National Research Council Canada
Conseil national de recherches Canada

Institute for Information Technology
Institut de Technologie de l'information

ERB-1077

NRC·CNRC

Using Inspection Technology in Object-oriented Development Projects

Oliver Laitenberger, Colin Atkison, and
Khaled El Emam
June 2000

Canada

NRC 44143

National Research Council Canada

Conseil national de recherches Canada

Institute for Information Technology

Institut de Technologie de l'information

Using Inspection Technology in Object-oriented Development Projects

Oliver Laitenberger, Colin Atkison, and Khaled El Emam

June 2000

Using Inspection Technology in Object-Oriented Development Projects

Oliver Laitenberger , Colin Atkison
Fraunhofer Institute for Experimental Software Engineering
Sauerwiesen 6
67661 Kaiserslautern, Germany
+49 6301 707 200
{Oliver.Laitenberger, Colin.Atkinson}@iese.fhg.de

Khaled El Emam
National Research Council, Canada
Institute for Information Technology
Building M-50, Montreal Road
Ottawa, Ontario, Canada K1A OR6
Khaled.El-Emam@iit.nrc.ca

ABSTRACT
Software inspection is a proven approach for detecting and removing defects immediately after software documents are created. However, the advance of software technologies, processes, and methods, such as the widespread adoption of object-orientation, raises new problems regarding software quality assurance with inspections. These primarily relate to the question of how managers can organize a software inspection in object-oriented development projects with respect to the examined documentation and, once it has been organized, how developers can perform the defect detection activity in a systematic manner. This paper presents the architecture-centric strategy for inspection organization and the perspective-based reading technique to address the two problems. The integration of these approaches in the inspection approach allows practitioners to set up and run cost-effective inspections in their object-oriented development projects. To support this claim with quantitative findings, this paper presents the results of a controlled experiment to determine the feasibility and cost-effectiveness of the approaches when used for the inspection of UML-based design documents.

Keywords
Software Quality Assurance, Inspections, Unified Modeling Language, Empirical Study

1 INTRODUCTION

A software inspection involves activities in which qualified personnel should find the most defects in a cost-effective manner. Despite the large body of inspection experience accumulated over the last 20 years, the increasing adoption of object-oriented development principles in the software industry raises new issues regarding software quality assurance with inspections. These issues require further work on the question of how managers can organize a software inspection with respect to the examined documentation and, once it has been organized, how developers can perform the defect detection activity in a systematic manner.

The importance of the first question stems from the fact that today's software development projects face many challenges due to their scale. They usually involve a large number of developers who are all working together to manufacture a single product. In doing so, the developers create documents that together describe what the software system is to do and how it does it. These documents, once they are created, may consist of hundreds of pages that obviously cannot be handled in a single inspection. Object-oriented development principles. Particularly, the recent publication of the Unified Modeling Language (UML) accentuates this problem, since crucial information about the software is distributed across several documents and diagrams. Hence, the challenge consists of organizing several inspections on the right document fragments. However, this requires a rationale for partitioning and grouping the documents and diagrams.

Once an inspection has been organized, the second question addresses the process that inspectors follow to scrutinize the selected document fragments for defects. In this inspection phase, inspectors read the documentation to determine whether quality requirements, such as correctness, consistency, testability, or maintainability, have been fulfilled. "Reading" implies the systematic examination of the software documents to extract, gain, and understand certain information about the software. Many existing inspection implementations assume that inspectors have the reading skills required for defect detection. However, these skills are seldom developed in any systematic manner in the education or training of software professionals or students. Adequate technical support for inspectors during defect detection can compensate for the lack of reading skills and, thus, potentially result in dramatic improvements in the cost-effectiveness of an inspection. Therefore, not only are more procedural reading techniques needed to alleviate this problem, but they must also be tailored to the inspection organization strategy.

In this paper, we present a new approach to inspection organization. The new strategy is based on the principle of organizing inspections around logical entities from the software architecture rather than around particular (types of) documents. The choice of the logical entities determines the document fragments that contain relevant information to be scrutinized for defects in one particular inspection. Although this approach, dubbed architecture-centric inspection organization, is generally applicable for inspection organization, we consider it particularly valuable for the inspection of documents developed according to object-oriented principles.

To address the second problem, we discuss reading techniques that inspectors use during the defect detection activity of an inspection. One of the reading techniques - perspective-based reading (PBR) - will be explained in more detail, since this technique represents a natural complement to the architecture-centric strategy for inspection organization and, at the same time, can be tailored for defect detection in any kind of object-oriented documentation. To demonstrate the cost-effectiveness of this technique, we describe a controlled experiment to compare the checklist-based reading approach (CBR) for defect detection in UML design documents to the perspective-based reading technique. The results of this experiment indicate that inspection teams discovered, on average, 58 percent of the defects in a software document using PBR and 43 percent using CBR. Moreover, while PBR teams exhibit an average cost per defect ratio of 56 minutes per defect, CBR teams exhibit an average cost per defect ration of 132 minutes per defect. In this way, inspection teams using PBR for defect detection have a higher effectiveness than CBR, as well as a lower cost per defect ratio than those applying CBR.

The remainder of this paper is organized as follows. Section 2 presents a short description of the inspection methodology. It explains the process and defines the terminology. Section 3 describes the architecture-centric strategy for inspection organization. Section 4 elaborates upon reading techniques for software inspections and perspective-based reading. Section 5 explains a controlled experiment to validate this approach. Section 6 concludes.

2 INSPECTION PRINCIPLES

Software inspection is an approach that allows the detection and removal of defects immediately after software documents are created. Since the seminal introduction of the generic notion of inspection to the software domain in the early 1970s [9], it has evolved into one of the most cost-effective methods for early defect detection and removal [18]. Its proponents claim that inspections can lead to the detection and correction of anywhere between 50 and 90 percent of defects [10]. Moreover, rework cost can be reduced considerably, since defects are typically found directly after they are introduced. Finally, early defect detection and removal improves the predictability of software projects and helps project managers stay within schedule, since problems are unveiled throughout the early development phases and costly rework cycles at the end of the development or maintenance project are therefore avoided. Considering the many benefits and the fact that low defect density is not one of the strong points of the object-oriented paradigm [13], object-oriented methods would benefit enormously from a systematic inspection method.

An inspection involves activities in which qualified personnel determine whether software documents are of sufficient quality for subsequent development activities. Figure 1 illustrates a typical software inspection approach.

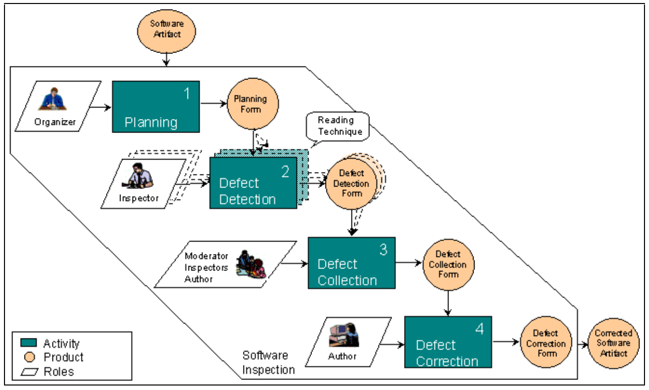


Figure 1: A Software Inspection

In this paper, we model an inspection in terms of its main activities, that is, inspection planning, defect detection, defect collection, and defect correction. There are two reasons for using this terminology. First, we want to emphasize that the problems and solutions we present in this paper are independent of any particular inspection implementation, such as that of Fagan [9] or Gilb and Graham [10]. Second, by modeling the inspection process in terms of its main activities we avoid ambiguity in the terminology.

Inspection planning is performed by an organizer who schedules all subsequent inspection activities. The organizer is often the project manager, since he or she organizes all activities for the software development project. The defect detection and defect collection activities can be performed either by inspectors (i.e., developers) individually or in a group meeting. Since recent empirical findings reveal that the synergy effect of inspection meetings is rather low in terms of defects [11], [21], [24], defect detection should be considered an individual rather than a group activity. In this case, the individuals must receive guidance in the form of systematic defect detection or reading techniques. Defect collection, on the other hand, is often performed in a team meeting (i.e., an inspection meeting) lead by an inspection moderator. The main goals of the team meeting are to agree on anomalies that inspectors have detected individually, to eliminate false positives, and to specify the defects for correction. Since a team meeting is effort consuming and since some development projects are performed at different sites or even in different countries, a synchronous team meeting may be replaced with another form of consolidation activity, such as depositions [24]. An inspection usually ends with the correction of the documented defects by the author.

3 ARCHITECTURE-CENTRIC INSPECTION ORGANIZATION

This section addresses the question of how to organize an inspection with respect to the examined documentation.

3.1 The Unit of Inspection

Before performing an inspection, the inspection organizer must determine the subject of the inspection, that is, the unit to be inspected. The term "unit" refers to the set of information that inspection participants scrutinize for defects in one particular inspection. The selection of the "right" unit represents a problem in the software domain because large software development projects have special problems due to their scale. The volume of the developed documentation is just too large to be handled in a single inspection. Hence, several inspections must be organized on different document fragments. However, this requires a rationale for partitioning and grouping them. This problem is comparable to the situation in other engineering disciplines. The various plans for a bridge, for example, cannot be inspected in a single inspection. Hence, they need to be partitioned and grouped into smaller units for which an inspection can be organized and performed.

Most existing inspection variations follow a document-oriented strategy for inspection organization. This strategy means that a particular inspection is organized around a particular type of document, such as a requirements, design, or code document (or parts of it), rather than the structure or content of the information represented in the documents.

Although at first sight the document-oriented approach appears to be a good strategy, it leads to two difficulties. First, crucial information is often distributed across various parts of a document or even across different document types. Thus, if the inspection is limited to a particular (part of a) document, an inspector may miss crucial information for a sound inspection. Object-oriented development methods [4], [6], [23] and the recent appearance of the Unified Modeling Language (UML) accentuate this problem because they usually use different types of diagrams to represent various sets of information. Hence, information about a given logical entity, such as a class or an object, can be described in many different documents, and a specific document can contain information about many different logical entities, that is, there is a many-to-many relationship between logical entities and diagrams. An inspection whose goal is to check a particular (part of a) document may end up either having to analyze many logical entities, or may only partially cover a logical entity that it describes.

The second difficulty arises when the document or parts of it is still too large after decomposing it. Hence, some authors recommend the use of size information alone as a further splitting criterion [8]. However, this recommendation causes some problems as the following example illustrates. Let us assume a code document (as part of a larger system) has 20000 LOC after following the document-oriented strategy for inspection organization. As suggested in the literature [8], this partition obviously is still too large to scrutinize for defects in a single inspection. Hence, the document needs to be partitioned further. One approach would be to split it into partitions of 500 LOC, which is recommended in the literature [10], and inspect the first 500 LOC in the first inspection, the next 500 LOC in the second inspection, and so on. However, the first 500 LOC provides information about the definition and declaration of variables, which are a necessary prerequisite for inspecting the other code fragments. This example demonstrates that the document-oriented approach and the use of size information alone to decide upon the unit of inspection do not solve the unit of inspection problem in each and every case.

3.2 Architecture-centric Principle

Software is unique among engineering products in that, strictly speaking, it is invisible and has no concrete material manifestation [5]. Whereas a civil engineer, for example, can inspect both the documentation of a bridge and the actual elements of it, or a mechanical engineer can inspect the documentation of an engine as well as the physical parts that he or she builds, a software engineer cannot actually look at a piece of a software system per se. He or she can only inspect the representations, the descriptions, or the documentation of it (or parts of it). This observation leads to a new solution to the unit of inspection problem.

The new solution distinguishes between a logical entity and the physical documentation of the entity. This situation is graphically depicted in Figure 2.

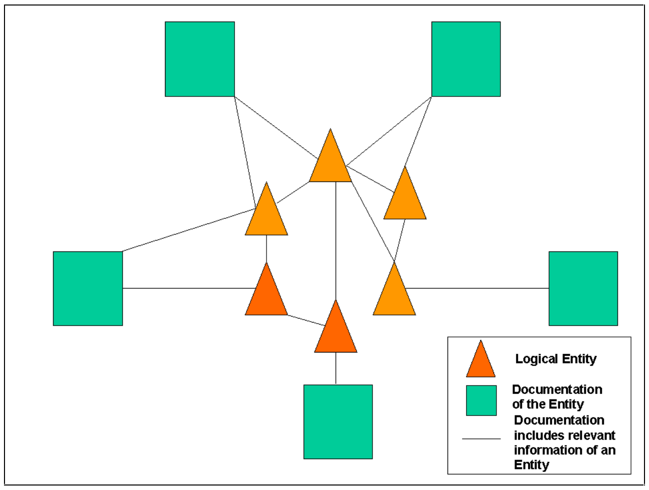

Logical Entity
Documentation of the Entity
Documentation includes relevant information of an Entity

Figure 2: Logical Entities and their Documentation

A triangle, or a line between two triangles, represents a logical entity. A square, on the other hand, represents the documentation of one or more logical entities. The new strategy suggests that an inspection be organized around logical entities rather than the physical documentation of the logical entities. The choice of the logical entity or the logical entities determines the document fragments that contain relevant information to be scrutinized for defects in one particular inspection.

Although the majority of modern software engineering methods, such as the Unified Process [14] or the Object Modeling Technique [23], explicitly separate logical entities from their documentation, this has never been considered useful for inspection organization. Of course, each method uses its own terminology, which may be different from the one used in this paper. The Unified Process, for example, distinguishes between structural elements, such as subsystems or classes, and models that describe the structural elements, such as use-case or collaboration diagrams [14]. Hence, the Unified Process uses the term "structural element" instead of logical entity and "model" instead of documentation of the logical entity. We recognize that various terms could have been used in this paper to describe this difference. Examples are "unit" or "concept" instead of "logical entity" and "model", "representation", or "description" instead of documentation. We decided to use the word "logical entity", since it best conveys the conceptual and invisible nature of software, and "documentation", since it best conveys the idea of something tangible that can be used for the purpose of inspection.

In a more general sense, the logical entities making up a system, and the relationship between them, are collectively viewed as the architecture of the system. The architecture encompasses the significant decisions about the organization of a software system, the major structural elements and their relationship that will comprise the system, and the composition of the elements into progressively larger subsystems. Due to the importance of the architecture, we call the new strategy the architecture-centric approach for organizing inspections.

3.3 The Benefits of the Architecture-centric Solution

The architecture-centric solution for inspection organization is beneficial for three reasons: First, the set of information for each logical entity by definition is logically self-contained and conceptually complete. It therefore provides an inspector with all crucial information for performing a sound inspection and, at the same time, represents the appropriate set of information that is intellectually manageable. The latter prevents inspectors from being swamped with a lot of unnecessary information. Second, the architecture-centric approach is scalable. If the documentation of a logical entity is still too large, an inspection organizer can look at the substructure of the logical entity and choose an appropriate logical entity of smaller granularity. This process can be repeated until the right scope for a single inspection is determined. Finally, the architecture-centric approach has been implicitly embedded in conventional structured development processes that use decomposition as a principle for structuring software systems. The most prominent example for such a process is the Cleanroom Process [15]. However, the use of the architecture-centric solution is not limited to conventional design principles. It can also be applied in the context of object-oriented development methods. This is important because over the past decade, object-oriented development methods have replaced conventional structured approaches as the embodiment of goodness in software development, and are now the approach of choice in most new software development projects. Software inspections must be tailored to this new situation.

3.4 Example: Architecture-centric Inspection Organization in the Unified Process

The Unified Process (UP) proposed by Jacobson, Rumbaugh, and Booch [14], is a generic process framework that can be specialized for a very large class of software systems. A specific instance of it is the Rational Unified Process. The UP is component-based, which means that the software system being built is made up of software components. The distinguishing aspects of the UP are captured in three key phrases – “use-case driven”, “architecture-centric”, and “iterative and incremental”.

For organizing an inspection, the use-case driven and architecture-centric properties of the UP are the most important. In the UP, use-cases are the driver for the architecture. The knowledge of the architecture in turn helps capture the requirements as use-cases. Hence, the development of both the use-cases and the architecture can be regarded as an iterative process.

Architecture-centric software inspections in the context of the UP can be organized around components, their interfaces, and their interactions. Components as seen from the development point of view are subsystems that have high internal cohesion and low external coupling and are reusable by other developers. A component as part of the architecture is best represented by multiple, coordinated architectural views. An architectural view is an abstraction of a use-case, design, implementation, process, and deployment model that focuses on its structure and essential elements. If, in the context of the development project, some of the models are not yet available, the inspection can be organized with the available models. For example, at the beginning of the project, there may only be the use-case model in the form of use-case diagrams as well as the design model in the form of class diagrams.

An example of the architecture-centric approach in the context of the UP is depicted in Figure 3, in which the top-level component, i.e., the system, is partitioned into two sub components. After partitioning, an inspection can be organized for each of the subcomponents.

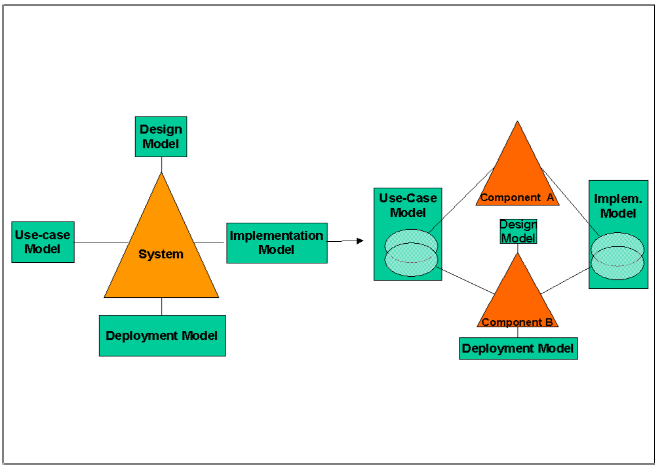


Figure 3: Architecture-centric Inspection Organization for the Unified Process

Figure 3 depicts the documentation of a software system. It consists of a use-case model, a design model, an implementation model, and a deployment model. Under the assumption that the size of the documentation is too large to handle it in one inspection, the documentation needs to be partitioned. A document-oriented strategy would suggest performing an inspection for the use-case models, the design models, the implementation models, and the deployment models. The architecture-centric strategy, on the other hand, considers the logical structure of the system for partitioning. In this example, the system consists of two components: Component A and component B. Hence, the architecture-centric principle suggests performing an inspection for the documentation of component A (subset of the use-case model and the implementation model) and an inspection for the documentation of component B (subset of the use-case model and the implementation model, design model and deployment model).

4 READING TECHNIQUES FOR DEFECT DETECTION IN INSPECTION

This section addresses the problem of how to provide defect detection support for inspection participants.

4.1 The Lack of Systematic Reading Techniques for Defect Detection

Although each of the presented inspection activities is important for a successful inspection, the most important one is the defect detection activity. In this phase of an inspection, inspectors read the software document(s) to determine whether quality requirements, such as correctness, consistency, testability, or maintainability, have been fulfilled. "Reading" implies the systematic examination of a document to extract, gain, and understand certain information about the inspected software. The ability to read, and to understand what has been read, are therefore critical skills for the participant of an inspection. Understanding itself is a necessary prerequisite for finding more crucial defects in the software documentation. These defects are often the expensive ones if detected in later development phases, and the most difficult to detect in an inspection, since they usually go well beyond more trivial defects, such as spelling mistakes.

4.2 Existing Reading Techniques

In practice, most industrial inspection implementations use either no specific reading approach (often termed ad-hoc) or checklist-based reading (CBR) during defect detection [9], [10]. Ad-hoc reading, as its name implies, provides no explicit advice for inspectors as to how to proceed, or what specifically to look for, during the reading activity. Hence, the results of the reading activity in terms of potential defects or problem spots are fully dependent on human experience and expertise. Checklists offer stronger support mainly in the form of yes/no-questions that inspectors need to answer while reading a software document. Gilb and Grahams' manuscript on software inspection states that checklist questions interpret specified rules within a project or an organization [10]. Although a checklist provides advice about what to look for in an inspection, it does not describe how to identify the necessary information and how to perform the required checks. Moreover, for CBR as well as for ad-hoc it remains unclear as to what degree a systematic reading process was applied.

Recently, Vic Basili proposed scenario-based reading [1] to offer more procedural support for defect detection. The basic idea of a scenario-based reading technique is the use of so-called scenarios. A scenario can be defined as an algorithmic guideline for the inspector that describes how to go about finding the required information in a software document, as well as what that information should look like. Hence, a scenario-based approach is more prescriptive than either the ad-hoc or the checklist-based technique. A particular promising scenario-based reading technique is perspective-based reading [2] [15].

4.3 Perspective-based Reading

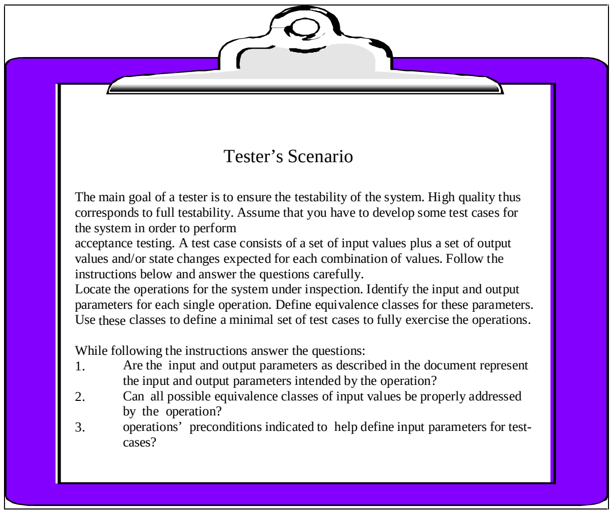
Goal of Perspective-based Reading

The basic goal of PBR is to examine the documentation of a software entity from the perspectives of the entity's various stakeholders for the purpose of identifying defects. An inspector in a perspective-based inspection reads the documentation from the perspective of a particular stakeholder in such a way as to determine whether it satisfies the stakeholders' particular needs. A stakeholder perspective may be, for example, a future user of the system who wants to ensure the completeness of the inspected analysis documents. If the documentation of the software entity meets the stakeholders' quality requirements, the end product, that is the final software system will meet the specified quality goals. The reading process itself is driven by a perspective-based reading scenario.

Perspective-based Reading Scenarios

Throughout the reading process, an inspector follows the instructions of a perspective-based reading scenario (in short: scenario). A scenario tells the inspector how to go about reading the documentation from one particular perspective and what to look for.

A scenario consists of an introduction, instructions, and questions framed together in a procedural manner. The introductory part describes the stakeholder's interest in the logical entity and explains the quality factors most relevant for this perspective. The instruction part describes what kind of document an inspector is to use, how to read the document, and how to extract the appropriate information from them. While identifying, reading, and extracting information, inspectors may already detect some defects. However, the motivation for providing guidance for inspectors in the form of instructions on how to perform the reading activity is three-fold. First, instructions help an inspector gain a focused understanding of the entity. Understanding involves the assignment of meaning to information in a particular document and is a necessary prerequisite for detecting more subtle defects which are often the expensive ones if detected and removed in later development phases. Second, the instructions require an inspector to actively work with the documentation rather than passively scanning it. Third, the architecture-centric strategy ensures that the relevant information for all stakeholders is available for scrutiny. However, since the attention of an inspector is focused on the information most interesting for a particular stakeholder, the inspector is not swamped with details irrelevant for the stakeholder's perspective. A process for scenario development is described in [15], [19].



Tester's Scenario

The main goal of a tester is to ensure the testability of the system. High quality thus corresponds to full testability. Assume that you have to develop some test cases for the system in order to perform
acceptance testing. A test case consists of a set of input values plus a set of output values and/or state changes expected for each combination of values. Follow the instructions below and answer the questions carefully.
Locate the operations for the system under inspection. Identify the input and output parameters for each single operation. Define equivalence classes for these parameters. Use these classes to define a minimal set of test cases to fully exercise the operations.

While following the instructions answer the questions:

1. Are the input and output parameters as described in the document represent the input and output parameters intended by the operation?
2. Can all possible equivalence classes of input values be properly addressed by the operation?
3. operations' preconditions indicated to help define input parameters for test-cases?

Figure 4: Reading from a tester's perspective.

Once an inspector has achieved an understanding of the documented information about an entity chosen by the architecture-centric approach, he or she can examine and judge whether it fulfils the required quality properties. For making this judgement an inspector is supported by a set of questions that are answered while following the instructions. Figure 4 shows an example for reading from the perspective of a tester.

5 AN EMPIRICAL STUDY

The architecture-centric inspection organization and the PBR technique are not silver-bullet techniques [5], that is, they do not cure every inspection-related problems in all industrial settings. As a consequence, the theoretical conditions under which these approaches help an inspection team detect the most defects most rapidly need to be identified and examined empirically. Such expectations or conditions may include, for example, statements such as the following: "when developers have little experience with the document type, the PBR technique is more effective than a traditional form of reading". Hence, different conditions need to be studied empirically to identify the techniques' strengths and weaknesses. The level of control imposed on the conditions determines whether a study qualifies as a case study, quasi-experiment, or controlled experiment [7]. The results of the studies help researchers gain an understanding of how a technique works and why the technique is useful. Practitioners, on the other hand, benefit from those studies because the results help them assess the leverage they can expect from a particular technique. This may influence their decision as to whether and how to adopt it in their projects. In this section, we describe the essence of a study to compare perspective-based reading (PBR), for defect detection in object-oriented design documents (documentation) of software systems (logical entity) using the notation of the Unified Modelling Language (UML) to the more traditional checklist-based approach (CBR). The experiment is described in full detail in [17].

5.1 Goal of the Study

We focus our evaluations on two important aspects of software inspections in object-oriented development: their effectiveness and their cost[1]. Effectiveness is defined as the proportion of defects in the document that were found during an inspection. Cost is defined in terms of the effort involved in finding a single defect. Effort is the most important factor in determining the cost of a software inspection.

We consider in this experiment the team results as our unit of analysis for the following reason. When using CBR, individual subjects do not adopt a particular perspective while reading the documents, whereas they do when they

[1] Another evaluative criterion of software inspections is their interval (i.e., elapsed calendar time) [24]. However, this is not addressed in the study.

are implementing PBR. With a perspective, a subset of the defects in the document have a high probability of being detected, while the remainder of the defects have a relatively low probability of being detected by that perspective [19]. Conversely, with CBR one would expect more uniformity in the probability of detection across defects. This reasoning makes it clear that we do not necessarily expect individual PBR inspectors to be more cost-effective than individual CBR inspectors. Rather, we expect the benefits of PBR to become apparent at the overall team level. Indeed, it is the team result that determines the results of an inspection. One may argue that individual variability influences the team results. However, in the context of our experiment, individual variability is controlled by random assignment of subjects to teams. We can therefore state the following expectations for the experiment:

1. **The Effectiveness of PBR is Greater than the Effectiveness of CBR for Teams.**
   We expect that inspection teams detect more defects using PBR than CBR. This results from the fact that the probability of finding a unique defect through PBR is greater than for CBR [19].
2. **Overall Inspection Cost is Lower with PBR than with CBR.**
   We would expect that the overall cost per defect for both phases to be smaller for PBR than for CBR.

Based on our expectations, we investigated the following two hypotheses for the experiment:

H_{01}	*An inspection team is as effective or more effective using CBR than it is using PBR.*
H_{02}	*An inspection team using PBR finds defects at the same or higher cost per defect than a team using CBR for all phases of the inspection.*

In the experiment, we investigate directional null hypotheses. A directional null hypothesis can be regarded as a statement that there is a difference between the two groups opposite to that predicted. According to the logic of testing statistical hypotheses [25], we are interested in being able to reject these null hypotheses. The hypotheses are preceded by a zero to indicate that these are the null hypotheses being tested. We use standard t-test as well as permutation tests to validate the hypotheses [11], [25].

5.2 Study Design

The experimental design is depicted in Table 1. We use a notational system in which X stands for a treatment and O stands for an observation; subscripts 1 and 2 refer to the sequential order of implementing treatments.

Experiment				
Group 1	X_{PBR}	O_1	X_{CBR}	O_2
Group 2	X_{CBR}	O_1	X_{PBR}	O_2

Table 1: The design of the experiment.

For the experiment, subjects were randomly assigned to two groups[2]: Group 1 and Group 2. The first group (Group 1) performed a reading exercise using PBR first (X_{PBR}), and then measures were collected (O_1). Subsequently, they performed a reading exercise using CBR (X_{CBR}), and again measures were collected (O_2). The second group followed the counterbalanced sequence. This is a classic 2x2 repeated measures design described more fully in [25].

5.3 Perspective-based Reading of Object-Oriented Design Documents

We identified three perspectives for the inspection of a design document. Hence, a PBR team consisted of three inspectors each of which had read the design document from one of the three perspectives. The three perspectives were a designer perspective, a tester perspective, and an implementer perspective An inspector reading the design documents from the point of view of a designer is primarily interested in the correctness between the design models and the analysis documents. Hence, he or she has to extract relevant information from the design documents and compare it to the one described in the analysis documents. An inspector reading the design documents from the perspective of a tester identifies the different operations that the system is to perform in the design models and tries to set up test cases with which he or she can ensure the correct behaviour of each operation. Then, the inspector is supposed to mentally simulate each operation using the test cases as input values and to compare the resulting output to the description in the analysis documents. Finally, an inspector reading the design documents from the perspective of an implementor makes sure that all required information is provided in the design models to implement the system. This involves completeness checks as well as more difficult checks on the feasibility of the design.

5.4 Subjects

The subjects in this experiment were 18 practitioners with various backgrounds. Before the course they primarily worked as programmers in industry and had various levels of experience in object-oriented programming. In the experiment, they were randomly assigned to one of the two groups and, within each group, to one of the three perspectives according to the experimental design.

[2] For the random assignment, the subjects drew numbers from an envelope.

5.5 Results

5.5.1 Effectiveness of Inspection Teams

Figure 5 depicts a box-plot of the defect detection effectiveness of the inspection teams when applying the two different reading techniques. Box-plots provide information about the distribution of data points, such as the center (mean value), spread (standard deviation), and minimum/maximum values.

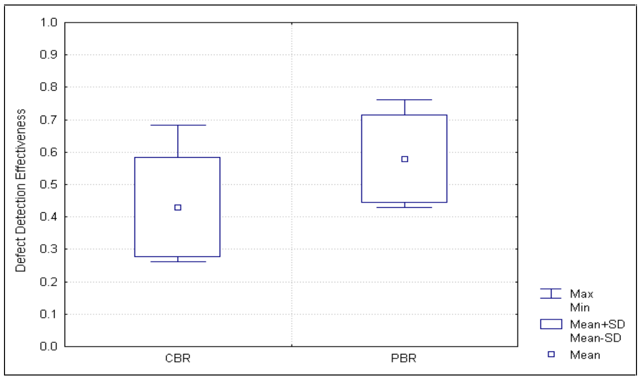


Figure 5: Defect Detection Effectiveness. (Mean values: CBR: 0.43; PBR: 0.58)

Overall inspection teams found more defects using the PBR technique than using the CBR technique. Using a matched-pair t-test the difference is statistically significant (t=3.17, p=0.025). Since the t-test makes several assumptions, we also performed a permutation test [11]. The permutation test revealed an exact p-value of 0.0313. We failed to detect a carry-over effect for the different sequences. This means that the sequence of using the various reading techniques did not influence the results of this study.

Based on these findings, we can reject hypothesis H_{01} and conclude that an inspection team is more effective, i.e., detect more defects, using the PBR technique than using CBR.

5.5.2 Cost-Effectiveness of Inspection Teams

Figure 6 depicts a box-plot of the cost per defect ratio of the overall inspection when using the two different reading techniques.

The team with the PBR technique had a lower cost per defect ratio using the CBR technique than using the PBR technique when considering the effort for the overall inspection. Using a matched-pair t-test the difference is statistically significant (t=-6.53, p=0.001). These findings were also confirmed with the permutation test (p=0.0156). We failed to detect carry-over effects. The study results therefore suggest that the PBR technique improve the cost-effectiveness of inspection teams.

Based on these findings, we can reject hypothesis H_{02}. This means that PBR teams are more cost-effective, i.e., find defects at a lower cost per defect ratio, than CBR teams.

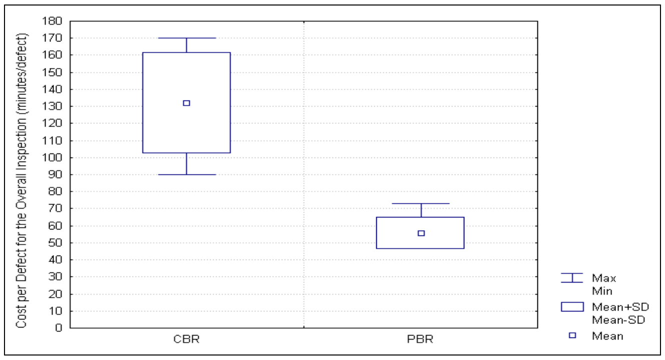


Figure 6: Cost per Defect for the Overall Inspection (CBR: 132 minutes per defect; PBR: 56 minutes per defect).

6 SUMMARY

Software inspection is regarded as one of the most effective methods for software quality improvement. To maintain its cost-effectiveness in the context of object-oriented development projects, two major inspection-related issues must be addressed. The first one is the question of how to organize an inspection with respect to the examined documentation. The second one is the question of how to provide systematic reading support for inspection participants.

In this paper, we have described a new approach for inspection organization. This approach distinguishes between logical entities and their documentation. We argue that logical entities from the software architecture should be used for inspection organization rather than their documentation. This helps ensure that inspectors are provided with appropriate amount of information for a sound inspection.

To support the defect detection activity of inspection participants in a systematic manner, we have explained the perspective-based reading technique. This technique requires inspectors to analyze the documentation of a logical entity from various stakeholder perspectives. In doing so, the technique provides procedural guidance on what to check and on how to perform the checking.

In a final step, we have presented the essence of a controlled experiment to compare the effectiveness and the cost per defect ratio of the perspective-based reading technique to checklist-based reading. The comparison was performed through a controlled experiment with practitioners participating in a course on object-oriented development. During the experiment the subjects used the CBR approach as well as the PBR approach for defect detection in design documents. The design documents were specified in the UML. Our experimental results indicate that the effectiveness of teams using PBR is greater than of those using CBR. Furthermore, we found that the cost per defect ratio using PBR is smaller than with CBR during the defect detection phase of inspections. Overall, we found that the cost per defect for the whole inspection is lower with PBR than with CBR. Therefore, PBR has effectiveness and cost advantages when compared to CBR.

Our findings provide evidence that the architecture-centric strategy and the perspective-based reading technique is a promising approach for achieving high quality in object-oriented development projects.

ACKNOWLEDGEMENT
We would like to thank Maud Schlich from Fraunhofer IESE, Germany, for her contribution to the success of the empirical study.

REFERENCES

1. V.R. Basili. Evolving and Packaging Reading Technologies. Journal of Systems and Software, 38(1), July 1997.
2. V. R. Basili, S. Green, O. Laitenberger, F. Lanubile, F. Shull, S. Sorumgard, and M.V. Zelkowitz. The Empirical Investigation of Perspective-based Reading. Journal of Empirical Software Engineering, 2(1):133-164, 1996.
3. J. Barnard and A. Price. Managing Code Inspection Information. IEEE Software, 11(2):59-69, March 1994.
4. G. Booch. Object Oriented Analysis and Design with Applications. Benjamin/Cummings, Redwood City, California, 2nd edition, 1994.
5. F. P. Brooks, Jr. No Silver Bullet: Essence and Accidents of Software Engineering. IEEE Computer, 20(4):10-19, April 1987.
6. D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremaes. Object-Oriented Development: The Fusion Method. Prentice Hall, 1993.
7. T.D. Cook and D.T. Campbell. Quasi-Experimentation: Design and Analysis Issues for Field Settings. Rand McNally College Publishing Company, Chicago, 1979.
8. D. A. Christenson, H. T. Steel, and A. J. Lamperez. Statistical Quality Control applied to Code Inspections. IEEE Journal Selected Areas in Communication, 8(2):196-200, February 1990.
9. M. E. Fagan. Design and Code Inspections to Reduce Errors in Program Development. IBM Systems Journal, 15(3):182-211, 1976.
10. T. Gilb and D. Graham. Software Inspection. Addison-Wesley Publishing Company, 1993.
11. P. Johnson and D. Tjahjono, Does Every Inspection Really Need a Meeting. Journal of Empirical Software Engineering, 3(1):9-35, 1998.
12. P. Good. Permutation Tests: A Practical Guide to Resampling Methods for Testing Hypotheses. Springer Verlag, 1994.
13. Les Hatton. Does OO Sync with How We Think? IEEE Software, 15(3):46-54, May 1998.
14. I. Jacobson, G. Booch, J. Rumbaugh, The Unified Software Development Process, Addison Wesley, 1998.
15. O. Laitenberger, Cost-effective Detection of Software Defects through Perspective-based Inspection. PhD-Thesis, University of Kaiserslautern, 2000.
16. O. Laitenberger and C. Atkinson. Generalizing Perspective-based Inspection to handle Object-Oriented Development Artifacts. In Proceedings of the 21nd International Conference of Software Engineering, 1999.
17. O. Laitenberger, C. Atkinson, M. Schlich, and K. El Emam. An Experimental Comparison of Reading Techniques for Defect Detection in UML Design Documents. Accepted for Publication in the Journal of Systems and Software, also published as ISERN-Technical Report 001.00/E, 2000.
18. O. Laitenberger and J.-M. DeBaud. An Encompassing Life-Cycle Centric Survey of Software Inspection. Journal of Systems and Software, jan. 2000.
19. O. Laitenberger, K. El Emam, and T. Harbich. An Internally Replicated Quasi-Experimental Comparison of Checklist and Perspective-based Reading of Code Documents. IEEE Transactions on Software Engineering, 2000.
20. R. C. Linger, H. D. Mills, and B. I. Witt. Structured Programming: Theory and Practice. Addison-Wesley Publishing Company, 1979.
21. A. A. Porter, H. P. Siy, C. A. Toman, and L. G. Votta. An Experiment to Assess the Cost-Benefits of Code Inspections in Large Scale Software Development. IEEE Transactions on Software Engineering, 23(6):329-346, June 1997.
22. Rational Software Coperation. Unified Modeling Language Documentation Set, Version 1.1, September 1997.
23. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. Object-Oriented Modeling and Design. Prentice Hall, 1991.
24. L. G. Votta. Does Every Inspection Need a Meeting? ACM Software Eng. Notes, 18(5):107-114, December 1993.
25. B. J. Winer, D. R. Brown, and K. M. Michels. Statistical Principles in Experimental Design, 3rd edition. McGraw Hill Series in Psychology, 1991.