Detecting Defects in Object Oriented Designs: Using Reading Techniques to Increase Software Quality

Guilherme H. Travassos[□] Forrest Shull Michael Fredericks Victor R. Basili[*]

Experimental Software Engineering Group and Institute for Advanced Computer Studies
University of Maryland at College Park
A. V. Williams Building, Bldg#115
College Park , MD, 20742
Fax 301 405 3691
{travassos, fshull, fred, basili}@cs.umd.edu

[□]COPPE/UFRJ
C.P. 68511 - Ilha do Fundão
Rio de Janeiro – RJ – 21945-180
Brazil

[*]Fraunhofer Center --Maryland
3115 AgLife Science Surge Bldg
College Park, MD 20742

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 set 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 some elements of these techniques are based on empirical results concerning an analogous set of reading techniques that supports defect detection in requirements documents. We present an initial empirical study that was run to assess the feasibility of these new techniques, and discuss the changes made to the latest version of the techniques based on the results of this study.

Keywords: Software Engineering Practices, Object Testing and Metrics, Object Oriented Software Quality, Software Inspection.

1. INTRODUCTION

Software inspections have been shown to be a practical method of ensuring that software artifacts, created during the software lifecycle, possess the required quality characteristics. For instance, inspections have been used to improve design and code quality by increasing defect removal during development [4]. In this way, inspections help reduce defects in a software system by ensuring that the software artifacts which are necessary for its construction correctly reflect the needs of stakeholders.

Software inspections aim to guarantee that developers deal with complete, consistent, unambiguous, and correct artifacts. Although most commonly applied to code documents, software inspections have also been used at earlier stages of the software lifecycle (e.g. in the requirements phase [1,14]) to detect potential problems as early as possible. Most publications concerning software inspections have concentrated on the best number and organization of inspection meetings while assuming that individual reviewers are able to effectively detect defects in software documents on their own (e.g. [5, 8]). However, there has been empirical evidence that team meetings do not contribute to finding a significant number of new defects that were not already found by individual reviewers [14,17].

"Software reading techniques" attempt to increase the effectiveness of individual reviewers by providing guidelines that can be used to examine (or "read") a given software artifact and identify defects. There is empirical evidence that software reading is a promising technique for increasing software quality for different situations and documents types, not just limited to source code [15]. It can be performed on all documents associated with the software process, and is an especially useful method for detecting defects since it can be applied as soon as the documents are written. In general, defects in software artifacts can be classified as omitted, ambiguous, inconsistent, incorrect, or extraneous information. However, such a list and the definitions

This work was partially supported by UMIACS and by NSF grant CCR9706151.
Dr. Travassos was also partially supported by CAPES-Brazil.

of each of its items obviously need to be tailored to the specific artifact being inspected.

In this paper, we restrict our focus to inspections of high-level Object-Oriented (OO) designs. A high-level OO design is a set of diagrams concerned with the representation of real world concepts as a collection of discrete objects that incorporates both data structure and behavior. High-level designs do not attempt to represent details of either the eventual implementation of the system or the "computational world" in which the system exists. Normally, high-level design activities start after the software product requirements are captured and represented by a textual description and/or use-cases [9]. So, concepts must be extracted from the requirements and described using the OO constructs, such as class, object, inheritance, polymorphism, aggregation, composition, and messaging. This means that requirements and design documents are built at different times and describe the system from different viewpoints and levels of abstraction. When high-level design activities are finished, the documents, basically a set of well-related diagrams, should be read in order to verify both whether they are consistent among themselves and whether they adequately capture the requirements. A design defect occurs if either of these conditions are not met.

This paper discusses some issues regarding the definition and application of reading techniques for high-level object-oriented design documents. The techniques are designed to be used to read requirements, use-cases and design artifacts within a domain in order to identify defects among them. To achieve this goal, our study has been organized as follows:

1. Understand and model relevant aspects of the process for creating OO designs,
2. Define reading techniques that are tailored to OO design,
3. Run an experiment to investigate the feasibility of the reading techniques when used in the inspection of an actual OO design.

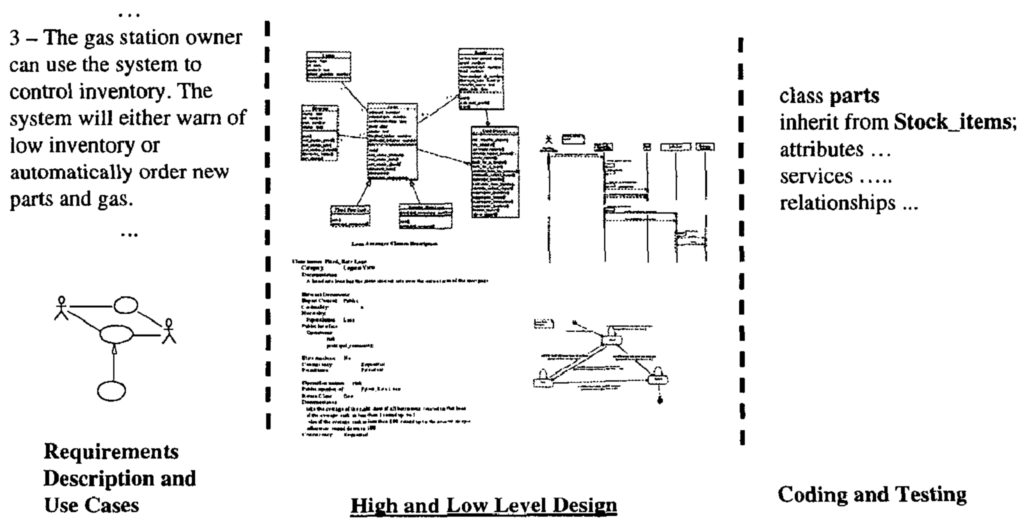
In the next sections, we present an overview of our work at each of these steps.

2. BASIC MODELS OF OO DESIGN

Requirements documents generally consist of a textual description of the functional and non-functional requirements for the software product and the related scenario definitions (i.e. use-cases). The goal of well-described requirements is to represent the problem to be solved by the software system, not to specify a particular solution [13]. In fact, a particular set of requirements could conceivably support multiple solutions for the problem.

In contrast, software design activities are concerned with the description of real world concepts that will be part of the future computational solution of the problem. (This is true regardless of the software lifecycle model used.) 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 (illustrated in Figure 1). 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. (That is, our emphasis is on high-level comprehension rather than low-level details of the architecture.) 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.

To support the construction of the design diagrams we chose UML [6] as the basic notation. UML is considered a notational approach and does not define how to organize development tasks. Therefore, it can be tailored to different development situations. We wanted to focus our reading techniques on the following high-


...
3 – The gas station owner can use the system to control inventory. The system will either warn of low inventory or automatically order new parts and gas.
...

class **parts**
inherit from **Stock_items**;
attributes ...
services .....
relationships ...

Requirements Description and Use Cases
High and Low Level Design
Coding and Testing

Figure 1 – A simplified view of an OO software process

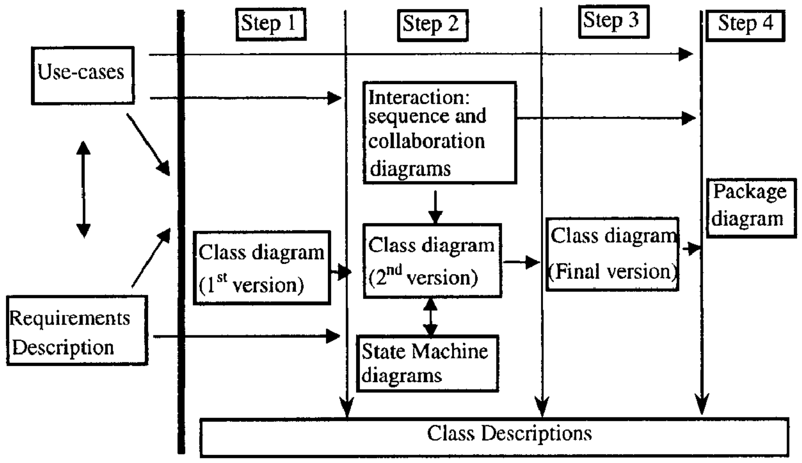

Step 1
Step 2
Step 3
Step 4
Use-cases
Interaction: sequence and collaboration diagrams
Package diagram
Class diagram (1st version)
Class diagram (2nd version)
Class diagram (Final version)
Requirements Description
State Machine diagrams
Class Descriptions

Figure 2 – Interdependencies among high-level design artifacts

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. A class description template was defined to allow developers to write down all the definitions that they used to build the models. This description is basically the data dictionary of the whole design and can be used to support the reading activities. As high-level activities do not deal with the low-level issues regarding the solution, deployment and activities diagrams were not used at this time. The organization of the activities is pictured in Figure 2. The picture does not try to show a sequential process view but illustrates the interdependencies among the design activities; arrows show how some diagrams are used as input to build new ones.

In summary, we make the following assumptions for this work:

- The high-level OO design of a system is composed of a number of separate but related diagrams: class diagrams, class descriptions, interaction diagrams, and state diagrams.
- Separate artifacts also represent the functional view of a system: requirements, use cases, and interaction diagrams.
- Interaction diagrams (e.g. sequence diagrams) contain both functional and design information about a system, and thus can be a useful mechanism for providing traceability between functional and design views. [1]
- Requirements and use-cases are essentially correct at the time of design inspection. Thus we can use the requirements as a reliable basis for the system description. (This assumption is necessary so that we need not compare the design to all possible domain knowledge; we assume that the relevant knowledge has already been correctly captured in the requirements.)
- Requirements are written with no reference to Object-Orientation. (This does not imply that we are faced with a chaotic development process; just that the requirements and design are organized along different principles, as shown in Figure 1.)

[1] The sequence diagrams are a component of the design that reflects information about the functional view of the system; we assume that each sequence diagram reflects the system's behavior during the execution of some functionality.

3. TAILORING READING TECHNIQUES TO OO DESIGN

As discussed before, reading techniques have been shown to be effective for increasing the quality of software artifacts. Artifacts from different stages of the software lifecycle still share a few common characteristics: They must all accurately and completely reflect the system specified in the requirements. They must not contain unnecessary constraints from other domains that influence the solution. They should not contain inconsistencies or ambiguities, so that downstream users of the artifact (i.e. developers at later stages of the lifecycle) can implement the system correctly.

Reading techniques often have the goal of "reading for analysis." While applying reading for analysis a reader attempts to answer the following question: "Given a document, how do I assess its various qualities and characteristics?" Reading for analysis is important for product quality; it can help us to understand both the types of defects we make and the nature and structure of the product [2]. Several families of techniques have applied reading to the analysis of different problem domains, such as:

- Defect-Based Reading (DBR) focused on defect detection in requirements, where the requirements were expressed using a state machine notation called SCR [7,14].
- Perspective-Based Reading (PBR) also focused on defect detection in requirements, but for requirements expressed in natural language [1].
- Use-Based Reading (UBR) focused on anomaly detection in user interfaces [18].

The work presented in this paper deals with further tailoring reading to detect defects in an OO design described by UML diagrams. Specifically, our main interest is to understand the way such documents should be read and describe reading techniques that support these ideas.

In creating a family of reading techniques, a set of models must be defined; the two most important of these are the models of **abstraction** and **use** [15]. The models of abstraction are the models of the important information in an artifact, and how it is organized. For an OO design, the models of abstractions are easy to identify at a high level: the information has already been described in a number of separate models or diagrams (e.g. state machines, class diagrams) as discussed at the end of the previous section. The models of use describe how the information in the artifact is used to detect defects. Before we can describe how to detect OO defects, we need to have some knowledge of the different kinds of defects to be sought. For this purpose, we borrow a defect taxonomy that had proven effective for requirements defects [1], as defined in Table 1 and illustrated in Figure 3. This taxonomy classifies defects by identifying related sources of information, which are relevant for the system being built. For example, the information in the 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 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

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

typically be prevented from appearing in the artifact, since it can only hurt clarity. Any artifact should also analyzed to make sure that it is self-consistent and clear enough to support only one interpretation of the final system. Table 1 shows how the definitions of [1] can be tailored to OO designs.

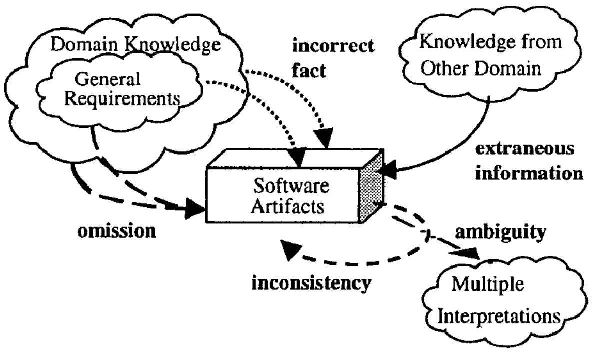


Figure 3 – Types of software defects, and the relevant information sources

The previous work on reading techniques for requirements [1, 14] created techniques that were concerned mainly with checking the correctness of the document itself (making sure the document was internally consistent and clearly expressed, and whether the contents did not contradict any domain knowledge). A major difference in the current study is that for checking the correctness of a design, the reading process must be twofold. As in requirements inspection, the correctness and consistency of the design diagrams themselves must of course be verified (through "horizontal reading"[2]) to ensure a consistent document. But a frame of reference is necessary in order to assess design correctness. Thus it is also necessary to verify the consistency between design artifacts and the system requirements (through "vertical reading"[3]), to ensure that the system design is correct with respect to the functional requirements. Thus we define a new family of reading techniques, which we call *Traceability-Based Reading (TBR)*, due to the fact that the central feature appears to be tracing information between design documents (to ensure consistency) and between design and requirements (to ensure correctness and completeness).

The first version of the reading techniques was strongly influenced by this need for traceability among the diagrams. One reading technique was defined for each pair of diagrams that could usefully be compared against each other. For example, sequence diagrams needed to be compared to state machines to detect whether, for a specific object, there are events, constraints or data (described in the state machine) that could change the way that messages are sent to it (as specified in the sequence diagram). 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 horizontal and vertical reading techniques is defined as illustrated in Figure 4. 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 were evaluated in the feasibility study.

4. AN EMPIRICAL STUDY

4.1. Description of the study

This study was carried out in the context of an undergraduate software engineering course at UMCP during the Fall 1998 semester. The course aimed to teach software engineering principles, which were required to be applied to the development of an application over the course of the semester. The 44 students had a mix of previous experience: 32% had some previous industry experience in software design from requirements and/or use cases, 9% had no prior experience at all in software design, and the remaining majority of the class (59%) had classroom experience with design but had not used it on a real system.

[2] Horizontal reading refers to reading techniques that are used to read documents built in the same software lifecycle phase. (See Figure 4.) Consistency among documents is the most important feature here.

[3] Vertical reading refers to reading techniques that are used to read documents built in different software lifecycle phases. (See Figure 4.) Traceability between the phases is the most important feature here.

However, all students were trained in OO development, UML and OO software design activities as a part of the course. The students were organized in 15 teams (14 teams with 3 students each and 1 team with 2 students) to develop the application, a "Loan Arranger" system responsible for organizing the loans held by a financial consolidating organization, and for bundling them for resale to investors. It was a small system, but contained some design complexity due to non-functional performance requirements.

3) Reading Sequence x State diagrams

For each sequence diagram do:

1. Read the sequence diagram to identify all the actions (messages) that an object is receiving and that change the values of its attributes.
   Underline and number the identified actions (messages) on the sequence diagram with a green pen.
2. Read the state diagram of each object to verify if there is a description of the state and an associated event description.
   Underline and number the identified description of the state and the associated event description on the state diagram with a blue pen and a green pen, respectively.
3. Read the sequence diagram to identify the conditions associated with the events.
   Circle them if they are found. Use the same number given to the associated events on the state diagram. Use a yellow pen.
4. Read the state diagram for each object to look for the description of the conditions.
   Circle them and write in the same number assigned to the condition on the sequence diagram if they are found. Use a yellow pen.

Table 2. Horizontal Reading: Verifying Sequence with respect to State Machine Diagrams (first version)

At the beginning of the project, each team received the requirements for the Loan Arranger application and was asked to undertake a requirements inspection; this inspection helped to improve each team's understanding of the system. From the requirements and a set of use-cases of the system, students were then asked to create an OO design of the system. At the end of the design process, we selected two of the best submissions and asked the teams to inspect one of these designs using the reading techniques. (Two designs were used so that we could ensure that no team inspected a design they had created. However, in order to keep results comparable, all but one team reviewed the same design. Table 3 summarizes the size of this design by reporting for each class the number of attributes, Weighted Methods/Class (WMC), Depth of Inheritance (DIT), Number of Children (NOC), and Coupling Between Objects (CBO). Additionally, there were 3 state diagrams and 5 sequence diagrams; the classes participating in each are marked.) In order to motivate the students to do a good job on the inspection, we gave the students the option of basing their eventual implementation of the system on the design they were inspecting, emphasizing that an effective inspection should lead to fewer problems in the implementation phase.

It should be noted that we had no control group; that is, we could not compare the subjects' effectiveness with the reading techniques to their own (or another segment of the class') effectiveness with another method for inspecting the OO design. This situation was necessary both because we were not aware of any other published methods for reviewing OO designs, and because we were in a classroom environment in which it was not possible to not teach a portion of the class in order to use them as a control group. Thus, we emphasize that this study was only meant to explore the feasibility of the reading techniques, not to evaluate their effectiveness.

Although all the reading techniques shown in Figure 4 could be applied, we identified a subset that was sufficient to guarantee coverage of the whole design and simplified the teams' work. Therefore, the teams used the following set of reading techniques:

- Horizontal Review
  1. Class Diagrams with respect to Class Descriptions
  2. Class Diagrams with respect to State Machine Diagrams
  3. Interaction (Sequence) Diagrams with respect to State Diagrams
  4. Interaction (Sequence) Diagrams with respect to Class Diagrams

- Vertical Review
  1. Class Descriptions with respect to Textual Requirements Description
  2. Interaction (Sequence) Diagrams with respect to Use-cases
  3. State Machines Diagrams with respect to Textual Requirements Description and Use-cases

Each team applied the whole set of techniques, but responsibilities were divided in such a way that each subject needed to deal only with a small number of techniques. So, one subject dealt with the vertical reading, while the second and third divided the horizontal reading techniques between them. After individual review, the students met as teams in order to review their individual lists and to create a final list that reflected a group consensus of the defects in the document.

In order to evaluate the feasibility of the techniques, we collected

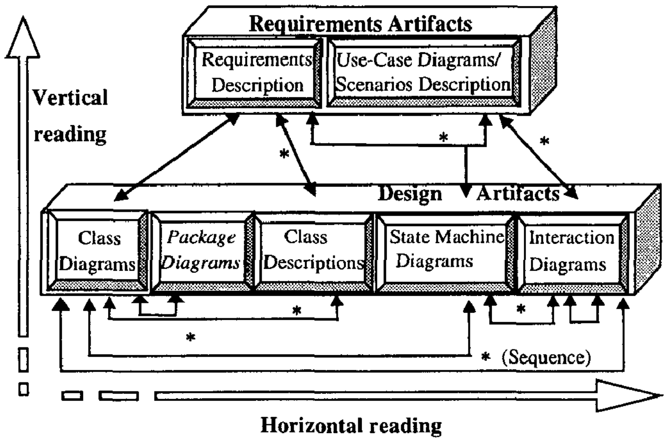


Figure 4 – Reading Techniques

a number of metrics. The subset relevant to our conclusions here is listed in Table 4. To get accurate answers to the subjective questions, we stressed that the students would not be graded on their responses; grades were based only on the degree of process conformance we saw reflected in the students' submissions. In a further effort to produce unbiased answers we made use of post-hoc interviews after the grades for the project had already been returned to the students.

Class Name	Attrs.	WMC	DIT	NOC	CBO	State Dgm. Exists?	Seq1 Contains?	Seq2 Contains?	Seq3 Contains?	Seq4 Contains?	Seq5 Contains?
Property	5	0	0	0	1						
Borrower	2	2	0	0	1						
Lender	3	1	0	0	2		yes				
Loan	3	3	0	2	4	yes				yes	
Fixed Rate Loan	0	1	1	0	4						
Adjustable Rate Loan	0	1	1	0	4						
Bundle	5	2	0	0	2	yes				yes	
Investment Request	4	1	0	0	1					yes	
Loan Arranger	0	15	0	0	4		yes	yes	yes	yes	yes
Financial Org.	1	0	0	0	2	yes	yes	yes	yes	yes	yes
Loan Analyst	1	12	0	0	3		yes	yes	yes	yes	yes

Table 3 – Size measures for the inspected design.

4.2. Partial Results

The primary result of our initial evaluation has provided some evidence that the concept of OO design reading techniques is feasible. Feasibility was assessed both quantitatively, by the number and type of defects reported, and qualitatively, by assessing reviewer satisfaction:

- The techniques did lead to defects being detected: subjects reported on average 11 defects (11.7 for horizontal readers; 10.4 for vertical), although the average time required was approximately 3 hours.
- Real and useful distinctions between horizontal and vertical reading were detected. 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). The difference for ambiguities/inconsistencies was statistically significant (t=2.05, p=0.047) although the difference for omissions/incorrect facts was not (t=0.88, p=0.382). These differences do match what we would expect to see if vertical readers were more focused on checking traceability with requirements while horizontal readers focused more on defects within the design diagrams. Additionally, about half of our subjects found the different types of reading to be worthwhile because reviewers came to the inspection meetings prepared with different defect lists and with expertise in different areas of the document.
- Subjectively, students agreed that using OO reading is worthwhile: the majority of subjects (84%) indicated that if they had to inspect a design document in the future, they would use the techniques they had learned (if only because they knew of no other techniques). There were some difficulties with the techniques, but these did not prevent them from being used: about 80% of respondents reported they had followed the techniques "very closely." There is very little guidance available on how to inspect designs, and any direction we could give students seemed to help them.
- As a secondary measure of user satisfaction, we asked subjects for a subjective evaluation of how well they thought they performed using the techniques. On average, they felt they had found over half (56%) of all defects in the document from individual inspection, and over two-thirds (69%) when they combined to work as teams. (That is, they believed only a minority of all defects in the design still remained after they had performed the inspection.) We do not present these descriptive statistics as measures of effectiveness: our experience from work with requirements inspection has indicated that reviewers often over-estimate their success rate, and our subjects were moreover inexperienced reviewers. However, we do present these numbers as subjective indicators of user satisfaction: our subjects seemed to think the techniques were helpful in finding defects.

When Collected	Metrics
Before the study	a) Details on subjects' amount of experience with requirements, design, and code
After individual review	b) Time spent on review (in minutes) c) Opinion of effectiveness of technique (measured by what percentage of the defects in the document they thought they had found) d) How closely they followed the techniques (measured on a 3-point scale) e) Number and type of defects reported
After team meeting	f) Time spent (in minutes) g) Usefulness of different perspectives (open-ended question)
In post-hoc interviews	h) How closely they followed technique (open-ended question, to corroborate d) i) Were the techniques practical, would they use some or all of them again (open-ended question)

Table 4 – Subset of metrics collected in the feasibility study

This study also uncovered specific needs for improvement in the techniques:

- One problem that was consistently reported was that the students had a difficult time determining the appropriate level of specificity for reporting faults. For example, the design that most teams reviewed contained a relatively weak state diagram. Subjects consistently asked whether they should report this weakness as a single defect or whether they should report every individual case where the state diagram conflicted with another diagram as a defect. If they reported the weakness of the state diagram as a single defect, it would not give the designer much insight as to the problem, but if every conflict between the state diagram and other diagrams were reported, the fault list (and time required to produce it) would be excessive. This particular situation is extreme, but students had similar problems with other defects in the design document. An analogous problem was experienced by subjects in earlier studies of requirements defects [15]; the effects due to requirements defects reported at different levels of detail were investigated in [11].
- Another important area identified for improvement was that the techniques should concentrate more on semantic information. We hypothesize that one major reason for the success of the Perspective Based Reading for requirements [15] is that the reading techniques require significant semantic processing by the reader. The reader must understand the requirements document enough to produce an additional artifact (use cases, test plan, data flow diagram). During the course of the semantic analysis, the reader discovers defects. Our first draft of OO design reading techniques involved verifying that certain words and attributes appearing in one diagram also appeared in the proper places in other sections. These techniques are largely syntactic in nature and do not require that the reader do the same level of semantic processing done in the requirements reading. However, the OO reading techniques might be augmented to ask readers to expand their semantic understanding by creating some other artifact. For example, testing scenarios (using an idea similar to use-cases) as well as actors, actions and constraints regarding a specific functionality can be extracted from the design artifacts [16].

5. ONGOING WORK

As discussed in the previous sections, some areas for improvement have been suggested from our initial study. Additionally, changes are being made to make the techniques more practical. Table 5 illustrates an example of the new version of the technique that addresses some of the concepts discussed in this section.

5.1. Semantic vs. Syntactic Focus

As discussed in section IV.2, this study identified some differences between reading to analyze different types of consistency. The complexity inherent in OO designs requires that both syntactic consistency (e.g. does the same class have the same attributes in different diagrams) and semantic consistency (e.g. do the multiple representations of the same class mean the same thing) must be considered. At the same time, based on the qualitative feedback from the subjects, it is obvious to us that reading techniques that emphasize the syntactic focus are tedious for the reader. Although syntactic consistency is necessary, the results of the feasibility study have motivated us to automate as much of it as possible, on the assumption that human cognitive skills are better applied to the verification of semantic information.

Aside from tool support, we are looking into other strategies for focusing readers' attention on semantic issues. The perspective-based techniques for requirements reading aided analysis of the document by asking the reader to generate a secondary artifact (use case, data flow diagram, etc.) that contained the same semantic information but using a different syntax [1]. During the process of generating this secondary artifact, the reader discovers defects. Active Design Reviews are a different approach to software inspections that focus readers on semantic issues through the use of open-ended questions [10]. Further feasibility studies are necessary to investigate whether these strategies can be effectively adapted to OO inspections.

5.2. Improving the Mechanism for Identifying Defects

Although this study demonstrated that reading techniques could feasibly be applied to detect defects in OO designs, the subjects reported that the specific steps of the techniques turned out to be a bit confusing when applied by hand. All the techniques we had defined guided the reader to use a certain marking system to highlight important concepts in the diagrams, then detect defects by identifying discrepancies among the markings. However, as the amount of defects increased, the markings became too confusing to help readers find specific problems. We hypothesize that this problem might be avoided if we asked readers to construct new artifacts (as was used in the techniques for requirements reading and as was discussed briefly in section V.1) rather than mark up existing artifacts.

5.3. Improving the Traceability Back to the Requirements

A strong result of the feasibility study was that subjects reported many difficulties when tracing design concepts back to the relevant requirements. This was true despite the fact that we used a relatively small requirements document and asked the subjects to apply reading techniques (which had elsewhere proven effective [15]) to analyze the requirements before beginning the experiment. We observed the following specific issues:

1. Reviewers in our study did have not strong experience in organized development. That is, most reviewers had built systems before but not using an organized software process. Usually, these reviewers built software using a short description of requirements and trying to code immediately, avoiding a more formal design process. As they didn't have source code in hand in this study, it was a bit more complicated for them to associate design solutions to the requirements.

6) Reading 6 – Sequence Diagrams x Use-cases

1) Take a **use-case** and read it to identify the functionality and the nouns that it includes. For each noun identify actions (behaviors) and possible information (data) exchanged with the other nouns. Mark the order (sequence) of the actions. Look for the condition that activates behaviors or actions.

☞ Underline and number the nouns with a blue pen as they are found

☞ Underline and number the identified actions (behaviors) in the order of actions with a green pen

☞ Label the information (data) in yellow as "Dij" where subscripts i and j are the numbers given to the nouns between which the information is exchanged.

2) Read the **sequence diagram** to identify if the corresponding functionality is represented and whether behaviors and data were represented in the right order. Number the actors and classes you have found using the same order you used in the use-case. Apply the same approach you used to mark the behaviors and data exchanges.

Could you find the same nouns in this sequence diagram? Are they showing up in the same order you had marked on the use case? Does this sequence diagram completely represent the use-case? Are some of the nouns identified in the use-case missing from this sequence diagram? How were these nouns represented (actors, classes, or attributes)?

☞ Mark the nouns on the use-case that are represented in the sequence diagram with a blue symbol (*).

☞ Mark the nouns in the sequence diagram with a blue symbol (*) if they are represented in a different order than on the use-case or if they appear only on the sequence diagram.

Can you identify the specified behaviors in this sequence diagram? Are the classes/objects exchanging messages in the same order that you observed on the use-case? Were the data that messages are carrying forward described by the use-case? Are the appropriate constraints being observed in this sequence diagram? Can you understand the expected functionality just by reading the sequence diagram? Are some details from the use-case missing here?

☞ On the sequence diagram, mark the behaviors with a green symbol (*) and data with a blue symbol (*)if they are not represented in the same order as in the use-case.

☞ Mark the constraints on the use-case with a yellow symbol (*) if they are not represented in the sequence diagram.

3) Read the use-case and sequence diagram to look for what is wrong between the two documents.
Is there an unmarked noun in the use case? If yes, it means that a concept was used to describe functionality but not represented on the sequence diagram. You have probably found an omission in the sequence diagram. Fill in a record to describe the problem.
Is there any noun on the sequence diagram that is marked? Does it not appear in the use-case? If yes, it means that a concept was used to describe functionality but not in the same way that was used on the use-case. You have probably found an incorrect fact. Otherwise, you have found an extraneous noun on the sequence diagram. Fill in a defect record describing the problem.
Has some behavior or data in the use-case been marked? Do the classes exchange messages in the same specified order? Are all data being sent in the right message? Were the constraints observed? Could you find some behavior or data in the sequence diagram that was not relevant to the use-case? If yes, it means that the sequence diagram is using information incorrectly. Fill in a defect record describing the problem.

Table 5 – Vertical Reading: Verifying Sequence Diagrams with respect to Use-Cases

2. These difficulties seem to contradict the expectation expressed in some work (e.g. [3, 12]) that the use of OO would help reduce the semantic gap between these phases of the software lifecycle. It appears instead that more effective strategies for uncovering the traceability between requirements and design must be developed before effective OO inspections can take place, at least by reviewers as inexperienced as our subjects. Moreover, this issue is also relevant if reviewers have experience with the problem domain and software development but little experience with the way in which requirements are captured by the design and potentially distributed among multiple classes.

5.4. Scaling Up to Larger Systems

The goal of any inspection is to check whether the entire system fulfills the required quality properties. Achieving this goal becomes more difficult as the system becomes larger. This stems from the fact that the entire system cannot be checked in one inspection because of its size and inherent complexity. Hence, several inspections need to be performed and each inspection needs to focus on some subset of the entire system.[4]

There are multiple ways of subdividing a system into units of inspection. We identified the following options for performing the subdivision for OO design reading:

A. Each inspection is focused on some subset of the system functionality (i.e. the system is divided up according to

[4] We assume that ultimately the entire design will be inspected. In practice, that may not be a realistic assumption since an organization may choose to invest in inspections for only portions of the system where the payoff is expected to be greater. However, for now we address only the question of what are useful subdivisions in an OO system, without recommending which subdivisions should be inspected.

concepts appearing in the requirements, use cases, or interaction diagrams).

B. Each inspection is focused on some subset of the "conceptual entities" found in the design (i.e. the system is divided up according to concepts appearing in the class diagrams, class descriptions, interaction diagrams, or state diagrams).

The similarities and differences between options A and B are important to understand. In both cases, the goal is to produce a unit of inspection that is self-contained and conceptually whole. In both cases, it is recognized that the mapping between requirements and design is nontrivial. This is because of our earlier assumption (in section II) that the requirements present a functional view of the system while the design presents an Object-Oriented view. Mapping from OO to a functional description is hard because it is rare that a set of functional requirements will describe entirely and only the functionality of a class. Rather, classes have to be designed by thinking about a reasonable design for the system as a whole, and so usually involve parts of many different types of functionality that may not be obviously related. Mapping from functionality (expressed as use cases) to OO is hard because one use case typically involves the interaction of multiple classes and behaviors in the OO design; it is not typical for the set of operations in one use case to be encapsulated in one class.

By inspecting only a subset of the components of a system in any inspection, we have the risk of missing some high-level functionality. Options A and B suffer from this risk in different ways. When the inspection is based on the functional description (Option A), it is relatively easy to assess whether all the relevant higher-level functionality has been represented correctly, but harder to detect whether a particular class contains enough behaviors concerned with that functionality. When the inspection is based on design concepts (Option B), the situation is reversed. Saying that Option A or Option B is appropriate for every type of system makes as little sense as saying that every system architecture should be "pipe-and-filter". Different options are suitable for different types of systems. The objective should be to apply either Option A or Option B to organize the inspections for a particular system, minimizing the more important type of defect for that system.

These issues highlight for us the necessity of developers using some mechanism to capture the evolution of system requirements from requirements descriptions to source code. Therefore we are now addressing how to represent requirements (both functional and non-functional) and their associated use-cases in order to improve the traceability to OO.

6. CONCLUSIONS

In this paper, we have described a set of techniques for reviewing OO designs. We have applied them in an initial study that has demonstrated their feasibility and provided specific indications for future improvement.

Where possible, we tried to base the OO reading techniques on lessons learned from analogous techniques applied to requirements documents. Some similarities between the two techniques were by design; for example, we adapted a taxonomy of defects that had been useful for requirements reading to focus the design reading on important areas. As a result of the study, however, we noted other similarities between the two reading techniques. For example, we found that reviewers had difficulty finding the "right" level of detail for expressing design defects usefully. This result mirrors difficulties that subjects had encountered while inspecting English-language requirements but which, due to the well-defined notation in which the entire design was expressed, we had not expected to be relevant. Results from this study also may indicate that an effective strategy for focusing reviewers on semantic aspects of a design is the construction of new artifacts containing some of the same information as the document being reviewed. If this indication is confirmed in further studies it would represent another similarity with the results of requirements reading.

However, we did discover crucial differences in design reading that will have to be accounted for in future versions. For example, in requirements reading, syntactic verification is much less important than semantic; the real focus of the inspection is on verifying the content. When applied to OO designs, however, syntactic reading becomes much more important, due to the number of separate but inter-related diagrams that must be kept consistent. At the same time, this study has confirmed that syntactic reading can be tedious and should be automated where possible.

A second important result is concerned with the definition of vertical and horizontal reading. The feasibility study has provided evidence that the use of each type can influence the types of defects found, and moreover, that both types are necessary to effectively find all defects in the design.

At this time, we are working on incorporating all of these findings into an improved version of the techniques for further study. We have designed and are putting together a lab package so interested researchers can review the techniques and our other experimental artifacts in more detail. Interested readers can find an initial lab package and the current set of techniques at http://www.cs.umd.edu/projects/SoftEng/ESEG/manual/tbr_package/.

7. ACKNOWLEDGMENTS

Our thanks to Dr. Yong Rae Kwon for his invaluable assistance designing and running this experiment as a part of the course CMSC 435 (Fall 1998) at University of Maryland, College Park. Our thanks also go to the students of CMSC 435 for their cooperation and hard work.

8. REFERENCES

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

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

[3] Coad, P. and Yourdon, E.. *Object-Oriented Analysis*, 2nd ed. Englewood Cliffs, NJ. Prentice Hall. 1991.

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

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

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

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

[8] Gilb, T., Graham, D.. *Software Inspection*. Addison-Wesley, Reading, MA, 1993.

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

[10] Knight, J., E. Myers, A.. An Improved Inspection Technique. *Communications of the ACM*, 36(11): 51-61, November 1993.

[11] Lanubile, F., Shull, F., and Basili, V.. Experimenting with Error Abstraction in Requirements Documents. In *Proc. of the Fifth International Symposium on Software Metrics*, Bethesda, MD, November 1998.

[12] Meyer, B.. Object Oriented Software Construction, Second Edition, Prentice Hall Inc., 1997.

[13] Pfleeger, S. L.. *Software Engineering: Theory and Practice*, Prentice Hall Inc., 1998.

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

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

[16] Vieira, M. E. R., Travassos, G. H.. An Approach to Perform Behavior Testing in Object-Oriented Systems. In Proceedings of TOOLS Asia'98, Beijing, China, September 98

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

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