This paper was accepted at the Workshop on Quantitative Approaches in Object-Oriented Software Engineering, in association with the 13th European Conf. on Object-Oriented Programming, Lisbon, Portugal, 1999.

Towards Techniques for Improved OO Design Inspections

Forrest Shull
Guilherme H. Travassos[1]
Vic Basili

Experimental Software Engineering Group
Department of Computer Science
University of Maryland at College Park
A.V. Williams Building
College Park – MD – 20742
USA

- Position Paper -

Keywords: OO Design, Reading Techniques, Software Quality, Software Inspection, Quantitative tracking of an OO development activity, Empirical studies on the use of OO measures for process management

I. 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 [Fagan76]. 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. 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 (e.g. [Fagan86, Gilb93]). 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 [Votta93, Porter95].

"Software 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. 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 [Porter95, Basili96, Basili96b, Fusaro97, Shull98, Zhang98]. Software reading can be performed on all documents associated with the software process, and is an especially useful method for detecting defects

[1] On leave from the Federal University of Rio de Janeiro – COPPE/Computer Science and System Engineering Department, partially supported by CAPES – Brazil

This work was partially supported by UMIACS and by NSF grant CCR9706151

since it can be applied as soon as the documents are written. So, artifacts like requirements documents, use-cases, scenarios descriptions, design diagrams, source code and so on, could be read throughout the software lifecycle allowing developers to look for defects. Inspections are especially important in the case of design documents, 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 of a system.

In this work, we concentrate specifically on inspections of high-level Object-Oriented (OO) designs. An OO design is a set of diagrams concerned with the representation of real world concepts as a collection of discrete objects that incorporate both data structure and behavior. Normally, high-level design activities start after the software product requirements are captured. So, concepts must be extracted from the requirements and described using the paradigm constructs. This means that requirements and design documents are built at different times, using a different viewpoint and abstraction level. When high-level design activities are finished, the documents, basically a set of well-related diagrams, can be inspected to verify whether they are consistent among themselves and if the requirements were correctly and completely captured.

This position paper discusses some issues regarding the definition and application of reading techniques that can be used to read requirements, use-cases and design artifacts within a domain in order to identify defects among them**.** The paper is organized as follows: section 2 discusses the procedural guidelines we have developed to assist reviewers in OO inspections. Section 3 discusses our work in finding metrics to evaluate inspection effectiveness, and introduces a taxonomy that we feel describes the different types of design defects. Section 4 concludes by describing our ongoing work motivated by open questions.

II. Techniques for Inspecting High-Level OO Designs

We identify two different types of OO design activities: high and low level. High-level design activities 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. Such an approach allows developers to understand the problem rather than to try to solve it. Low-level design activities deal with the possible solutions for the problem; they depend on the results from the high-level activities and nonfunctional requirements, and they serve as a model for the code. Our interest is to define reading techniques that could 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. Since low-level designs use the same basic diagram set as the high-level design, but using more detail, reviews of this kind can help ensure that low-level design starts from a high-quality base.

In this work, we concentrate on high-level artifacts represented using UML [Fowller97], which is considered a notational approach and does not define how to organize development tasks. Therefore, it can be tailored to different development situations. UML diagrams capture the static and dynamic view of the real world as described by the object-oriented constructs. We focused our reading techniques on the following high-level design diagrams: class, interaction (sequence and collaboration), state machine and package. Usually, these are the main UML diagrams that developers build for high-level OO design. They capture the static and dynamic views of the problem, and even allow the teamwork to be organized, based on packaging information.

The design content needs to be compared against the requirements, which can likewise be described using a number of separate diagrams to capture different aspects. In particular, 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].

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. We defined one reading technique for each pair 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 could be captured and all the concepts and expected behaviors regarding this functionality were represented. Table 2 shows the resulting reading technique for this comparison, including the marking mechanisms used to assist the reading in detecting defects. The full set of our reading techniques is defined as illustrated in Figure 1, which differentiates horizontal (comparisons of documents within a single lifecycle phase) from vertical (comparisons of documents between phases) reading. The lines between the artifacts indicate that there is a reading technique to be used to read one against the other. The lines marked with a (*) represent the techniques that have been evaluated in a feasibility study.

This feasibility study was run in the context of an undergraduate software engineering course at UMCP during the Fall 1998 semester. Students were asked to apply the techniques to inspect a design of the semester project they had to implement. A detailed description of the study is provided in a web-based lab package [http://www.cs.umd.edu/project/SoftEng/ESEG/manual/tbr_package/]. This study provided evidence for the feasibility of these techniques and suggested specific ways in which they might be improved. As a result we are currently developing a second version of the techniques.

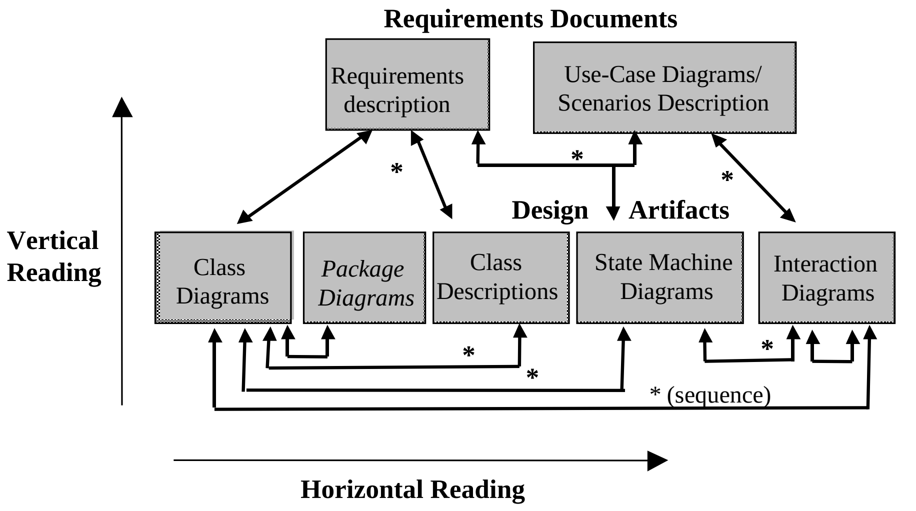

Figure 1 –Reading Techniques

6) Reading 6 -- Use-cases x Sequence Diagrams
For each use-case do:

1) Read the use-case description and identify the nouns
   Underline and number them with a blue pen as they are found
2) Read the sequence diagram to identify if all nouns (objects) are represented
   Mark the underlined nouns with a blue symbol (*) if they are represented in the sequence diagram.
3) Read the use-case and for each noun identify actions (behaviors) and possible information (data) exchanged with other nouns. Mark the order (sequence) of the actions. Look for the condition that activates the behaviors or actions.
   Underline the identified actions (behaviors) with a green pen and information (data) with a yellow pen and write in the same number given to each noun
   Number the identified actions (behaviors) in the order of actions with a green pen
4) Read the sequence diagram and verify if the behaviors and data were represented in the right order
   Mark the behaviors with a green symbol (*) and data with a yellow symbol (*) if they are represented in the right order in the sequence diagram.

Table 1. Vertical Reading: Verifying Sequence with respect to Use-Cases Diagrams (first version)

III. Measures of Effectiveness

A natural measure of the effectiveness of an inspection process is the number of defects found. But this requires a definition of what exactly a design defect is. For this purpose, we borrow a defect taxonomy that had proven effective for requirement defects [Basili96], as defined in Table 2. This taxonomy classifies defects by identifying the general sources of information that are relevant for the system being built, and reasoning about where mismatches may exist between them. For example, the information in the *design artifact* must be compared to the *general requirements* in order to ensure that the system described by the artifact matches the system that is supposed to be built. Similarly, a reviewer of the *design artifact* must also use general *domain knowledge* to make sure that the artifact describes a system that is meaningful and can be built. At the same time, *irrelevant information* from other domains should typically be prevented from appearing in the artifact, since it can only hurt clarity. Any artifact should also be analyzed to make sure that it is self-consistent and clear enough to support only one interpretation of the final system. Table 2 shows how the definitions of [Basili96] can be tailored to OO designs.

Defect	***General Description***	***Applied to design***
Omission	Necessary information about the system has been omitted from the software artifact.	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	Some information in the software artifact contradicts information in the requirements document or the general domain knowledge.	A design diagram contains a misrepresentation of a concept described in the general requirements or requirements document.
Inconsistency	Information within one part of the software artifact is inconsistent with other information in the software artifact.	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	Information within the software artifact is ambiguous, i.e. any of a number of interpretations may be derived that should not be the prerogative of the developer doing the implementation.	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	Information is provided that is not needed or used.	The design includes information that, while perhaps true, does not apply to this domain and should not be included in the design

Table 2 – Types of software defects, and their specific definitions for OO designs.

The taxonomy has proved helpful to our study because it allows more precise metrics to be collected than simply "number of defects." Instead, we can assess the number of defects found in each category. This can help us understand the effect of various types of reading techniques. For example, in our feasibility study mentioned in the last section, there is a measurable difference in the types of defects found due to whether the subject had used horizontal or vertical reading. Subjects using vertical reading tended, on average, to report slightly more defects of omission and incorrect fact (i.e. of types of defects uncovered by comparisons against the requirements) than those using horizontal reading (6.8 versus 5.4 defects). Likewise, subjects using horizontal reading tended to report more defects of ambiguity and inconsistency (i.e. of types of defects uncovered by examination of the design diagrams themselves) than subjects using vertical reading (5.3 versus 2.9).

V. Conclusions and Ongoing Work

Our initial feasibility study was useful in that it demonstrated that such reading techniques can be used as part of design inspections, and do help reviewers detect defects. It has also identified weaknesses in the first version of the techniques that have led to a second version. However, in assessing the effectiveness of this new version we will need to make use of more sophisticated evaluation methods and metrics.

Our primary need at this point in the work is better metrics for assessing inspection effectiveness. Most importantly, we need a measure that takes into account "false positives," i.e. items that are reported by reviewers but which should really not be considered defects. False positives are important because they require resources to be expended to investigate them, but do not actually lead to any improvements with the document. In the worst case, a "correction" to a false positive may actually introduce new defects into the design. Minimizing false positives is thus an important goal of any inspection technique. Understanding false positives is difficult because it requires understanding what types of design defects really will cause serious problems at later stages of the lifecycle. That is, it requires a study of defects and their impact throughout the lifecycle, not just in the design phase. To develop this understanding, a case study that allows us to follow a single project from beginning to end may be the logical next step. As for all case studies, however, it would not be possible to determine how widely applicable the results were.

Another direction for this study would be to develop qualitative as well as quantitative means of assessment. Qualitative methods would help us understand the process as well as the results. For example, we could make use of interviews and questionnaires with subjects who are applying the techniques to understand how satisfied they were, what difficulties they had, and where improvements might be made. A more extreme approach would be to use protocol analysis, in which a subject is asked to think aloud while applying the technique. This provides insights at a much more detailed level as to where the subject encounters difficulties understanding or following the guidelines, or if he or she feels it necessary to augment the techniques with approaches of their own. Qualitative methods such as these have been effectively applied in studies with similar types of goals. Examples include studying the work practices of maintainers to design useful tool support [Singer96], and studying how developers use OO frameworks in order to design effective learning strategies [Basili98].

Finally, another necessary type of analysis will involve the defect taxonomy introduced in section 3. We should make use of expert opinion (where experts include skilled practitioners and researchers) to understand if the taxonomy captures the important types of defects in OO designs. Then we can search for correlation between defect types and review effectiveness. For example, a representative taxonomy would give us a tool to answer questions such as: "Are certain types of defects particularly difficult for inexperienced reviewers to find?"

All of the analysis methods listed here will be necessary to get a more refined understanding of whether our techniques for OO design inspections are truly effective and useful. Perhaps more importantly, such methods would give us insight into where the process might be changed to improve the inspection activity.

Acknowledgements

Our thanks go to Dr. Yong Rae Kwon for his assistance defining the techniques, and to Michael Fredericks for his help running the experiment.

References

[Fagan76] M. E. Fagan; 1976. Design and code inspections to reduce errors in program development. IBM Systems Journal, 15(3):182-211

[Fagan86] M. Fagan; "Advances in Software Inspections." *IEEE Transactions on Software Engineering*, 12(7): 744-751, July 1986.

[Basili96] V. R. Basili, S. Green, O. Laitenberger, F. Lanubile, F. Shull, S. Sorumgard, M. V. Zelkowitz; The Empirical Investigation of Perspective-Based Reading, Empirical Software Engineering Journal, I, 133-164, 1996

[Basili96b] V. Basili, G. Caldiera, F. Lanubile, and F. Shull. Studies on reading techniques. *In Proc. of the Twenty-First Annual Software Engineering Workshop*, SEL-96-002, pages 59-65, Greenbelt, MD, December 1996.

[Basili98] V. R. Basili, F. Lanubile, F. Shull; Investigating Maintenance Processes in a Framework-Based Environment, *In Proc. of the ICSM'98 – International Conference on Software Maintenance*, Bethesda, MD, 1998

[Fowller97] M. Fowller, K. Scott; UML Distilled: Applying the Standard Object Modeling Language, Addison- Wesley, 1997

[Fusaro97] P. Fusaro, F. Lanubile, and G. Visaggio. A replicated experiment to assess requirements inspections techniques, *Empirical Software Engineering Journal*, vol.2, no.1, pp.39-57, 1997.

[Gilb93] T. Gilb, D. Graham. Software Inspection. Addison-Wesley, reading, MA, 1993.

[Jacobson95] I. Jacobson, M. Christerson, P. Jonsson, G. Overgaard; Object-Oriented Software Engineering: A Use Case Driven Approach, Addison-Wesley, revised printing, 1995

[Porter95] A. Porter, L. Votta Jr., V. Basili. Comparing Detection Methods for Software Requirements Inspections: A Replicated Experiment. IEEE Transactions on Software Engineering, 21(6): 563-575, June 1995.

[Shull98] F. Shull. *Developing Techniques for Using Software Documents: A Series of Empirical Studies*. Ph.D. thesis, University of Maryland, College Park, December 1998.

[Singer96] J. Singer and T. C. Lethbridge, "Methods for Studying Maintenance Activities", in *Proc. of 1st International Workshop on Empirical Studies of Software Maintenance,* Nov. 1996, pp. 105-110

[Votta93] L. G. Votta Jr. Does Every Inspection Need a Meeting?. *ACM SIGSOFT Software Engineering Notes*, 18(5): 107-114, December 1993.

[Zhang98] Z. Zhang, V. Basili, and B. Shneiderman, An empirical study of perspective-based usability inspection. Human Factors and Ergonomics Society Annual Meeting, Chicago, Oct. 1998.