Generalizing Perspective-based Inspection to handle Object-Oriented Development Artifacts

Oliver Laitenberger and Colin Atkinson
Fraunhofer Institute for Experimental Software Engineering
Sauerwiesen 6
67661 Kaiserslautern, Germany
+49 (0)6301707200
{laiten, atkinson}@iese.fhg.de

ABSTRACT
The value of software inspection for uncovering defects early in the development lifecycle has been well documented. Of the various types of inspection methods published to date, experiments have shown perspective-based inspection to be one of the most effective, because of its enhanced coverage of the defect space. However, inspections in general, and perspective-based inspections in particular, have so far been applied predominantly in the context of conventional structured development methods, and then almost always to textual artifacts, such as requirements documents or code modules. Object oriented-models, particularly of the graphical form, have so far not been adequately addressed by inspection methods. This paper tackles this problem by first discussing the difficulties involved in tailoring the perspective-based inspection approach to object-oriented development methods and, second, by presenting a generalization of the approach which overcomes these limitations. The new version of the approach is illustrated in the context of UML-based object-oriented development.

Keywords
Software Inspection, Reading Techniques, Perspective-based Inspection, Object-Orientation, Fusion, UML

1 INTRODUCTION

Since Fagan's seminal work in 1976 [8], software inspection has emerged as one of the most effective quality assurance techniques in software engineering. Fagan, and others, have shown that software inspection can lead to the detection and correction of anywhere between 50 and 90 percent of the defects in a software artifact [9], [11]. Moreover, since inspections can uncover defects shortly after they are introduced, rework costs (i.e., the costs associated with correcting defects) are considerably reduced. On average, the introduction of code inspection reduces rework costs by 39 percent and the introduction of design inspection reduces rework costs by 44 percent [5].

A full inspection usually consists of numerous activities including planning, defect detection, defect collection, and defect correction. However, it is the defect detection activity, or "reading" as it is commonly called, that is considered the key part of an inspection [2] and which therefore needs to be supported with adequate reading techniques. Moreover, empirical evidence suggests that reading techniques rather than inspection process variations have the biggest impact on inspection effectiveness [22].

Several kinds of reading techniques have been defined in the literature, the simplest of which is the ad-hoc reading approach [8]. As its name implies, this technique provides no explicit advice as to how to proceed, or what specifically to look for, during the reading activity, so inspectors must resort to their own intuition and experience to determine how to go about an inspection. A significant improvement over the ad-hoc approach is the so called checklist approach [11], in which an inspector is at least given a list of questions to answer. The checklist-based technique thus gives inspectors advice about *what* to look for in an inspection.

The next level of sophistication is offered by scenario-based reading techniques [2]. The basic idea of a scenario-based reading technique is the use of so called scenarios that describe *how* to go about finding the required information, as well as *what* that information should look like. In doing so, scenario-based reading techniques assign clear responsibilities to inspectors and require each of them to take an active role in an inspection. In these two ways, they are similar to active design reviews suggested by Parnas and Weiss [21] for the inspection of design artifacts. However, active design reviews provide little if any guidance to inspectors about how to perform the reading activity.

Of the several families of scenario-based reading techniques defined to date [3], [6], [23], experiments have shown the Perspective-Based Reading (PBR)[1] technique to be among the most effective. The basic idea behind this approach is to inspect an artifact from the perspectives of its individual "customers", with the assumption that collectively these will increase the coverage of the defect space. In doing so, the PBR technique synthesizes ideas that have already appeared in previous articles on software inspection, but have never been worked out in detail. For example, Fagan [8] reports that a piece of code should be inspected by its real tester, while Fowler [10] suggests that each inspection participant should

1.In the context of this paper, we use the term "perspective-based inspection" to refer to inspection processes that adopt the PBR technique for defect detection.

take a particular point of view when examining the work product. Graden et al. [12] state that inspectors must denote the perspective (customer, requirements, design, test, maintenance) from which they have evaluated the deliverable. Such viewpoint-oriented approaches follow the current thinking on quality: everybody, even someone internal to an analysis, design, or coding process, is considered to be a customer and also has customers [17]. Since customers are interested in different quality factors or see the same quality factor quite differently [20], a software artifact needs to be inspected from each customer's viewpoint.

Unfortunately, software inspections in general, and perspective-based inspections in particular, have been used primarily in connection with textual artifacts resulting from conventional structured development processes, such as requirements documents or code modules. Object-oriented artifacts, particularly of the graphical from, have so far not been adequately addressed by inspection methods. This represents a problem for two reasons. First, over the past decade object-oriented development methods have replaced conventional structured methods as the embodiment of in software development, and are now the approach of choice in most new software development projects. Inspection methods that are limited to conventional structured methods, therefore, will become less and less relevant as these methods are superseded. Second, despite its many beneficial features, low defect density is not one of the strong points of the object-oriented paradigm. On the contrary, some empirical studies have shown that object-oriented artifacts are more error-prone than functional ones [13], [14]. At least one reason for this situation is that most of the leading object-oriented development methods [4], [7], [24] lack comprehensive reading techniques for inspection. Object-oriented methods would, therefore, benefit enormously from the availability of such techniques. We believe systematic, viewpoint-based inspection approaches, such as the perspective-based inspection approach, offers one of the best ways of accommodating the complexity of object-oriented systems.

This papers aims to address the need for more mature inspection approaches in object-oriented development by generalizing the perspective-based inspection approach to handle a wider range of development artifacts. To this end, section 2 first identifies the limitations in the current formulation of the PBR approach that prevent it from being easily applicable in an object-oriented context. Section 3 then presents a more general version of PBR which addresses the identified shortcomings. Section 4 follows with an illustration of how this can be applied in the context of a UML-based object-oriented development project. Finally, Section 5 concludes.

2 LIMITATIONS OF THE CURRENT FORMULATION OF PERSPECTIVE-BASED READING

To date, there have been two published applications of the PBR technique. The original publication on PBR [3] describes NASA's use of the technique for the inspection of requirements specifications, while the second [19] details how a car parts manufacturing company applied the PBR technique for the inspection of source code. While both of these publications provided a useful working description of the PBR concept, they were more concerned with the experimental validation of the underlying viewpoint premise than on providing a generally applicable definition of the technique. They consequently adopted an interpretation of PBR which best suited the immediate needs of the application in hand, rather than focused on the subtleties involved in applying the principle to a wide range of different development artifacts.

From these publications, it is possible to distil the following working definition of the technique. In essence, the basic goal of PBR is:

"to read a software artifact from the perspectives of the artifact's various customers for the purpose of identifying defects."

As an abstract concept, this definition is simple and clear enough. The limitations in the current formulation of PBR arise not from this definition per se, but rather from the way in which researchers and practitioners interpret the key terms "defect", "customer", "artifact", and "to read". Although the existing interpretations of these terms work quite well in the current publications, they are too "loose" for PBR to be applied effectively in an object-oriented context. To better understand the difficulties, we elaborate upon the various interpretations in the following subsections before tackling these issues by presenting more concise definitions.

Interpretation of "Defect"

In the current formulation of PBR the term "defect" has not been precisely defined. In most existing inspection methods, such as [8], [21], a defect is usually interpreted solely as a fault in a software document that must be detected and repaired in order for the software artifact to be correct. Such an interpretation, however, limits software inspection, and perspective-based inspection in particular, to correctness as the only quality factor. However, this focus on correctness (or the lack thereof) is unnecessarily restrictive. For example, the maintainer of an artifact is not only interested in its correctness, but also in the extent to which it embodies good design practice and the ease with which it can be modified. From such a perspective, an artifact might be correct, but may be considered defective because it is too poorly constructed for maintenance purposes.

Definition of Perspectives

The use of the term "customer" in the current formulation of PBR causes two difficulties. First, a customer is usually a consumer, or recipient, of an entity, so the use of this term to define perspectives implies that an artifact is to be read only from the viewpoint of "recipients" of the artifact. However, the creators or developers of an artifact usually have just as much interest in its quality as the recipients or consumers.

Second, the word "customer" also has an implication of immediacy in a relationship which does not always apply in the context of software development. Often participants who could not reasonably be viewed as customers also have an interest in the quality of an artifact. For example, in Basili et al. [3] a "User" is presented as an example of a perspective from which to inspect a requirements document. However, can a "User" really be regarded as a customer (i.e., immediate consumer or recipient) of a requirements document? Not really. Users often do not even see the requirements document used to define a system.

In fact, in the examples given to date, all the perspectives are defined with respect to standard roles in a software project, not necessarily with respect to the immediate customers of the artifact. If anything, these perspective are defined with respect to the customers of the “system” as a whole rather than to the specific artifact under examination.

Artifacts versus Descriptions
The problem of how to define the perspectives from which to read an artifact is related to the deeper question of how the artifacts themselves are defined. Software is unique among engineering products in that strictly speaking it has no concrete material manifestation. Whereas a civil engineer, for example, can inspect the actual elements of a bridge that results from his endeavours, or a mechanical engineer can inspect the actual elements of an engine that he builds, a software engineer cannot actually look at a piece of a software system per se. He or she can only inspect representations, or *descriptions*[1] of the software product, such as design models or source code.

Such a description of an artifact can be viewed as a reification of the artifact which makes it tangible. Reification is a part of the software development process and encompasses the activities of describing artifacts, providing them in the form of physical documents and packaging them.

It is not necessary to distinguish between software artifacts and their descriptions in contexts where the following two conditions both hold:

1. There is a one-to-one correspondence between an artifact and its description.
2. An artifact has a fairly concrete manifestation in the final delivered software product.

However, in circumstances where these are not true the distinction is extremely important, and in fact is critical to the effective formulation of software inspection in general, and perspective-based inspection in particular. Both of these conditions were true in the two existing applications in which PBR has been used to date. In [3] there is a one-to-one correspondence between a requirements document and the system, and the latter obviously has a concrete manifestation in the delivered product - in fact, it “is” the delivered product. Similarly, in [19] there is a one-to-one correspondence between source code modules and functions, and the latter can readily be identified in delivered executables. As a consequence, the current version of PBR was formulated under the assumption that these conditions are valid. Figure 1 illustrates this concept. It indicates that no distinction is made between a software artifact and its description (because there is assumed to be only a single description for each artifact) when defining the perspectives from which to perform the inspection.

1. Various terms could have been used here, such as model, representation, or document, but we chose to use the word description since it best conveys the idea of something that can be graphical or textual.

Especially in the design phase of a project, however, there is often a many-to-many relationship between the artifacts that form part of a diagram, and their various descriptions. For example, in SA/SD a given function can appear in various data flow diagrams or structure charts (i.e., can have various descriptions), and conversely each such diagram can contain various functions (i.e., various artifacts). Hence, this is a many-to-many relationship. Similarly in most object-oriented methods [4], [7], [24], a given abstraction, such as a class or object, can be described in many class diagrams, and a class diagram can contain many classes.

Moreover, modern development philosophies, such as object-orientation, incorporate abstractions which have no concrete realization in the final delivered software. With most object-oriented languages, for example, even classes have no existence in the final system, and these are much more concrete than other abstractions, such as abstract classes, which play an important role in many object-oriented languages.

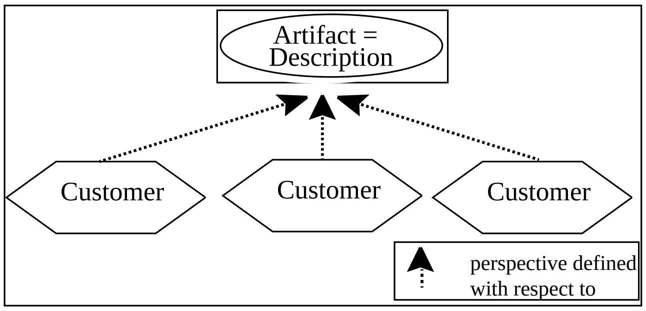


Figure 1: Assumption of PBR

Reading as Part of the Development Process
The reading process, and the scenarios which describe it, are of course one of the key elements of the PBR approach. A reading scenario tells an inspector how to go about reading an artifact from a particular perspective and what to look for. In the current formulation of PBR, the "reading" scenarios place a significant emphasis on the “creation” of models as well as on their analysis. For example, Basili et al. [3] state that "each reader produces some physical model which can be analysed to answer questions based on the perspective. For example, the team member reading from the perspective of the tester would design a set of tests for a potential test plan, the team member reading from the perspective of the developer would develop a high-level design, and the team member representing the user would create a user manual".

There are two difficulties with this formulation of PBR. The first is that it stretches the word “reading” beyond its natural meaning. “Reading” implies the systematic examination of an artifact’s description to extract and gain certain information for a particular purpose (e.g., for detecting defects etc.). However, a construction activity which corresponds to a major phase of a development process, such as the creation of a high-level design, would seem to go beyond simply “reading.”

The second, and more serious, problem arises when the artifact that an inspector is required to create would normally be created anyway, even in the absence of inspections. It seems reasonable that the inspector should be responsible for creating artifacts which are used solely for the purpose of inspections, but when an artifact would be created anyway (perhaps just at a different point in the process) it seems questionable to assign this responsibility to the inspector. However, all the examples of "physical" models given in the existing publications on PBR, such as those cited in the previous section, involve entities which are generally regarded as products of a software development project irrespective of whether PBR is being used. A high-level design, for example, is something that would normally be generated by a designer as a part of a normal development process. If PBR requires an inspector to generate this artifact as part of the reading process in an inspection, this must mean one of two things:

1. either the inspector is duplicating activities performed by others. For example, the inspector creates a design for the purpose of inspection which is later recreated by the designer for the purpose of implementation, or
2. the inspector is performing activities which are normally assigned to others. For example, the task of creating a high-level design is normal performed by a designer rather than an inspector.

3 A GENERALIZED VERSION OF PERSPECTIVE-BASED READING

In this section we discuss ways of overcoming the difficulties identified in the previous section, and present a more general version of PBR which we believe contains the optimal set of solutions to the identified issues. By addressing these problems we aim to place the PBR technique on a more sound footing, and make it scaleable to a larger range of development artifacts and paradigms. The essence of this new version of PBR is captured by the following working definition. The basic goal of PBR is to:

"examine the various descriptions of a software artifact from the perspectives of the artifact's various stakeholders for the purpose of identifying flaws."

In the following subsections we explain the rationale for this generalized definition of PBR by addressing, in turn, each of the problems identified in the previous subsections.

Interpretation of “Defect”

To enable PBR to realize its full potential it is necessary to remove the narrow focus on defects used in the current PBR formulation. The most obvious way of achieving this is to redefine the word “defect” to reflect the broadened interpretation. However, since this term has such a well established and accepted meaning [15], we prefer to introduce a new term which subsumes the established concept. A “flaw” is defined to be:

“*any property of an artifact or description which stops it from meeting its quality requirements.*”

This definition recognizes the importance of all the quality factors which may be important for a software artifact [20], while still accommodating the traditional focus on defects [16]. A defect is simply viewed as a special form of flaw in which the quality criterion is correctness. By defining the goal of PBR in terms of flaws rather than just defects, other quality shortcoming can be the focus of an inspection. For example, the failure to meet particular coupling or cohesion requirements in the design of an artifact might be a flaw of interest to a maintainer.

Definition of Perspectives

The problems arising from the use of the word “customer” to define the perspectives can be easily solved by replacing it with the word “stakeholder”. This word not only accommodates the creators of an artifact as valid perspectives from which it can be examined, but it also removes the immediacy implied by the word “customer”. Thus, a stakeholder can be any party interested in the quality of an artifact, whether it be a software engineer playing a traditional process role, or creators and customers of the artifact who have a much more immediate role in its production and consumption.

Artifacts versus Descriptions

As mentioned in the previous section, the assumption that an artifact and its description are identical (or at least in a one-to-one correspondence) is not generally valid. More often, there is a one-to-many or a many-to-many relationship between artifacts and descriptions. Consider, for example, artifacts that typically appear in an object-oriented system, such as classes or methods. During analysis and design these artifacts are each described through a collection of diagrams (e.g., use-case diagrams, class diagrams, statecharts diagrams etc.). Similarly, a given instance of these types of diagrams typically describes numerous classes and/or methods. Thus, there is usually a many-to-many relationships between the artifacts and the various descriptions of the artifacts.

In the absence of a one-to-one relationship, it is no longer possible to regard an artifact and its description as a single entity. Therefore, a major step in generalizing PBR is to explicitly recognize the distinction between software artifacts (e.g., systems, subsystems, classes, functions, objects, attributes) and their descriptions (e.g., class diagrams, use case diagrams, code modules etc.) and to define the reading technique accordingly.

Separating artifacts from their descriptions, however, raises the difficult question of how the inspection perspectives should be defined. In the previous formulation of PBR, this was not an issue because a software artifact and its description were regarded as a single entity, so the stakeholders were obviously defined with respect to this entity. However, when artifacts and descriptions are regarded as separated entities, it is no longer clear what the inspection perspectives should be defined with respect to. Both alternatives are actually feasible.

As illustrated in Figure 2, the so called “artifact-oriented” approach regards artifacts as the units of inspection, and the perspectives are defined with respect to the artifacts. What this means in practice is that an inspection is organized around, and focuses upon, a software abstraction, such as a class, an object or a method. In contrast, as illustrated in Figure 3, in the so called “description-oriented” approach, it is the descriptions that are regarded as the units of inspection from which to define the inspection perspectives. What this means in practice is that an inspection is organized around, and focuses upon, a particular software description, such as a class diagram, a use-case diagram or a source code element.

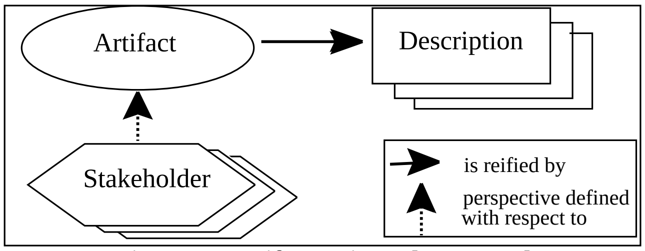


Figure 2: Artifact-oriented Approach

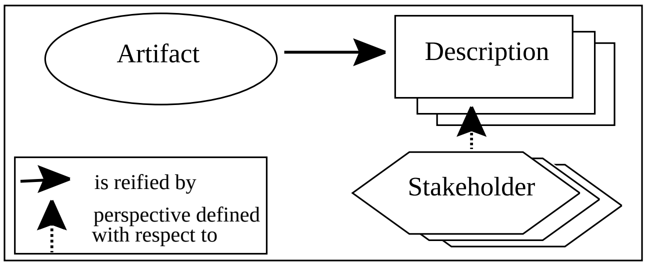


Figure 3: Description-oriented Approach

At first sight the description-oriented approach might appear to be the more natural, because it is the descriptions which are inspected. It seems strange to organize an inspection around artifacts which, by definition, can not actually be directly inspected. However, there are three reasons why we believe that the apparently counterintuitive artifact-oriented approach is actually the most effective.

First, the ultimate goal of any quality assurance activity, such as perspective-based inspection, is to ensure the quality of the final artifact, that is, the quality of the final software system and its components. In this respect, the descriptions of these artifacts are only of secondary importance, and only provide a means to an end. Second, although the relationship between artifacts and descriptions is in general many-to-many, an artifact usually has far fewer descriptions than a description has artifacts. For example, in an object-oriented development project, an artifact, such as a class or an operation, typically has between five and ten different descriptions. However, certain kinds of descriptions, such as class diagrams, may describe dozens of artifacts (e.g., classes). This asymmetry in the relationship cardinalities makes it a much more daunting task to organize inspections around descriptions rather than artifacts. Finally, defining perspectives with respect to descriptions can be unnatural for certain roles. For example, how can you inspect a class diagram from the perspective of a tester, when a class diagram cannot be tested? This implies that the perspectives (i.e., stakeholders) should not be defined with respect to descriptions.

This reasoning assumes that the relationship between artifacts and descriptions is well defined. In other words, given an artifact, the development method makes it clear which descriptions contain information about that artifact. Both approaches still work if the relationship is less well defined, but the successful completion of an inspection becomes much more difficult, since it is easy to miss information which can be critical for determining the quality of the artifact, or alternatively, it may take excessive effort to locate all the appropriate information (i.e., to find all the relevant descriptions).

Although this discussion might seem somewhat philosophical, it is fundamental not only for perspective-based inspection but for inspection techniques in general. However, we have found little discussion on this issue in the literature (e.g., [8], [18], [21]). Most existing inspection methods seem to make no distinction between an artifact and its descriptions.

Reading as Part of the Development Process

The perspective-based approach to inspection requires the inspector to gather and understand significant amounts of information about the artifact under consideration. However, it is not important *who* creates the descriptions from which this information is obtained. Of greater significance is *when* the required descriptions are created.

The problems in the current formulation of PBR with respect to the development process can therefore be largely addressed by returning the responsibility for creating the majority of descriptions back to development engineers, and focusing the reading activity on the extraction and examination of information rather than on the creation of descriptions. This is not a black/white solution because the extraction of information itself can be interpreted as creating new descriptions (i.e., collections or presentations of information). However, by applying the principle that inspectors, as part of the reading technique, should only create new descriptions if they would not normally be created, a reasonable and practicable separation of concerns is achieved. The creation of descriptions or artifacts that would normally be created even in the absence of inspections should be left to the usual development engineer (e.g., designer, tester etc.). In small projects this may actually turn out to be the same person as the inspector (i.e., one person playing two roles), but this does not diminish the value of conceptually separating concerns. If, to support inspections, a description or artifact needs to be created earlier than it normally would be (e.g., a test case) this can be reflected in the corresponding scenario by requiring the inspector to “arrange for” the description to be created, or some equivalent language. This ensures that the description is created *when* it is needed for the inspection, but still enables the actual work of creation to be performed by someone other than the inspector.

To support this approach, a more general form of scenario structure is required. As depicted in Figure 4, we suggest that this new form of scenario should consist of three major sections: introduction, instructions and questions. This structure is similar to the one described in the current formulation of PBR, the difference being in the content, particularly of the instructions and questions sections.

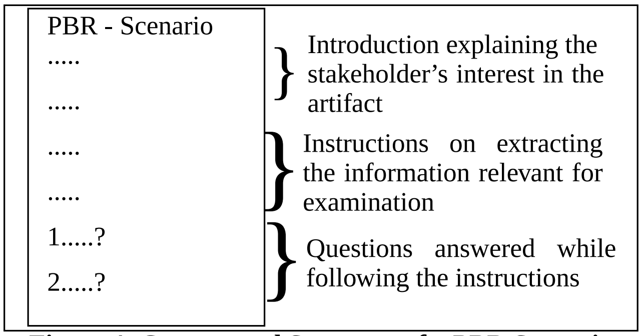

PBR - Scenario
.....
.....
.....
.....
1.....?
2.....?
Introduction explaining the stakeholder's interest in the artifact
Instructions on extracting the information relevant for examination
Questions answered while following the instructions

Figure 4: Content and Structure of a PBR Scenario

The introduction describes the stakeholder's interest in the artifact and explains the quality factors most relevant for this perspective.

The instructions describe what kind of descriptions an inspector is to use, how to read the descriptions, and how to extract the appropriate information from them. While identifying, reading, and extracting information, inspectors may already be able to detect some flaws. However, the primary goal of the instructions are three-fold: First, instructions help an inspector gain a focused understanding of the artifact. Understanding involves the assignment of meaning to a particular description and is a necessary prerequisite for detecting more subtle and more difficult flaws, which are often the expensive ones to remove if detected in later development phases. Second, the instructions require an inspector to actively work with the descriptions. Finally, the attention of an inspector is focused on the most relevant information, which avoids the swamping of inspectors with unnecessary details.

Once an inspector has achieved an understanding of the artifact, he or she can examine and judge whether the artifact as described fulfils the required quality factors. For making this judgement an inspector is supported by a set of questions which are answered while following the instructions. Hence, instructions and questions are framed together in a procedural manner. Defining the content of a scenario in this manner is in line with a more natural definition of "reading", which is the systematic examination of an artifact's descriptions to gather certain information for a particular purpose.

The success of the PBR technique relies on the ability of software engineers not only to follow existing PBR scenarios but to create new scenarios. This might be because of the need to accommodate new stakeholders or new artifact types. In the process of scenario creation, the first thing that needs to be determined is what type of artifact is to form the unit of inspection. This largely depends on the nature of the underlying development method. In function-oriented development methods, typical artifact types may be systems, subsystems, components, modules, or functions. In object-oriented development approaches typical artifact types are classes, objects, and operations (i.e., methods) as well as systems, subsystems and modules.

Once the inspected artifact has been determined, the next step is to define the required scenarios. To do this the scenario developer can follow the process explained below:

1. The first process step is to identify the types of descriptions that contain pertinent information about a particular artifact. This may be textual descriptions, such as textual design documents, or graphical models. It may be possible to identify them with the help of a product or process model since those define the descriptions that must be created for each artifact as part of the development method.
2. The second step is to specify the various stakeholders that have a vested interest in the artifact under inspection. As a starting point, the scenario developer may look at stakeholders that have a particular role in the software development process. These roles may be the producer of a preceding description of the artifact (if existing), the producer of a subsequent description of the artifact (if existing), the tester, and the maintainer. The user of the artifact as well as domain experts may be helpful as well. Each of these represents a different (technical) perspective on the inspected artifact. If a description is not of interest to any stakeholder, its value to the overall software development process is questionable.
3. For each of the perspectives, a scenario developer identifies what type of description and what kind of information in the descriptions is most important for a particular stakeholder (e.g., to perform his or her role in the software development process), how to identify, and how to extract this kind of information. For this, the scenario developer may interview the different stakeholders.
4. Once this has been performed, the scenario developer sets up the introduction part of the scenario by describing the interests of a stakeholder. Then, he or she develops instructions about how to identify and extract the required information. The granularity should have enough detail for an inspector to follow the given instructions step by step. Furthermore, it is important to somehow make inspectors document the extracted information (e.g., marking them with a coloured pen or writing parts of the information down). This captures what information an inspector has checked, for possible repetition at a later stage.
5. The fifth and final step in defining a scenario is to set up the questions an inspector is to answer based on the extracted information and the understanding of the artifact he or she has achieved. Characteristics of typical problems in a particular environment, illustrated by flaw distributions, are useful information for developing the questions since they are often typical representatives of problems in an environment. However, only those questions are to be included in a scenario that an inspector can answer with the understanding he or she can achieve based on the extracted information.

This process describes in a generic manner how to identify perspectives and how to create an initial set of scenarios. The crafted scenarios are generic in the sense that they can be reused for the inspections of the same kind of artifact within or even across projects. In practice, scenarios are rarely if ever defined completely from scratch, but are typically adapted from previous scenarios based on the experience gained from applying them.

Armed with this enhanced version of PBR, including a prescriptive process for setting up as well as executing the reading process, we are now in a position to illustrate how PBR can be applied to object-oriented development artifacts.

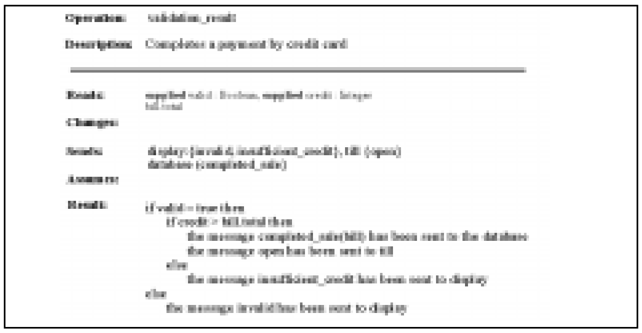
4 PERSPECTIVE-BASED READING IN OBJECT-ORIENTED DEVELOPMENT

As mentioned in the previous section, in development methods which allow an artifact to have multiple descriptions, and vice versa, PBR is much more effective when the mapping between the artifacts and descriptions is well defined. In other words, for every artifact that might be the subject of an inspection, it should be clear which descriptions (e.g., models, documents) contain information about that artifact. Of the leading object-oriented methods in widespread use, the one which comes closest to meeting this goal is the Fusion method [7]. Fusion is very precise about what specific models should be created as part of an object-oriented development project, and what information these models should contain. In contrast, most other leading object-oriented methods are vague about what models to produce and the extent of their information content. As a consequence, when inspecting an artifact it is not easy to ensure that all the information (i.e., models) describing properties of the artifact have been found and checked. Fusion also has the advantage that it uses a mix of textual and graphical models, and therefore reinforces the idea that the descriptions used and identified for perspective-based inspection can be of any kind. For these reasons, we use Fusion as the basis of the example. However, in view of the ubiquity and importance of the recently standardized Unified Modelling Language (UML) [25] we use the UML notation instead of Fusion's own notation by adopting the substitution strategy defined in [1]. This does not affect the ideas conveyed in the example. On the contrary, it should make them accessible to a wider audience.

Point of Sale System

The example is part of a *point of sale* system which is responsible for keeping track of the merchandise sold in a store, and handling the purchase of this merchandise. The main components of the system are the *central control point*, from which managers observe and enter merchandise information, the *database*, which stores the merchandise and sales information, and the *check-out control points*, each of which handles the sales from a particular check-out point. The system also interacts with external objects, such as a *credit card validation database*, to determine whether a credit card is valid. In the example, we focus on part of the functionality of a *check-out controller component*, specifically, the operation which deals with the credit card validation information obtained from a *credit card database*.

The Fusion method draws a strict boundary between the analysis and design phases of a development project. In the analysis phase, a system (or major subsystem, such as a check out control point) is described in terms of two main models: a class diagram (or object model as it is called in Fusion), and an interface model. In this example, the class diagram describes the different classes of relevance to the *check out control point* and how they are related to each other. The interface model is actually composed of two distinct submodels: the life-cycle model and the operation model, both of which are textual in nature. The life-cycle model identifies the operations which the subsystem exports (and thus has to implement) and describes acceptable execution sequences for them. The operation model provides a detailed declarative description of each of these operations in terms of preconditions and postconditions. Each individual operation description is termed an "operation schema" in Fusion. Figure 5 shows the operation schema for the operation *"validation_result"* which is responsible for dealing with the information provided by a credit card validation database in response to a prior request for a card check. As shown in Figure 5, a Fusion operation schema has seven so called "clauses". Apart from the name of the operation, which appears in the first clause, called "Operation", the two most important clauses are the "Assumes" clause and the "Result" clause. The first of these is a Boolean condition which states what must be true for the operation to be guaranteed to execute correctly, and the second is a Boolean condition which describes what becomes true as a result of the operation executing correctly. Both the preconditions (Assumes clause) and the post condition (Result clause) are written in terms of the entities modelled in the class diagram for the subsystem being analysed. The relevant part of the class diagram is shown in Figure 6.



```
Operation:    validation_result

Description:  Completes a payment by credit card

Reads:        supplied valid : Boolean, supplied credit : Integer
              bill.total

Changes:

Sends:        display {invalid, insufficient_credit}, till {open}
              database {completed_sale}

Assumes:

Result:       if valid = true then
                  if credit > bill.total then
                      the message completed_sale(bill) has been sent to the database
                      the message open has been sent to till
                  else
                      the message insufficient_credit has been sent to display
              else
                  the message invalid has been sent to display
```

Figure 5: Operation Schema

The purpose of the "Reads", "Changes", and "Sends" clauses is to define the scope of the operation by summarizing certain crucial pieces of information from the "Assumes" and "Results" clauses. The "Reads" clause identifies the information which the operation needs in order to do its job, but does not change. Parameters preceded with the keyword "supplied" are passed from the environment as input. The "Changes" clause, on the other hand, identifies those items which the operation may changes as it does its job. These are the entities in the subsystem which record the effects of the operation. The "Sends" clause lists the messages which the subsystem sends to other entities in its environment when it is performing the operation. The final clause, "Description", simply gives an informal description of the operation's effects and assumptions.

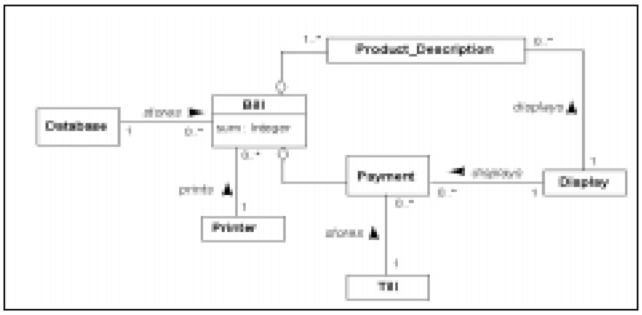


Figure 6: Class Diagram

The operation schema in Figure 5 indicates that the purpose of the *validation_result* operation is to complete a payment by a credit card. The Reads clause indicates that the operation requires three pieces of information to function, one called *valid* of type *Boolean*, one called *credit* of type *Integer*, and the third is the *total* attribute of the object *Bill*. The fact that these parameters are both preceded by the keyword "supplied" indicates that they are provided by the environment. The Changes clause indicates that this operation has no effect on the state of the system, while the sends clause indicates that the operation causes the system to send four messages, *invalid* and *insufficient_Credit* to the object *display*, *open* to the object *till*, and *completed_sale* to the object *database*. An empty Assumes clause, as in this case, actually corresponds to the value *True*. This means that the operation has no precondition and is therefore guaranteed to succeed under all circumstances. Finally, the Result clause indicates the conditions under which the operation sends the various messages depending on the values of the items in the reads clause.

The Fusion design phase requires a completely distinct set of descriptions (i.e., models) to be created. Several of these have been superseded with the advent of the UML, but the most important type of design diagram in Fusion, the object interaction diagram, has merely been renamed in the UML to collaboration diagram. Fusion requires a separate collaboration diagram to be created for each operation identified in the analysis phase. The purpose of this diagram is to describe how the effects of the operation are achieved through the interaction of a group of objects. Figure 7 shows a collaboration diagram for the *validation_result* operation specified in Figure 5.

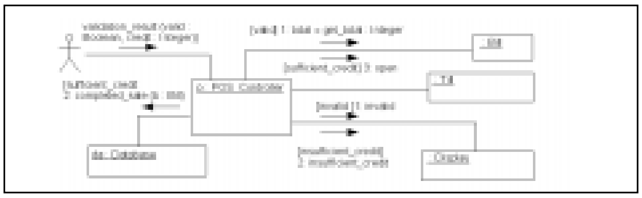
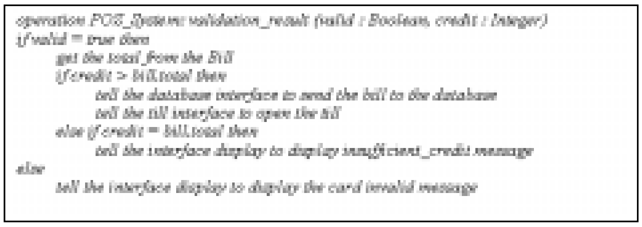

validation_result (valid : Boolean, credit : Integer)
[valid] 1: total = get_total : Integer
: Bill
[sufficient_credit] 3: open
: Till
[sufficient_credit]
2: completed_sale (b : Bill)
c: POS_Controller
[invalid] 1: invalid
db: Database
[insufficient_credit]
2: insufficient_credit
: Display

Figure 7: Collaboration Diagram

In a sense, a collaboration diagram of the form illustrated in Figure 7 gives a partial, graphical description of the algorithm used by the operation to fulfil its responsibilities. The notation provided by the UML allows conditions and branches to be described which determine the execution of the operation according to the values of the input parameters or the state of the system. However, because this algorithmic information is only partial, Fusion recommends that it be supplemented with a regular pseudocode description of the form illustrated in Figure 8.



```
operation POS_System: validation_result (valid : Boolean, credit : Integer)
if valid = true then
        get the total from the Bill
        if credit > bill.total then
                tell the database interface to send the bill to the database
                tell the till interface to open the till
        else if credit = bill.total then
                tell the interface display to display insufficient_credit message
else
        tell the interface display to display the card invalid message
```

Figure 8: Operation Pseudocode

Another important description in the Fusion method is the data dictionary. This is not shown here, but it is essentially a table containing textual descriptions of each artifact modelled in the system.

Inspecting the Operation "Validation_Result"

The first task in setting up an inspection is to define precisely which type of software artifact will be the subject of the inspection. In this example, it is the *"validation_result"* operation of the check out point subsystem. In other words, the "artifact" in our example is the *"validation_result"* operation. Of course, this is only one of the many operations of the point of sale system, each of which should be inspected individually.

Once the type of the artifact has been determined, the process described in the previous section can be used to define the reading scenarios. According to this process, the first step is to identify the relevant description types of the artifact. In our case, these are the operation's schema, the class diagram, the operation's collaboration diagram, the operation's pseudocode description, and the data dictionary.

The next step is to define the stakeholders that have an interest in the quality of the artifact, and thus represent a perspective from which to inspect it. Any person, or role, which is in some way affected by the artifact's quality, however remote, can serve as the basis of an inspection perspective. In this example we will consider the typical stakeholders used in perspective-based inspection, which are defined in terms of the roles in the development process. Hence, the stakeholders we consider are requirements engineer, designer, and tester. This list is not exhaustive, but serves to illustrate how the PBR approach would function in an object-oriented project.

The next step is to identify which of the description types are of relevance to the different perspectives. This information is best captures in a table, as illustrated in Table 1.

	Requirements Engineer	Designer	Tester	Maintainer
Operation Schema	✔	✔	✔	
Class Diagram	✔			
Collaboration Diagram		✔		✔
Operation Pseudocode		✔	✔	✔
Data Dictionary	✔			

Table 1: Assignment of Perspectives to Descriptions

The final step is to define the actual scenarios, one for each perspective. Obviously due to space limitations it is not possible to show complete scenarios in their full generality, since these typically run into several pages. Instead we aim to illustrate the essence of what a scenario would look like.

Reading from the Perspective of a Requirements Engineer
The concern of the requirements engineer is to ensure that the specification of the operation at the end of the analysis phase is complete and error free. In particular this means that there must be no inconsistencies between the various analysis models that carry information relevant to the operation. The requirements engineer's scenario, therefore, describes the activities that need to be performed, and the precise constraints that must be checked, in order to be confident that no inconsistencies exist. Figure 9 depicts the requirements engineer scenario.

Assume you are inspecting an operation from the perspective of a requirements engineer. The main concern of a requirements engineer is to ensure the consistency of the various descriptions of the operation in the analysis models. High quality therefore corresponds to few inconsistencies. The development products which are of relevance are the operation's schema, the class diagram and the data dictionary. Follow the instructions below and answer the questions carefully.
Locate the analysis class diagram, the data dictionary and the schema for the operation under inspection. Identify the clauses and highlight them with a pen. Carefully examine the clauses in the operation schema to ensure that they refer only to concepts that appear in the class diagram or the data dictionary. Then examine the clauses to ensure that the functionality of the operation is fully defined and that there are no inconsistencies.
While following these instructions answer the following questions:

1. Is every class, attribute or association named in the operation schema defined in the class diagram?
2. Is every type named in the operation schema defined in the data dictionary?
3. Are the initial conditions for starting up a function clear and correct?
4. Are the effects of a function specified under all possible circumstances?

Figure 9: Scenario for Req. Engineer's Perspective

The careful application of this scenario would reveal the following inconsistency between the operation schema and the class diagram. The operation schema uses an attribute of *Bill* called *total* to determine when to send particular messages, but in the class diagram, no such attribute exists. Instead there is an attribute called *sum*. This inconsistency would be revealed by the first question in the scenario, and obviously would need to be corrected in one or other of the descriptions.

Reading from the Perspective of a Designer
The task of the designer is to define how the required behaviour specified in the operation schema is to be achieved in terms of interactions between objects in the system. When inspecting from the perspective of the designer, therefore, the goal is to ensure that the various descriptions of this interaction are consistent with one another. Figure 10 depicts the designer's scenario.

The careful application of the designer scenario would reveal the following inconsistency between the operation schema and the collaboration diagram for *validation_result*. The schema indicates that under certain circumstances the message *open* should be sent to the object *till*. However, no such message appears in the collaboration diagram. In fact, *till* has no incoming message at all. Instead a message called *open* is sent to the object *Bill* at the exact point in the algorithm when it should be sent to *till*. This is obviously a mistake in the collaboration diagram that needs to be corrected.

Assume you are inspecting a system operation from the perspective of a a designer. The main task of a designer is to describe how the operation meet its responsibilities in terms of interactions between objects. High quality is determined by correctness of the design with respect to the specification, and the satisfaction of performance goals.
Locate the collaboration diagram, the pseudocode description and the schema for the operation. For each possible outcome of the postcondition, ensure that the appropriate messages are dispatched between the appropriate objects to achieve the desired goal. Mark the outcome as well as the message with a coloured pen. Check that the outcomes and the messages described in the pseudocode and the collaboration diagram are consistent.
While following these instructions answer the following questions:

1. For every message that is defined in the operation schema, is there a corresponding message sent in the collaboration diagram
2. For every attribute that is changed in the operation schema, is an appropriate message sent to the corresponding object in the collaboration diagram?
3. Are there any discrepancies between the algorithms defined in the collaboration diagram and the pseudocode description?

Figure 10: Scenario for Designer's Perspective

Reading from the Perspective of a Tester
The concern of the tester is to ensure that the operation is defined in a way that is testable. The basic idea, therefore, is for the inspector to work through various test cases, and to ensure that the descriptions of the operation are correct with respect to these test cases. Traditional testing concepts are thus highly applicable here, such as black box/white box testing, equivalence class partitioning, etc. Figure 11 depicts the tester's scenario.

The careful application of the tester scenario would reveal the following defect in the pseudo code description of the operation. Analysis of the branch conditions in the inner "if" statement indicates that certain allowed values of the input value *credit* are not catered for in the branching structure, namely, the situation where *credit* is less than the *total* attribute of *bill*. While this is unfortunate for the customer concerned, it is nevertheless a valid situation which must be catered for. The algorithm must, therefore, be corrected.

Assume you are inspecting an operation from the perspective of a tester. The main goal of a tester is to ensure the soundness of an operation. High quality thus corresponds to correctness and robustness. You will need to analyse test cases for the operation, so if they are not available arrange for them to be created. 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 operation schema and the pseudo code description for the operation under inspection. In the operation schema, identify the parameters which are preceded by the "supplied" keyword. Identify the equivalence classes for these parameters, and also the attributes named in the "reads" clause and document them. Using these equivalence classes, identify the minimal set of test cases needed to fully exercise the functional interface of the operation. In the pseudocode description of the operation identify every conditional branch or loop which represents an execution branch. Identify an additional set of test cases which ensure that each branch would be executed.
While following the instructions answer the following questions

1. Do the branches in the pseudocode description match the condition outcomes in the operation schema?
2. Are all possible sets of input values properly addressed by the operation schema and the pseudocode description?
3. Are operations preconditions indicated?

Figure 11: Scenario for Tester's Perspective

Reading from the Perspective of a Maintainer
The task of a maintainer is to ensure the maintainability of the system. In practice this means that the complexity of the operation's design needs to kept to a minimum, and that it should adhere to well established design principles formulated to maximize maintainability. Figure 12 depicts the

maintainer's scenario.

Assume you are inspecting an operation from the perspective of a maintainer. The main goal of a maintainer is to ensure that the collaboration diagram is written in a way that can be easily changed and maintained. High quality, therefore, means the conformance to specified design guidelines (low coupling, high cohesion) and the minimization of complexity.
Locate the collaboration diagram and the pseudocode description for the operation. Examine the diagram and the descriptions to identify points of converge from good design practice.
While following the instructions answer the following questions:

1. Are there any ways in which the number of objects, or the number of messages could be reduced?
2. Are there any cycles of messages in the collaboration diagram?
3. Is there any way in which the control structure of the operation could be simplified?
4. Do the messages entering an object indicate the possibility of low cohesion (are the messages totally unrelated)?
5. Is there a particularly high number of messages between a pair of objects?

Figure 12: Scenario for Maintainer's Perspective

5 CONCLUSION

Inspections have become an indispensable tool in the quest for higher quality software systems. However, even the more advanced inspection techniques, such as perspective-based inspection, have failed to fully make the transition from traditional structured development methods to more modern software approaches, such as object-oriented development. These development methods consequently have a major weakness in the area of systematic inspection.

In this paper, we have tackled this problem by first clearly identifying and elaborating the reasons why existing inspection methods, such as perspective-based inspection, are currently not formulated in a way that enables them to be scaled-up to meet the inspections needs of a wider range of artifacts and methods, and then by defining a generalized version of the PBR approach which addresses these problems. Most of the ideas embodied in this new approach are not limited to PBR, but should be of value to a wide range of inspection techniques. The main motivation for this work, however, was the support of more mature inspection techniques in object-oriented development. To demonstrate that the generalized PBR approach meets this goal, an example was presented which illustrates how the approach would be used in the context of a UML-based object-oriented development project.

Researchers and practitioners may benefit from this work in two ways. First, by providing a practical and concrete definition of PBR, researchers have a solid base upon which to perform quantitative investigations of the benefits of perspective-based inspection in future. Second, practitioners are provided with concrete advice on how to instantiate the generalized version of PBR for the inspection of object-oriented artifacts, especially in the early phases of development. Considering the lack of quality assurance techniques for object-oriented analysis and design descriptions, we believe this paper makes a step in filling this gap. Furthermore, practitioners can leverage their existing inspection approaches with a systematic reading technique even for artifacts developed according to conventional structured development methods.

REFERENCES

[1] C. Atkinson. Adapting the Fusion Process to support the UML. *Object Magazine*, 1997.

[2] V.R. Basili. Evolving and Packaging Reading Technologies. *Journal of Systems and Software*, 38(1), July 1997.

[3] 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.

[4] G. Booch. *Object Oriented Analysis and Design with Applications*. Benjamin/Cummings, Redwood City, California, 2nd edition, 1994.

[5] L. Briand, K. El-Emam, T. Fußbroich, and O. Laitenberger. Using Simulation to Build Inspection Efficiency Benchmarks for Development Projects. In *Proceedings of the Twentieth International Conference on Software Engineering*, pages 340–349. IEEE Computer Society Press, 1998.

[6] B. Cheng and R. Jeffrey. Comparing Inspection Strategies for Software Requirements Specifications. In *Proceedings of the 1996 Australian Software Engineering Conference*, pages 203–211, 1996.

[7] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremaes. *Object-Oriented Development: The Fusion Method*. Prentice Hall, 1993.

[8] M. E. Fagan. Design and Code Inspections to Reduce Errors in Program Development. *IBM Systems Journal*, 15(3):182–211, 1976.

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

[10] P. Fowler. In-process Inspections of Workproducts at AT&T. *AT&T Technical Journal*, 65(2):102–112, mar 1986.

[11] T. Gilb and D. Graham. *Software Inspection*. Addison-Wesley Publishing Company, 1993.

[12] M. E. Graden, P. S. Horsley, and T. C. Pingel. The Effects of Software Inspections on a major Telecommunications-project. *AT&T Technical Journal*, 65(3):32–40, May/June 1986.

[13] L. Hatton. Does OO Sync with How We Think? *IEEE Software*, 15(3):46–54, May 1998.

[14] W. H. Humphrey. *A Discipline for Software Engineering*. Addison-Wesley, 1995.

[15] Institute of Electrical and Electronics Engineers. *Standard Glossary of Software Engineering Terminology*, 1983.

[16] Institute of Electrical and Electronics Engineers. *IEEE Standards Collection - Software Engineering - 1994 Edition*, 1994.

[17] S.H. Kan, V.R. Basili, and L.N. Shapiro. Software quality: An overview from the perspective of total quality management. *IBM Systems Journal*, 33(1):4–19, 1997.

[18] J. C. Knight and E. A. Myers. An Improved Inspection Technique. *Communications of the ACM*, 36(11):51–61, 1993.

[19] O. Laitenberger and J.-M. DeBaud. Perspective-based Reading of Code Documents at Robert Bosch GmbH. *Information and Software Technology*, 39:781–791, March 1997.

[20] J. A. McCall. Quality Factors. In John J. Marciniak, editor, *Encyclopedia of Software Engineering*, volume 2, pages 958–969. John Wiley and Sons, 1994.

[21] D. Parnas. Active Design Reviews: Principles and Practice. *Journal of Systems and Software*, 7:259–265, 1987.

[22] A. A. Porter, H. Siy, A. Mockus, and L. G. Votta. Understanding the Sources of Variation in Software Inspections. *ACM Transactions on Software Engineering and Methodology*, 7(1):41–79, January 1998.

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

[24] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.

[25] Rational Software Cooperation. Unified Modeling Language Documentation Set, Version 1.1, September 1997.