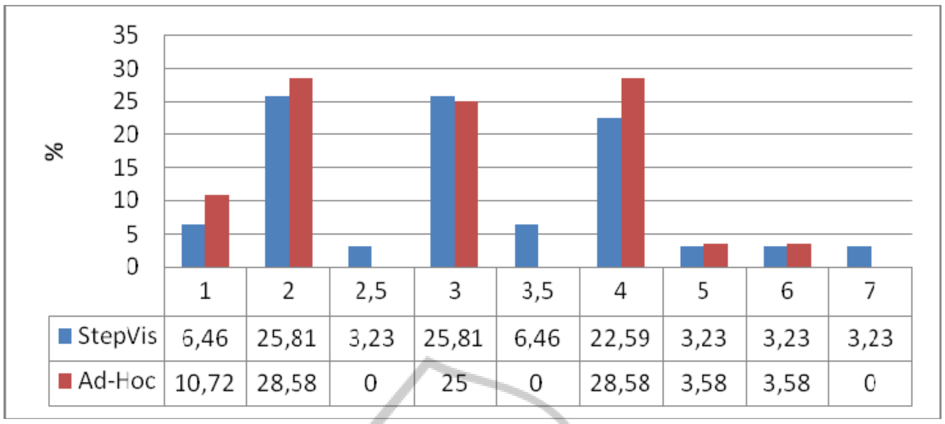
	None	Theroretical Academic Activities	Practical Academic Activities	One Real Project	Some Real Projects
🟦 StepVis	0	58,07	32,26	3,23	0
🟥 Ad-Hoc	6,46	48,39	29,04	0	3,23

Figure 4: Participants' experience in source code inspection.

influence on the results.

Based on the results and the analysis of the questionnaires, the following points can be highlighted:

- When the StepVis technique is used, the inspectors obtain a deep comprehension of the code since this technique requires that all elements of the code are abstracted (from the inner to the outer, however simple they are).
- Despite the probable disadvantage of the necessity of abstracting every element of the code, the use of StepVis enhanced the identification of defects once the whole project was better understood.



	1	2	2,5	3	3,5	4	5	6	7
🟦 StepVis	6,46	25,81	3,23	25,81	6,46	22,59	3,23	3,23	3,23
🟥 Ad-Hoc	10,72	28,58	0	25	0	28,58	3,58	3,58	0

Figure 5: Participants' experience (years) in Java programming language.

- The effectiveness of the code inspection activity was not influenced by the techniques. However, the efficiency was influenced by the technique once the use of StepVis provides evidence of the time improvement for executing the inspection process.
- The effectiveness and efficiency related to StepVis computational support of the CRISTA tool allows the inspection data to be processed and some reports to be generated. This aids in the code comprehension and, consequently, in the identification of defects.
- The computational support of the CRISTA tool allows the inspection data to be processed and some reports to be generated. This aids in the code comprehension and, consequently, in the identification of defects.

In summary, it seems that the StepVis technique provides better conditions for defect identification than the Ad-Hoc inspection.

A new experiment has been conducted with two research purposes: (1) map the types of defects that are found when applying StepVis and their association with the object-oriented features, (2) evaluate the decisions and workflow used by inspectors when applying StepVis aiming at identifying strategies for applying this technique.

ACKNOWLEDGEMENTS

Our thanks to the Brazilian funding agency CNPq.

REFERENCES

Almeida, J. R., Camargo, J. B., and Basseto, B. 2003. Best practices in code inspection for safety-critical software. *IEEE Software*, 20(3):56–63.

Aurum, A., Petersson, H., and Wohlin, C., 2002. State-of-the-art: software inspections after 25 years. Software Testing Verification and Reliability, 12(3):133–154.

Basili, V. R., Caldiera, G., Lanubile, F.,and Shull, F. 1996a. Studies on reading techniques. In: *Annual Software Engineering Workshop*, 21, Greenbelt, Maryland. NASA/Goddard Software Engineering Laboratory Series, December, pp. 59–65.

Basili, V.R., Green, S., Laitenberger, O., Shull, F., Sørumgård, S., and Zelkowitz, M. 1996b. *The empirical investigation of Perspective-Based Reading. Empirical Software Engineering*, 1(2):133–164.

Basili, V., Green, S.,Laitenberger, O.,Lanubile, F.,Shull, F.,Sorumgard, S.,and Zelkowitz,M. 1998. Lab package for the empirical investigation of perspective-based reading, University of Maryland. Available from: http://www.cs.umd.edu/projects/SoftEng/ESEG/manual/pbr_package/manual.html. Last accessed: 29 August 2013.

Belgamo, A., Fabbri, S., and Maldonado, J. C. 2005.TUCCA: Improving the effectiveness of use case construction and requirement analysis. In: *Proceedings of International Symposium on Empirical Software Engineering*, ISESE 2005, Noosa Heads, vol. 1.

Bernhart, M., Mauczka, A., Grechenig, T. 2010. Adopting Code Reviews for Agile Software Development. Agile Conference.

Brothers, L., Sembugamoorthy, V., Muller, M. 1990. ICICLE: groupware for code inspection. In: *Proceedings of the ACM Conference on Computer-supported cooperative work*, CSCW.

Caserta, P.,and Zendra, O., 2011. Visualization of the static aspects of software: a survey. *IEEE Transactions on Visualization and Computer Graphics*, 17(7):913–933.

Deline, R., Czerwinski, M., and Robertson, G., 2005. Easing program comprehension by sharing navigation data. *IEEE Symposium on Visual Languages and Human-Centered Computing*, pp. 241–248.

Dunsmore, A., Roper, M., and Wood, M., 2003. The development and evaluation of three diverse techniques for object-oriented code inspections. In: *IEEE Transactions on Software Engineering*, 29(8):677–686.

Elberzhager, F., Münch, J., and Nha, V. 2012. A systematic mapping study on the combination of static and dynamic quality assurance techniques. *Information and Software Technology*.

Fagan, M. E. 1976. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(7):182–211.

Fagan, M. E. 1986. Advances in software inspections. *IEEE Transactions on Software Engineering, 12*(7):744–751.

Gilb, T., and Graham, D. 1993. Software Inspection. Wokingham, England: *Addison-Wesley*.

Hernandes, E., Belgamo, A., Fabbri, S. 2013. Experimental studies in software inspection process – a systematic mapping. In: *International Conference on Enterprise Information Systems, ICEIS*.

Holzmann, G. J. 2009. SCRUB: a tool for code reviews. Laboratory for Reliable Software, Jet Propulsion Laboratory, California Institute of Technology.

Humphrey, W.S. 1989. Managing the software process. Addison-Wesley Longman Publishing Co.

Johnson, B., and Shneiderman, B. 1991. Tree-maps: aspace-filling approach to the visualization ofhierarchical information structures. In: *Proceedings ofthe 2nd Conference on Visualization*.

Kelly, D.,and Sheppard, T. 2000. A novel approach to inspection of legacy code. *Proceedings of Practical Software Quality Techniques*, PSQT'00, Austin, Texas.

Kelly, D., and Sheppard, T. 2002. Qualitative observations from software code inspection experiments. *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research, CASCON '02*.

Knight C., and Munro M., 1999. Visualising software – a key research area. *Proceedings of the International Conference on Software Maintenance, ICSM'99*, IEEE Press.

Knight C., and Munro M., 2001. Visualising the non-existing, *IASTED International Conference: Computer Graphics and Imaging*, Hawaii, USA.

Ko, A.J., Myers, B.A., Coblenz, M.J.,and Aung, H.H, 2006. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Transactions on Software Engineering*, 32(12): 971–987.

Laitenberger, O., and Debaud. J., 1997. Perspective-based reading of code documents at Robert Bosch GmbH. Information and Software Technology.

Li, X. A., 1995. Comparison-based approach for software inspection. *Proceedings of the 1995 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '95.

Linger, R. C., Mills, R. C., and Witt, B.I. 1979. Structured Programming: Theory and Practice. *Addison-Wesley*.

Mäntylä, M. V., and Lassenius, C. 2009. What types of defects are really discovered in code reviews. *IEEE Transactions on Software Engineering*, 35(3):430–448.

Marucci, R. A., Fabbri, S. C. P. F., Maldonado, J. C., and Travassos, G.H. 2002.OORTs/ProDeS: Definição de técnicas de leitura para um processo de software orientado a objetos. In: *Simpósio Brasileiro de Qualidade de Software*, Gramado, Brazil.

Mayrhauser A., and Vans, A. M. 1998. Program understanding behavior during adaptation of large scale software. *Proceedings of the 6th International. Workshop on Program Comprehension, IWPC '98*, Italy, pp. 164–172.

McMeekin, D. A., Von Konsky, B. R., Chang, E. J.,and Cooper, D. 2009.Evaluating Software Inspection Cognition Levels Using Bloom's Taxonomy. *22nd Conference on Software Engineering Education and Training, CSEET '09*.

Pfeiffer, J., and Gurd, J. 2006. Visualisation-based tool support for the development ofaspect-oriented programs. Aspect-Oriented Software Development Conference.

Porto, D., Mendonça, M., and Fabbri, S., 2009a. CRISTA: A tool to support code comprehension based on visualization and reading technique. *17thIEEE International Conference on Program Comprehension*.

Porto, D, Zamboni, A., Mendonça, M., and Fabbri, S. 2009b. Manutenção de código apoiada pela ferramenta CRISTA. *Anais do VI Workshop de Manutenção de Software Moderna, 2009*, VI WMSWM, Ouro Preto.

Remillard, J. 2005. Souce code review systems. IEEE Software.

Robbilard, M. P., Coelho, W., and Murphy, G. C., 2004. How effective developers investigate source code: an exploratory study. *IEEE Transactions on Software Engineering*, 30(12): 889–903.

Russel, G. W. 1991. Experience with inspection in ultralarge-scale developments, *IEEE Software, 8*(1):25–31.

Siy, H., and Votta, L. 2001. Does the modern code inspection have value? In: *Proceedings of IEEE International Conference on Software Maintenance*.

Travassos, G. H., Shull, F., Carver, J., and Basili, V. R. 2002. Reading techniques for OO design inspections, Technical Report CS-TR-4353, UMIACS-TR-2002-33, University of Maryland, Maryland, 56 p. Available from: http://drum.lib.umd.edu/bitstream/1903/1193/1/CS-TR-4353.pdf.

Vinz, B. L., and Etzkorn L. H., 2006. A synergistic approach to program comprehension. In: *International Conference on Program Comprehension*, ICPC 2006, pp. 69–73.

Wohlin, C., Runeson, P., and Höst, M., 2000. Experimentation in Software Engineering – An Introduction. Sweden: *Springer*.