A Family of Reading Techniques for OO Design Inspections

Guilherme H. Travassos[†,•] travassos@cs.umd.edu
Forrest Shull[‡] fshull@fraunhofer.org
Jeffrey Carver[†] carver@cs.umd.edu

[†]**Experimental Software Engineering Group**
Department of Computer Science
University of Maryland at College Park
A.V. Williams Building
College Park, MD 20742
USA

[•]**Computer Science and System Engineering Department**
COPPE
Federal University of Rio de Janeiro
C.P. 68511 - Ilha do Fundão
Rio de Janeiro – RJ – 21945-180
Brazil

[‡]**Fraunhofer Center - Maryland**
University of Maryland
4321 Hartwick Road
Suite 500
College Park, MD 20742
USA

ABSTRACT

Inspections can be used to identify defects in software artifacts. In this way, inspection methods help to improve software quality, especially when used early in software development. Inspections of software design may be especially crucial since design defects (problems of correctness and completeness with respect to the requirements, internal consistency, or other quality attributes) can directly affect the quality of, and effort required for, the implementation. We have created a new family of "reading techniques" (so called because they help a reviewer to "read" a design artifact for the purpose of finding relevant information) that gives specific and practical guidance for identifying defects in Object-Oriented designs. Each reading technique in the family focuses the reviewer on some aspect of the design, with the goal that an inspection team applying the entire family should achieve a high degree of coverage of the design defects.

In this paper, we present an overview of this new set of reading techniques. We discuss how these techniques were developed and suggest how readers can use them to detect defects in high level object oriented design UML diagrams.

Keywords: OO Design, Reading Techniques, Software Inspection, Software Quality, Empirical Software Engineering

1. Introduction

A software inspection aims to guarantee that a particular software artifact is complete, consistent, unambiguous, and correct enough to effectively support further system development. For instance, inspections have been used to improve the quality of a system's design and code (Fagan, 1976). Because they rely on human understanding to detect defects, they have the advantage that they can be performed as soon as a software work artifact is written and can be used with of different artifacts and notations. Typically, inspections require individuals to review a particular artifact, then meet as a team to discuss and record defects, which are then sent to the document's author to be corrected. Because a team typically performs an inspection, they are a useful way of sharing technical expertise about the quality of the software artifacts among the participants. And, because developers become familiar with the idea of reading each other's artifacts, they can lead to more readable artifacts being produced over time.

On the other hand, the dependence on human effort, causes nontechnical issues to become a factor: reviewers can have different levels of relevant expertise, can get bored if asked to review large artifacts, can have their own feelings about what is or is not important, or can be affected by political or personal issues. For this reason, there has been an emphasis on defining processes that people can use for performing effective inspections.

Most publications concerning software inspections have concentrated on improving the inspection meetings while assuming that individual reviewers are able to effectively detect defects in software documents on their own. Fagan (1986) and Gilb and Graham (1993)

In *Proceedings of WQS'2000 - Workshop Qualidade de Software*, at the XIV Brazilian Symposium on Software Engineering. Joao Pessoa: Brazilian Computer Soceity, 2000. v.1. p.225-237.

emphasizes the inspection *method*[1], identifying the phases of planning, detection, collection and correction for it. Having been the basis for many of the review processes now in place (e.g., at NASA (1993)), they have inspired the direction of much of the research in this area, which has tended to concentrate on improving the review *method*. However, they do not give any guidelines to the reviewer as to how defects should be found in the detection phase; both assume that the individual review of these documents can already be done effectively.

Proposed improvements to Fagan's method often center on the importance and cost of the meeting. However, empirical evidence has questioned the importance of team meetings by showing that meetings do not contribute to finding a significant number of new defects that were not already found by individual reviewers (Votta, 1993) (Porter, 1995). This line of research suggests that efforts to improve the review *technique*, that is, the process used by each reviewer to find defects in the first place could be of benefit.

One approach to doing this is provided by *software reading techniques*. A reading technique is a series of steps for the individual analysis of a software product to achieve the understanding needed for a particular task (Basili et al., 1996). Reading techniques attempt to increase the effectiveness of inspections by providing procedural guidelines that can be used by individual reviewers to examine (or "read") a given software artifact and identify defects. Rather than leave reviewers to their own devices, reading techniques attempt to capture knowledge about best practices for defect detection into a procedure that can be followed. Families of reading techniques have been tailored to defect inspections of requirements (for requirements expressed in English or SCR, a formal notation) and to usability inspections of user interfaces. There is empirical evidence that software reading is a promising technique for increasing the effectiveness of inspections on different types of software artifacts, not just limited to source code (Porter, 1995)(Basili et al., 1996)(Basili et al., 1996b)(Fusaro et al., 1997)(Shull, 1998)(Zhang, 1998).

In this work, we describe a family of software reading techniques for the purpose of defect detection of high-level Object-Oriented (OO) designs diagrams represented using Unified Modeling Language (UML) [Fowller00]. The Object-Oriented Reading Techniques (OORTs) consist of 7 different techniques that support the reading of different design diagrams. The development of these techniques has been supported by a series of empirical experiments.

With these experiments we are looking for answers for the following questions:

- Is the idea of object-oriented reading techniques feasible?
- Are the techniques technically sounded and described in such way that they can be used to inspect high-level object-oriented design?
- Can the techniques be used in the context of a controlled software development process?
- Are the techniques usable in an industrial software development process?

By applying what we had learned about inspections with PBR (Shull et al., 2000) to this new domain, we were able to empirically evolve the techniques and demonstrate their effectiveness. The results we have so far provide evidence that the OORTs are feasible and can support readers in identifying different types of design defects (Travassos et al., 1999a)(Shull et al., 1999)(Travassos et al., 1999b).

Section 2 briefly describes object-oriented design in terms of the information that is important to be checked during software inspections. Section 3 introduces the reading

[1] In this text we distinguish a "technique" from a "method" as follows: A technique is a series of steps, at some level of detail, that can be followed in sequence to complete a particular task. We use the term "method" as defined in (Basili, 1996), "a management-level description of when and how to apply techniques, which explains not only how to apply a technique, but also under what conditions the technique's application is appropriate."

In *Proceedings of WQS'2000 - Workshop Qualidade de Software*, at the XIV Brazilian Symposium on Software Engineering. Joao Pessoa: Brazilian Computer Soceity, 2000. v.1. p.225-237.

techniques, showing the different types of defects such techniques are intended to identify and an outline of the whole set of techniques. The fourth section discusses how the techniques were developed. Finally, some suggestions for future work are discussed in the conclusions.

2. Object Oriented Designs in UML

A high level design is a set of artifacts concerned with the representation of real world concepts. As a consequence of using the object-oriented paradigm these concepts are represented as a collection of discrete objects that incorporate both data structure and behavior.

High-level design activities start after the software product requirements are captured; they deal with the problem description but do not consider the constraints regarding it. That is, these activities are concerned with taking the functional requirements and mapping them to a new notation or form, using the paradigm constructs to represent the system via design diagrams instead of just a textual description. Instead of using this approach to solve the problem, developers are trying to understand it. At the end, a set of well related, but notational different, artifacts are built. Since high level design is built at a different time than the requirements, using a different viewpoint and abstraction level, it is difficult to inspect these documents to verify both whether they are consistent among themselves and if the requirements were correctly and completely captured.

The main interest of this work is to define reading techniques that can be applied on high-level design documents. We feel that reviews of high-level designs may be especially valuable since they help to ensure that developers have adequately understood the problem before defining the solution. Because the low-level design builds a model for the code with a specific solution to the problem described in the high-level design, it is important that the quality of the high-level design be as high as possible. Reviews of this kind can help ensure that low-level design starts from a high-quality base.

More specifically, the reading techniques described in this work are tailored to inspections of high-level design artifacts that capture the static and dynamic views of the problem using UML notation: class, sequence, and state diagrams. Usually, these are the main UML diagrams that developers build for high-level OO design. To compare design contents against requirements, we expect that there will be a textual description of the functional requirements that may also describe certain behaviors using more specialized representations such as use-cases [Jacobson95].

Thus, we identify the following as important sources of information for ensuring the quality of a UML high level design (Travassos et al, 1999b):

- A set of functional requirements that describes the concepts and services that are necessary in the final system;
- Use cases that describe important concepts of the system (which may eventually be represented as objects, classes, or attributes) and the services it provides;
- A class diagram (possibly divided into packages) that describes the classes of a system and how they are associated;
- A set of class descriptions that lists the classes of a system along with their attributes and behaviors;
- Sequence diagrams that describe the classes, objects, and possibly actors of a system and how they collaborate to capture services of the system;
- State diagrams that describe the internal states in which a particular object may exist, and the possible transitions between those states.

In *Proceedings of WQS'2000 - Workshop Qualidade de Software*, at the XIV Brazilian Symposium on Software Engineering. Joao Pessoa: Brazilian Computer Soceity, 2000. v.1. p.225-237.

3. Reading Techniques for high-level design

Each reading technique can be thought of as a set of procedural guidelines that reviewers can follow, step-by-step, to examine a set of diagrams and detect defects. The types of defects on which our techniques are focused, as listed in Table 1, are based on earlier work with requirements inspections (Shull et al., 2000). The defect taxonomy is important since it helps focus the kinds of questions reviewers should answer during an inspection.

Type of Defect	***Description***
Omission	One or more design diagrams that should contain some concept from the general requirements or from the requirements document do not contain a representation for that concept.
Incorrect Fact	A design diagram contains a misrepresentation of a concept described in the general requirements or requirements document.
Inconsistency	A representation of a concept in one design diagram disagrees with a representation of the same concept in either the same or another design diagram.
Ambiguity	A representation of a concept in the design is unclear, and could cause a user of the document (developer, low-level designer, etc.) to misinterpret or misunderstand the meaning of the concept.
Extraneous Information	The design includes information that, while perhaps true, does not apply to this domain and should not be included in the design.

Table 1 – Types of software defects, and their specific definitions for OO designs

We defined one reading technique for each pair or group of diagrams that could usefully be compared against each other. For example, use cases needed to be compared to interaction diagrams to detect whether the functionality described by the use case was captured and all the concepts and expected behaviors regarding this functionality were represented. The full set of our reading techniques is defined as illustrated in Figure 2, which differentiates horizontal[2] (comparisons of documents within a single lifecycle phase) from vertical[3] (comparisons of documents between phases) reading.

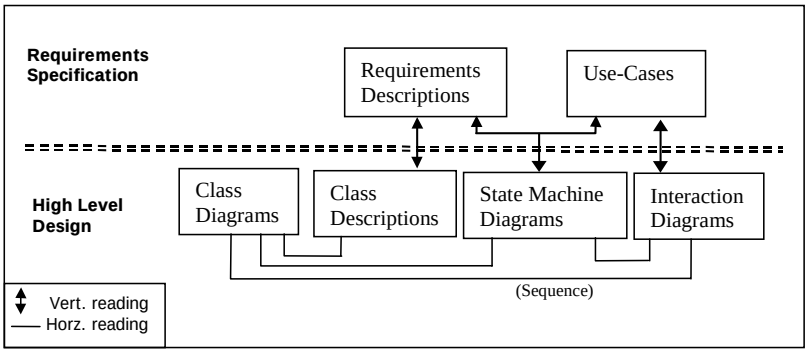

Figure 2 – Set of OO Reading Techniques

While horizontal reading aims to identify whether all of the design artifacts are describing the same system, vertical reading tries to verify whether those design artifacts represent the right system, which is described by the requirements and use-cases. So, the goal is that when all the techniques are used together, then all the quality issues in the design are covered. The development team can use the whole set of the techniques, but if some design

[2] Consistency among documents is the most important feature here.
[3] Traceability between the phases is the most important feature here.

In *Proceedings of WQS'2000 - Workshop Qualidade de Software*, at the XIV Brazilian Symposium on Software Engineering. Joao Pessoa: Brazilian Computer Soceity, 2000. v.1. p.225-237.

artifacts do not exist, there is no impact on the design inspection process[4]. The horizontal techniques should be performed before the vertical techniques, however, a subset or reordering of the techniques may be chosen based on important attributes of the design to be reviewed. This is particularly interesting when developers are dealing with specialized application domains. For example, consider a system whose functionality is based mainly on its reaction to stimuli where state machine diagrams are common. In this situation, it could be beneficial to use the reading techniques that focus on state machine diagrams before using the reading techniques that focus on the other design diagrams. For conventional systems, such as database systems, the semantic model of the information and the flow of the transactions seem to be the important information. Therefore, a subset of the techniques could be picked that focus on this information. In this situation, first reading the class diagram against the sequence diagrams seems to be a good idea then continuing with the rest of the techniques.

Further description about the process of applying the reading techniques can be found in (Travassos et al., 1999b). Information about the techniques and a complete definition for all the terms and definitions used in the context of this paper can be found in (Shull et al., 1999), which is accessible via the web.

4. The development of OORT's

The evolution of OORT's was supported by empirical experiments. We have modified and improved the techniques based on a series of empirical studies. Figure 3 shows the series of experiments conducted since 1998.

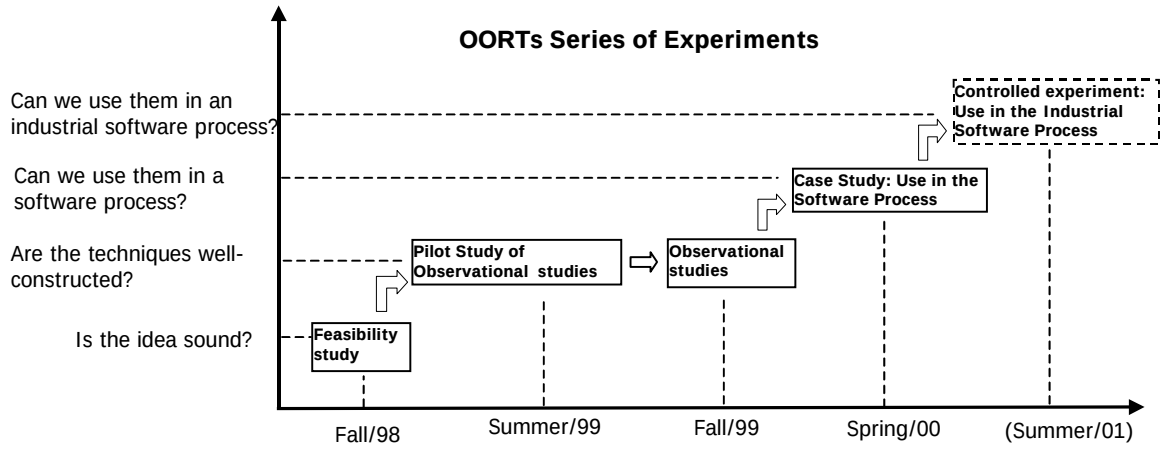

Figure 3 –Experimentation process to develop OO Reading

Initial validation was accomplished by means of a study (Travassos et al., 199) (Shull et al., 1999) that provided evidence for the feasibility of these techniques. Using the techniques did allow teams to detect defects, and in general subjects agreed that the techniques were helpful. Also, the vertical techniques tended to find more defects of omitted and incorrect functionality, while the horizontal techniques tended to find more defects of ambiguities and inconsistencies between design documents, lending some credence to the idea that the distinction between horizontal and vertical techniques is real and useful. A full description of the results can be found in (Travassos et al. 1999).

[4] However, this situation is not true for the software process as a whole. Some artifacts are important, such as a class diagram if missing implies that the design didn't capture the static view of the problem.

Further studies have been undertaken to improve the practical applicability of the techniques. As a result of specific feedback from the feasibility study, we developed a second version of the techniques. The feasibility study had identified *global* issues for improvement, that is, issues that affected the entire process, such as the amount of semantic versus syntactic checking. This version of the techniques was then studied using an observational approach (i.e., using experimental methods suitable for understanding the process by which subjects apply the techniques) (Travassos et al., 1999b). Because this observational approach was a somewhat unusual approach, we first performed a pilot study to debug the observational approach and get it to work in our setting. The observational approach was necessary to understand what improvements might be necessary at the level of individual steps, for example, whether subjects experience difficulties or misunderstandings while applying the technique (and how these problems may be corrected), whether each step of the technique contributes to achieving the overall goal, and whether the steps of the technique should be reordered to better correspond to subjects' own working styles.

Reading 3 -- Sequence x State diagrams

Goal: To verify that every state transition for an object can be achieved by the messages sent and received by that object.

Inputs to Process:

1. Sequence diagrams that describe the classes, objects, and possibly actors of a system and how they collaborate to capture services of the system.
2. State diagrams that describe the internal states in which an object may exist, and the possible transitions between states.

For **each state diagram,** perform the following steps:

I. **Read the state diagram to understand the possible states of the object and the actions that trigger transitions between them.**

INPUTS: State diagram (SD).

OUTPUTS: Transition Actions (marked and labeled in green on SD);
Discrepancy reports.

A. Determine which class is being modeled by this state diagram.
   1) **If you can't determine the class that is being modeled, then something has been omitted or is ambiguous. Indicate this on a discrepancy report form.**

B. Trace the sequence of states and the *transition actions* (system changes during the lifetime of the object, which trigger a transition from one state to another) through the state diagram. Begin at the start state and follow the transitions until you reach the end state. Make sure you have covered all transitions.

C. Highlight transition actions (represented by arrows) as you come to them using a green pen. For example, the state diagram provided in Example 5 contains seven transition actions. The arrow leading from the state labeled "authorizing" back to itself represents an action that does not actually change the state of the object. Give each action a unique label [A1, A2, …].

D. Think about the states and actions you have just identified, and how they fit together.
   1) **Make sure that you can understand and describe what is going on with the object just by reading the state machine. If you cannot, then the state machine is ambiguous. Indicate this on the discrepancy report form.**

Figure 4 – An excerpt of a Horizontal Reading

These observational investigations showed that horizontal and vertical reading techniques really find different types of defects. It lead us to produce a third version of the techniques, exploring more the semantics behind the design models and reflecting the observed way readers used to apply the techniques while inspecting design artifacts. We also

In *Proceedings of WQS'2000 - Workshop Qualidade de Software*, at the XIV Brazilian Symposium on Software Engineering. Joao Pessoa: Brazilian Computer Soceity, 2000. v.1. p.225-237.

changed from reporting defects to reporting discrepancies[5], reflecting the fact that inspectors and designers may have different ideas about the design, so discrepancies must be evaluated by the designer to determine if they are real defects. The observational studies also allowed us to get some clarifications about the role of domain knowledge for these two sets of reading techniques, especially for horizontal reading. Since horizontal reading is a largely syntactic check of consistency between two design diagrams, it is not expected to require domain knowledge. However, we found that domain knowledge does not influence design inspections using the techniques. Indeed, development expertise played a more important role. Additional improvements were made regarding usage training and how readers report discrepancies. Figures 4 and 5 show current version excerpts of horizontal and vertical reading techniques. These techniques can be compared with the previous ones presented in (Travassos et al., 1999b) to verify how empirical experimentation helped to evolve them.

The first three experiments aimed to apply the techniques regardless the software process being accomplished. They concentrated only in the high-level design activity and how developers were using the techniques rather than trying to understand the effects of the techniques when used in the context of a full software development process. To understand the use of these techniques in a software development process a fourth experiment was performed. Partial results highlight the benefits of using such techniques to inspect design models. However, more data analysis is still necessary before a complete discussion of the results.

5. Ongoing Work

The Object Oriented reading techniques (OORTs) have been, and still are, evolving since their first definition. New issues and improvements have been included based on the feedback of readers and volunteers. Throughout this process, we have been trying to capture new features and to understand whether the latest version of the reading techniques keeps its feasibility and interest. We have found observational techniques useful, because they have allowed us to follow the reading process as it occurred, rather than trying to interpret the readers' post-hoc answers as we have done in the past. Observing how readers normally try to read diagrams challenged many of our assumptions about how our techniques were actually being applied.

However, one question is still open in this area. It regards the level of automated support that should be provided for such techniques. The observational studies have allowed us to understand which steps of the techniques can feel especially repetitive and mechanical to the reader. So, the clerical activities regarding the reading process using OORTs must be precisely defined and identified. For this situation, further observational studies play an important role and they should be executed aiming to collect suggestions on how to automate the clerical activities concerned with OORTs.

Currently, the techniques were used in different contexts and by more than 150 different expertise level developers, from academy to industry. The results we have so far support answers for our first two questions. Moreover, we have observed from the preliminary results from last experiment (Figure 3, Spring/00) that the techniques are ready to be tested in real projects. Experimental replications are planned to take place in different companies and research groups. In each experiment different issues regarding the techniques can be identified in order to evolve them or understand them at a deeper level. This series of

[5] Discrepancies are differences that show up between documents. They can become defects after the end of the whole reading process.

In *Proceedings of WQS'2000 - Workshop Qualidade de Software*, at the XIV Brazilian Symposium on Software Engineering. Joao Pessoa: Brazilian Computer Soceity, 2000. v.1. p.225-237.

experiments is an evolutionary process. The feedback from the readers and the observation of the techniques usage are playing an important role as we work towards a useful and feasible set of reading techniques for OO design.

Reading 7 -- State Diagrams x Requirements Description and Use-cases

Goal: To verify that the state diagrams describe appropriate states of objects and events that trigger state changes as described by the requirements and use cases.

Inputs to process:

1. The set of all state diagrams, each of which describes an object in the system.
2. A set of functional requirements that describes the concepts and services that are necessary in the final system.
3. The set of use cases that describe the important concepts of the system

For **each state diagram**, do the following steps:

I. **Read the state diagram to basically understand the object it is modeling.**

II. **Read the requirements description to determine the possible states of the object, which states are adjacent to each other, and events that cause the state changes.**

INPUTS: Requirements Description (RD)

OUTPUTS: Object States (marked in blue on SD)

Adjacency Matrix

A. Put away the state diagram and erase any (*) from that are in the requirements from previous iterations of this step. Now, read through the requirements looking for places where the concept is described or for any functional requirements in which the concept participates or is affected. When you locate one of these, mark it in pencil with a (*) so that it will be easier to use for the remainder of the step. Focus on these parts of the RD for the rest of the step.

B. Locate descriptions of all of the different states that this object can be in. To locate a state, look for attribute vales or combinations of attribute values that can cause the object to behave in a different way. When you locate a state underline it with a blue pen and give it a number.

C. Now identify which one of the numbered states is the Initial state. Using a blue pen, mark it with an "I". Likewise mark the end state with an "E".

D. When you have found all of the states, on a separate sheet of paper, create a matrix with 1..N across the top and 1..N down the left side, where 1..N represents the numbers that you gave to the states in the previous step.

E. For each pair of states, if the object can change from the state represented by the number on the left hand side to the state represented by the number on the top row, then mark the box at the intersection of the row and column. If you can determine the event(s) that cause the state change put that in the box, if not just put a check mark (the event will be determined in a later step). If you can determine that it is not possible for the transition to happen then place an X in the box. If you cannot make a definite determination then leave the box blank for now.

F. For any event that you have identified above, if there are any constraints described in the requirements, then write those by the event in the matrix.

III. **Read the Use cases and determine the events that can cause state changes.**

INPUT: Use Cases

OUTPUT: Completed Adjacency Matrix

A. Read through the use cases and find the ones in which the object participates. Focus on these for the rest of the step.

B. For each box in the adjacency matrix that has a check mark in it, look through the use cases and determine what event(s) can cause that transition. These events may not be obvious and may require you to abstract the use-cases and think about what is actually going on with each object. Erase the check mark and write this event(s) in its place.

C. For each box that is blank in the adjacency matrix, see if any event that can cause that transition is described in the use cases. If it is, then write that event in the box, if not then place an X in the box.

…

Figure 5 – An excerpt of a Vertical Reading Technique

The results of these experiments will be published in future technical publications, which will be available at http://www.cs.umd.edu/projects/SoftEng/ESEG.

In *Proceedings of WQS'2000 - Workshop Qualidade de Software*, at the XIV Brazilian Symposium on Software Engineering. Joao Pessoa: Brazilian Computer Soceity, 2000. v.1. p.225-237.

Acknowledgements

This work was partially supported by UMIACS and by NSF grant CCR9706151.
We recognize the support, management and dedication of Prof. Victor R. Basili for this research work. Dr. Travassos also recognizes the partial support from CAPES- Brazil.

References

Basili, V. R.; Green, S.; Laitenberger, O.; Lanubile, F.; Shull, F.; Sorumgard, S. and Zelkowitz, M. V. (1996) The Empirical Investigation of Perspective-Based Reading, Empirical Software Engineering Journal, I, 133-164.

Basili, V.; Caldiera, G.; Lanubile, F. and Shull, F. (1996b). Studies on reading techniques. *In Proc. of the Twenty-First Annual Software Engineering Workshop*, SEL-96-002, pages 59-65, Greenbelt, MD, December.

Fagan, M. E. (1976). "Design and Code Inspections to Reduce Errors in Program Development." IBM Systems Journal, 15(3):182-211.

Fagan, M. (1986). "Advances in Software Inspections." IEEE Transactions on Software Engineering, 12(7): 744-751, July.

Fowller, M.; Scott, K. (2000). UML Distilled: Applying the Standard Object Modeling Language, Second edition, Addison- Wesley. ISBN 0-201-65783-X

Fusaro, P.; Lanubile, F. and Visaggio, G. (1997). A replicated experiment to assess requirements inspections techniques, *Empirical Software Engineering Journal*, vol.2, no.1, pp.39-57.

Gilb , T. and Graham, D. (1993). Software Inspection. Addison-Wesley, reading, MA.

Jacobson, I.; Christerson, M.; Jonsson, P. and Overgaard, G. (1995). Object-Oriented Software Engineering: A Use Case Driven Approach, Addison-Wesley, revised printing.

NASA. (1993). National Aeronautics and Space Administration, Office of Safety and Mission Assurance. "Software Formal Inspections Guidebook". Report NASA-GB-A302, August 1993.

Porter, A.; Votta Jr., L. and Basili, V. (1995). Comparing Detection Methods for Software Requirements Inspections: A Replicated Experiment. IEEE Transactions on Software Engineering, 21(6): 563-575, June.

Shull, F. (1998). *Developing Techniques for Using Software Documents: A Series of Empirical Studies*. Ph.D. thesis, University of Maryland, College Park, December 1998.

Shull, F.; Travassos, G. and Basili, V. (1999). Towards Techniques for Improved OO Design Inspections. Workshop on Quantitative Approaches in Object-Oriented Software Engineering (in association with the 13th European Conf. on Object-Oriented Programming), Lisbon, Portugal. On line at http://www.cs.umd.edu/projects/SoftEng/ESEG/papers/postscript/ecoop99.ps.

Shull, F.; Travassos, G. H.; Carver, J. and Basili, V. R. (1999). Evolving a Set of Techniques for OO Inspections. Technical Report CS-TR-4070, UMIACS-TR-99-63, University of Maryland, October. http://www.cs.umd.edu/Dienst/UI/2.0/Describe/ncstrl.umcp/CS-TR-4070

Shull, F.; Rus, I. and Basili, V. (2000). How Perspective Based Reading can Improve Requirements Reading. IEEE Computer, July.

Travassos, G.; Shull, F.; Fredericks, M., and Basili, V. (1999). Detecting Defects in Object-Oriented Designs: Using Reading Techniques to Improve Software Quality. In the Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), Denver, Colorado.

Travassos, G. H.; Shull, F. and Carver, J. (1999b). Evolving a Process for Inspecting OO Designs. XIII Brazilian Symposium on Software Engineering: *Workshop on Software Quality*. Florianópolis, Curitiba, Brazil, October. On line at http://www.cs.umd.edu/projects/SoftEng/ESEG/papers/ postscript/wqs99.ps.

In *Proceedings of WQS'2000 - Workshop Qualidade de Software*, at the XIV Brazilian Symposium on Software Engineering. Joao Pessoa: Brazilian Computer Soceity, 2000. v.1. p.225-237.

Travassos, G. H.; Shull, F.; Carver, J. and Basili, V. R. (1999c). Reading Techniques for OO Design Inspections, 24th Annual Software Engineering Workshop, NASA/SEL, Greenbelt, USA, Dezembro. On line at http://sel.gsfc.nasa.gov/website/sew/1999/topics/travassos_SEW99paper.pdf.

Votta Jr., L. G. (1993). "Does Every Inspection Need a Meeting?" ACM SIGSOFT Software Engineering Notes, 18(5): 107-114, December.

Zhang, Z.; Basili, V. and Shneiderman, B. (1998). An empirical study of perspective-based usability inspection. Human Factors and Ergonomics Society Annual Meeting, Chicago, October.

In *Proceedings of WQS'2000 - Workshop Qualidade de Software*, at the XIV Brazilian Symposium on Software Engineering. Joao Pessoa: Brazilian Computer Soceity, 2000. v.1. p.225-237.