Software Review Guidelines

Walcélio Melo
Oracle / UCB
196 Van Burren St,
Suite 200
Herndon, VA, 20170
wmelo@computer.org

Forrest Shull
Fraunhofer Center - Maryland
University of Maryland
4321 Hartwick Road, Suite 500
College Park MD 20742
301-403-2705
fshull@fc-md.umd.edu

Guilherme H. Travassos
Computer Science Department
COPPE/UFRJ
Bloco H – Suite 322-6 - CT
Rio de Janeiro RJ 21945-180,
Brazil
55 21 2562-8712
ght@cos.ufrj.br

August 2001

Table of Contents

1 Introduction

Rework is still a great source of software expenditures in many software organizations. According to Wheeler et al (1996), Boehm and Duncker commented that the cost of rework at TWR and HP, respectively, was approximately 30%. Dion and McGarry said that in their companies rework cost in average was responsible for approximately 40% of the total software development cost.

Several approaches can be used to reduce this huge amount of effort. Many authors have considered reviews an inexpensive and effective approach for reducing rework. For instance, NASA obtained 29% of total improvement in its processes and 10% in the reliability on its products with the adoption of software review activities in its projects.

Software review activities can be applied at many points during the software development process and can be used to discover defects in any type of deliverables or internal work products. Therefore, we can say that software reviews have as an objective the "purification" of software artifacts. [Pressman, 1997]

Software deliverables such as source code, project design or requirements specifications should be reviewed, since the more a deliverable is reviewed the better, freer of defects and more complete it will be. The reasons for reviewing software deliverables are analogous to those for reviewing written text. Many people have had the experience of writing a piece of text that is felt to be of high quality, only to read it over later and be surprised to discover grammatical, orthographic, and concordance mistakes as well as ideas that could simply be expressed in a better way. Often, an independent, objective reader finds mistakes that were not found by the authors. This means that a good reader will probably be a good reviser.

Moreover, it is known that engineers spend up to 1/3 of their time compiling and testing, relying on these activities to detect defects in their thought process. For example, Humphrey (1989) noticed that in a class of 12 students, they spent approximately 30% of the time compiling and testing. Then, as soon as he asked the review technique to be systematically applied, this number fell to 10%, and, by the end of the course, it had reached 0.3% (approximately). Spending less time on compiling and testing allowed the students to pay more attention to development, with increased efficiency.

In order to have a high level of software quality, reviews are useful in each phase of development (if possible), especially of code, with the following aims:

- To have a more comprehensible project, that facilitates comprehension by other developers, by describing in a condensed way what is described in the code.
- Saving implementation time, by removing problems with faulty logic and missing functionality before implementation.
- Improving the efficiency of reviews, since fewer artifacts need to be reviewed together and defects are removed incrementally, rather than at one time.
- Assessing and incorporating improvements to the project before compilation.

In summary, when the projects are revised before implementation, it is possible to remove defects and find more interesting and efficient ways of doing the process. It can also make the project clearer.

This document aims at presenting a set of guidelines and processes for software review. Our goals are the following:

- To improve software organization margins by significantly reducing rework.
- To increase the quality of deliverables by discovering defects before they are delivered to the customers;
- To augment productivity of software developers by providing them with standard and guidelines for the review processes that they will be responsible for performing.

To achieve these goals, this document is organized as follow. We first present the definition of software quality. Since, software review aims improving software quality, it is important that the concepts behind the idea of software quality be very established. Then, we discuss about software quality engineering. As there are many different types of review approaches, we also comment the most important kinds of software review approaches presenting their pros and cons. After that, we present in more details the software review processes for two kinds of reviews: inspection and structured walkthrough. Then, we present a set of guidelines for helping our software development conducting either inspection or walkthrough processes.

2 Definitions

2.1 Software Quality

Before we discuss software quality in a more formal way, it is interesting to analyze a quite known "definition", given by Philip Crosby.

> *"The problem... of quality is not what people do not know about it. The problem is what they know...In this sense, quality has much in common with sex. Everyone is interested in it (under certain conditions, of course). Everyone thinks they understand it (even if they do not want to explain it). Everyone thinks the execution is only a question of following natural inclinations (after all, we have already had a good performance!). And, of course, the great majority of people believed the problems in these areas were caused by other people (They would only had to have time to do things right)."*

A funny definition aside, quality is a term still poorly studied and rarely scientifically applied. Before we make the term *software quality* formal, we will define what each one of these terms mean.

According to Pressman (1997), a software system is a set of instructions that, when executed, produce the desired function and performance; data structures which allow programs to manipulate information adequately; documents which describe the operation and programs' use.

In the dictionary, **quality** is defined in the following way: "property, attribute or condition of things or people that distinguish them from the others and determines their nature".

Now that we have defined each one of the terms in separate, we can define software quality. To do so, many definitions have been proposed.

The simplest definition is that "quality begins and finishes in the client", that is, it is up to the client to define the criteria for a quality product according to his own, necessarily subjective, perception. Unfortunately, this definition, though simple, does not help a lot. As software engineers, we are responsible for the construction of systems, without client involvement at every step, based on well-defined parameters so that the products will satisfy the clients at the end.

The opinion given by Philip B. Crosby gives a better definition: "quality is conformity with the specifications, which is measured by the cost of non-conformity." In this case we find the word "specifications", that is, quality consists in building correctly what has been specified. Moreover, we find the word "measure", that is, we can measure the quality of a product objectively, based on its non-adherence to its specifications. Then it follows that "the first step towards quality is to know the client's specifications".

According to Pressman (1997), **software quality** is the conformity to functional and non-functional requirements (for example, performance) that have been explicitly declared, to development patterns clearly documented, and to implicit characteristics expected of all professionally developed software. In sum, quality consists of a set of requirements and a product or service that is in conformity with those requirements and, for this reason, that completely fulfill all the client's needs.

The most important part in the concept of quality is that it must be understood and identified inside the organizational environment, associating it to the practices, procedures, and its culture. Only after understanding what is quality, inserting its concept in the corporation's environment, an organization will be able to achieve excellence and, therefore, differentiate itself from its market competitors.

2.2 Software Quality Assurance

Software quality assurance, according to Pressman (1997), is a systematic and planned pattern of actions that are demanded to guarantee software quality.

Activities that can be used to check software quality are:

- Software review: Systematic reading activities performed by the technical staff with the sole objective of finding analysis and design defects in software artifacts produced in the initial phases of software development.

- Testing: A multi-step strategy combined with methods for producing representative test cases help guarantee effective defect detection.

- Patterns and formal procedures: These are patterns and procedures imposed by the client, or rules that direct how the project must be developed.

- Change control: Contributes to quality by formalizing the order of changes, evaluating the nature of the change and controlling its impact.

- Software metrics: Used to trace software quality and to evaluate the impact of various methodologies and procedures.

- Registering and keeping of records: Offer information collection and dissemination procedures.

Software review activities, beyond having both their efficiency and efficacy proved, are especially interesting, since they can be done in software development process initial phases. The only necessary materials are the analysis and project activities by-products documents. As shown in

Figure 1, the sooner a defect is identified, the lower the cost to repair it.

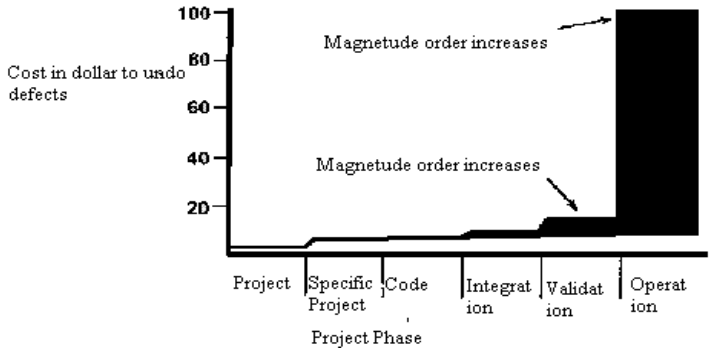

Figure 1: Defects repair cost

Some data help us illustrate the importance of revision for software quality in the development process. For example, Table 1 shows rework cost in many organizations.

	Rework	Source
TWR	30%	Boehm 1987
NASA	40%	McGarry 1987
HP	33%	Duncker 1992
Raytheon	41%	Dion 1993

Table 1: Rework cost (from Wheeler et al, 1996)

This cost could be reduced with the practice of revisions when applied at early phases of the software development process. This is shown in the NASA example, demonstrated on Table 2, NASA obtained 29% of total improvement in its processes and 10% in the reliability on its products with the adoption of software review activities in its projects. In the next section, we will study the different revision techniques in detail.

Factor	% Improvement when used	% Variation in reliability
Quality Assurance	29	10
Revision	29	10
Documentation	27	8
Chief Programmer	8	1
Cascade Model	6	1
Structured Code	3	1
Tools Use	3	1

Table 2: Quality improvement factors at NASA

3 Types of Review

Reviews can be categorized into different types, depending on the technique applied. The most common categories are formal review, internal review, walkthrough, and inspection.

3.1 Formal reviews[1]

Formal review is accomplished directly by the customer, with the objective of providing client feedback to the developer. Its process is basically the following: when a phase in the development cycle is finished, the developer calls for a meeting with his staff and presents what has been done to the client. The client reviews the material and notes what is not clear, what does not match the system requirements, and what is missing. Then, the developers explain why the project was developed as it was and this discussion continues until a common understanding is reached. Afterwards, developers make the changes agreed between them and the client.

The advantage of this type of review is that it can be performed at any development point, so developers do not have to wait until the end of a phase. They only have to feel the need to discuss some point with the client.

Its disadvantage is that, since it follows an *ad hoc* process, many defects in the logical conception of the system may pass unnoticed during the meeting.

3.2 Internal Review

The internal review cycle is the oldest method of project review. In it, the developer distributes copies of the project to many people, chosen by the developer himself. These people will then analyze the project and give feedback to the developer.

A main advantage of using this method is that the developer receives a more directed feedback, and this facilitates the job of correcting mistakes. Also due to the fact that it involves a lower number of people, it also costs less.

The disadvantage of this process is that the inspection results will represent the personal interests of the reviewers (e.g. a programmer chosen as reviewer may primarily emphasize implementation considerations and de-emphasize usability issues), and this may generate contradictory results.

3.3 Walkthrough

The walkthrough is a review process that focuses on the consensus and, through it, removes problems. In this process there is a moderator responsible for the direction of ideas and a reporter responsible for the exposition of the problems found. The discussion is about the report produced by the reporter and the moderator is responsible for keeping the revision meeting's focus, in this case, walkthrough. When there is an impasse, the moderator decides it.

One of the main advantages of the walkthrough process is that everyone participates in all phases; so when one needs to leave, another can, with fewer problems, take his place. Another advantage is

[1]The word formal is used here following the definition used by Wheeler, Brykczynski and Meeson (1996).

that the members integrating the group can learn by observing and working together with people who are in the group for a longer period of time.

One of the main disadvantages of this technique is that it is not very rigorous, since it is an informal process, and this leaves space for mistakes. This is because the fact of finding mistakes is a consequence of the process and not an objective to which it is directed. Another disadvantage comes from the fact that it does not always keep record of the work.

3.4 Inspection

Software inspection is a rigorous defect detection process. It is similar to the walkthrough, but, since it is more rigorous, it ensures that a higher percentage of defects are detected.

The classical inspection process begins with developers presenting the artifact to be inspected and addressing any questions of the reviewers. After the developers' presentation, the developers are removed from the process and only come back to the meeting to solve any problems that come up. At this time, the reviewers study the project and detect mistakes. Afterwards, the moderator chooses a reporter who will point out the defects found. Discussion is held about these problems and, when there is an impasse, the moderator is responsible for solving them. After the end of the meeting, the project is given to the developer responsible for the correction of mistakes found.

During the meeting, the staff decides if a re-inspection of the corrected artifact will be done, based on the number of mistakes found and their seriousness.

The advantages of this inspection process are: there is an effective mistake detection mechanism, and due to this, qualitative and quantitative project feedback is given earlier. Also, a record of the inspections is kept, allowing the evaluation of the inspection job. Record keeping also allows an evaluation of the defects made by the developers' group, helping to give guidance on the training needs of the enterprise.

The disadvantage is that the client stays completely out of the inspection work. Therefore, the project assumes that the understanding of user needs was correct, and this is not always true.

3.5 Review versus Testing

Maybe the greatest obstacle to a review is convincing the people involved in the process (system developers, analysts who will work as inspectors, etc.) that it is better to study and review a project than to rely on finding defects during compilation and test.

When reviews are first begun in an organization, most of the time many of the defects discovered will be noticed to be of the same type. A reviewer can use this information to save time, by understanding where defects are more likely to be found. This type of optimization based on patterns in the data is impossible in compilers, since they do not record information between compilations.

Code review is more effective than test because in review the faults in the code are found directly, while testing uncovers only the symptoms of problems, requiring debugging to find the direct cause. The seriousness of the wrong behavior by the system does not have a relation to the type of mistake, since even simple mistakes can cause complex behaviors.

With review, it is possible to check the program's methods, control flow, and syntax, using a mental picture what will be executed. If something is not well understood, it means there is a defect, which can be corrected before implementing, debugging, and testing the program.

In debugging, the work is different. The logic of the program is not analyzed, and therefore, many problems are not detected. Humphrey (1989) describes a case study where a group of engineers lost three months trying to find a defect, until they called in a group to perform a review. This group, in two hours, found the defect and 71 others.

However, this does not mean that software construction can be done without testing. Deutsch and Willis (1988) recommend combining testing with review for more effective results. In their view, the quality-engineering ideal is to remove as many defects as possible before testing. Nevertheless, this does not mean that all the defects can be removed. Because of this fact, the union of both review and testing can achieve much more efficient results than the use of either one of these techniques in isolation.

Testing is the controlled execution of software in order to find defects. Therefore, ideally, to test a piece of software, all input values to the code should be predicted, since the testing results depend directly on them. However, this is impossible and economically impracticable. So, testing is done with the most critical inputs and according to economic restrictions. Thus the greatest testing problem is in resolving the tension between the need to test the greatest number of inputs and the need to do so in the shortest amount of time and with the lowest costs.

By proceeding testing with review, the majority of defects can be eliminated early, and developers begin to have a broader view of the system's complexity. This will leave them better prepared for the moment they face this complexity and the defects that may possibly appear during test.

4 Software Inspections

Software inspections are a type of software quality assurance activity that can be performed throughout the software lifecycle. Because they rely on human understanding to detect defects, they have the advantage that they can be done as soon as a software work artifact is written and can be used on a variety of different artifacts and notations. Because a team typically accomplishes them, they are a useful way of passing technical expertise as to good and bad aspects of software artifacts among the participants. And, because they get developers familiar with the idea of reading each other's artifacts, they can lead to more readable artifacts being produced over time. On the other hand, because they rely on human effort, they are affected by non-technical issues: reviewers can have different levels of relevant expertise, can get bored if asked to review large artifacts, can have their own feelings about what is or is not important, or can be affected by political or personal issues. For this reason, there has been an emphasis on defining processes that people can use for performing effective inspections.

Fagan (1976) initially developed the software inspection process. Software inspection aims at guaranteeing the software is free of defects, complete, consistent, with no ambiguities and correct enough to support the maintenance that must be done.

Software inspection is a process capable of providing better quality software and reducing development time and, therefore, the costs required for producing software. Some studies demonstrate that inspection can detect up to 80% of all software problems in the development phase

[Fagan, 1976]. Inspection increases productivity by reducing the cost of rework, which is a problem facing every developer. Consequently, time is saved in project development.

4.1 Costs and Benefits

Many enterprises, including IBM, HP, AT&T, American Express, and NASA, have published articles on the benefits of inspection (see Table 3). In general, it is possible to see that decreases in the quantity of defects, effective as soon as inspections are introduced. In typical software development, many of the defects introduced in one phase are propagated to later phases; sometimes those defects are caught relatively soon, in the next downstream phase, but in many cases they may contribute to erroneous representations of the system in multiple phases before being caught. For example, a problem in the software specifications may not be caught until implementation, possibly requiring changes to the specification, design, and code. A key benefit of introducing inspection is that it reduces the number of defects allowed to propagate downstream; some estimates calculate that inspections allow only 0.3% to 10% of defects to make it out of a given lifecycle phase [Wheeler, Brykcznski, Meeson, 1996].

Along with this gain in quality, there is a commensurate gain in productivity, since the number of defects found in the test phase is along with the overall quantity of rework.

All these benefits cost approximately 15% of the project's total value, the time and effort required to adequately support inspections. Nevertheless, the gain in time and productivity and the profits with this economy compensate the expenditures.

Figure 2 shows, on average, the development costs of a piece of software in its many phases. We can observe that the greatest part of a system's costs is in integration, and only a small part is invested in the elicitation of requirements with the user. One of the proposed benefits due inspections is that if an inspection on the project and analysis requirements research phases is accomplished, the costs will increase in the requirements phase, but they will decrease greatly in the test and integration phase.

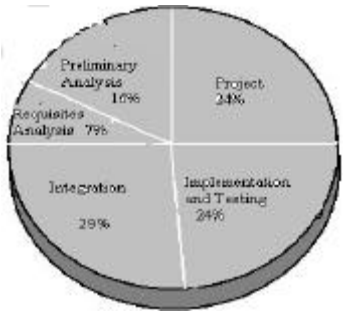

Figure 2: Software development costs [Wheeler, Brykczski and Meeson, 1996]

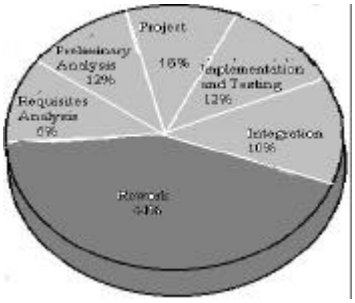

Figure 3: Rework total cost [Wheeler, Brykczski and Meeson, 1996]

Figure 3 shows that, of all work done during the development of a piece of software, 44% of effort and costs is spent with rework. This leads to the observation that it is necessary to change the way software is being developed, since a large percentage of effort is being spent on the correction of effort that was spent previously and, as already shown in

Figure 1, a mistake that passes from one phase to another increases costs out of proportion to its original impact on the phase of origin.

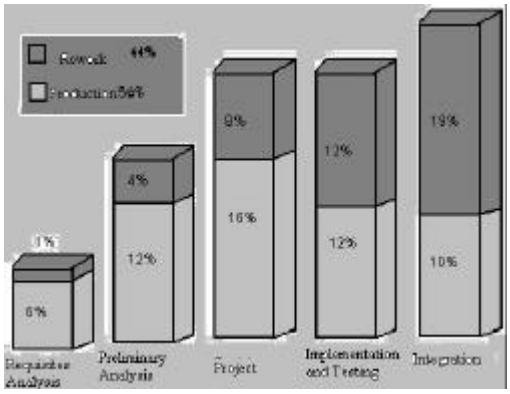

Figure 4: Rework distribution during development [Wheeler, Brykczski and Meeson, 1996]

Figure 4 describes the same information as Figure 3, but it is divided by phases. Thus, rework is small in the requirements phase, but is huge in relation to the test and integration phase. It can be said that, in this phase, twice as much time is spent on corrections than on development.

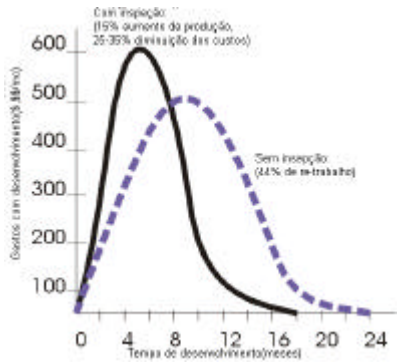

Figure 5: Software dev. Expenditures with and without inspections

Figure 5 shows graphs expenditures with inspection (in black) and without inspection (in blue). We can see that although costs are larger in the initial phase in when inspections are used, less is spent over the course of the project because less time and effort needs to be spent on reword, and because defects are not allowed to propagate and grow from one phase to another.

Organization	Product	Type of product inspected	Results
AT&T	Telecommunications systems	Requirements, design, code and test	Inspection has increased productivity and quality by 14%, being 20x more efficient than the test.
HP	(Varied)	design, code, test, documentation	An audit revealed an ineffective inspection process. Problems under discussion.
		Code	2 mistakes detected per hour. It is unlikely that 80% of defects could be caught by tests.
BRN	Telecommunications systems	Code	1 defect detected per hour. The process was 20x more efficient than test.
BullHN Information Systems	Operating system	Requirements, design, code, test, documentation.	The team composed of 4 people was twice as efficient as the one composed of 3.
IBM	Operating systems	design and code	23% increase in code productivity and 38% reduction of mistakes found in test phase.
ICL	Operating systems	design	40% to 50% increase in defect detection. 1.2 hours per defect in inspection compared to 8.4 hours with the test.
JPL	Space systems	Requirements, design, code, test	0.5 hours to find mistakes versus 5 hours for other techniques.
MEL	(Varied)	design, code	Return on investment calculated at 8:1. In 75 inspections the result was 7000 hours saved.
Shell Research	Geophysical software	Requirements	1 mistake found every 3 minutes. Return on investment calculated at 30:1.

Table 3: Benefits of inspection - Wheeler et al (1996)

4.2 Inspection Metrics

Implementing an inspection process is even more beneficial when metrics are collected on its performance to compose a qualitative record of the work. This kind of data has a fundamental importance, since it allows us to evaluate if the inspection process is really being effective, if it is finding the defect it should. Moreover, it is possible to act on the data to improve the process; for example, to offer focused training to software engineers based on perceived weaknesses.

This advantage can be ruined, however, if the data is used for purposes other than those for which it was intended. One common danger is that management may want to use the data collected to evaluate an individual who performs at a level below that which is desired. It is up to the inspection staff to address this type of problem, addressing the issues in their presentations and training courses so that the data collected will be used for the organization's growth and not to punish individuals.

Through the analysis of inspection data, it is possible to identify the areas with potential problems and to evaluate the effects of progress on a long-term basis. The collected data fall into three categories:

- Defects: By analyzing the types of defects that are found in software artifacts, it is possible to analyze their cause and thus to prevent new mistakes of the same type during development. Moreover, the defects detected are the primary measure of the inspection process' effectiveness.

- Effort: The time spent per inspection is always analyzed and compared to the effort spent on rework. The effort relative to the inspection work must always be less than the rework effort. If this is not true, the inspection process is not providing satisfactory results.

- Deliverable size and type. The deliverable to be analyzed must be less than a predetermined size. Larger deliverables should be divided into smaller one, which can be inspected separately. This division is based on the deliverable type. For example, business requirements can be measured in pages or amount of business functions; source code can be measured in number of lines of source code (NLSC), etc.

The various inspection experiences described by diverse authors such as Briand, El Elman and Melo (1999), Fagan (1976), Weller (1993), Wheeler, Brykczxynski, Messon (1996), support some common observations:

> **Compromise**: The inspection process requires expenditures on personnel training and data collection. For the inspection benefits to compensate for the necessary expenditures, the process should be followed and constantly perfected, throughout the lifecycle. The naïve idea that many managers have that going straight to code inspection saves time causes the process to be less effective than it could be.
>
> **Pilot program**: A pilot program is an effective approach for introducing the inspection process in an organization. Through the pilot project, people get familiarized with the inspection process and difficulties that might are detected quickly, allowing wider dissemination throughout the organization to be done more effectively.
>
> **Data**: Data analysis is what will provide a qualitative result from the inspection. Through selected data it is possible to check the inspection's final output, that is, the defects found. It is also needed to provide feedback to project managers, especially important when inspections represent a new way of working. Finally, data analysis is also important so that false positives can be identified, and developers and managers can be trained to avoid them.

There are many ways of extracting inspection metrics. The most common ones are instant measures and defect removal technique's relative leverage.

4.2.1 Instant measures

It is not possible to calculate the full effectiveness of reviews on a project before development work is finished and the set of all defects is understood. When development work is complete, it is possible to understand the contribution of reviews at the greatest level of detail.

However, defects found during a particular phase can be calculated during review and used to evaluate review quality. "Snapshots" of this type allow, for instance, a comparison between the defects found between design and code review. Nevertheless, it is hard to say if review is being effective or superficial with this type of measure.

The average rate of defects found per hour indicates the effectiveness of time spent on review. For example, if the number of defects found per hour is decreasing and no new material is being created, continued review can be expected to be unproductive.

4.2.2 Defect Relative Leverage of the Defect Removal Technique (DRL)

DRL is used to evaluate the defect removal efficiency. This metric is calculated as the relationship between the average number of defects removed per hour in two lifecycle phases. It is used more in lifecycles that contain many review activities and a test phase. For example: in code review, 9.95 defects were found per hour and in the unit test, 1.64. Therefore, DRL is 9.95 / 1.64 = 6.07

The great advantage of this calculation is that it provides the ability to analyze the effectiveness of the several defect removal phases.

Other aspects may be analyzed for inspection metrics, such as:

- The average preparation effort per unit of material;
- The average examination effort per unit of material;
- The average defect number found per unit of material;
- The average hours spent to find a defect;
- The average number of serious defects found per unit of material;
- The average number of hours required finding a serious defect.

5 Inspection Process

Software inspection is the most detailed way to perform a review. When an inspection process begins, all documentation is thoroughly studied. Work is inspected in its many phases and necessary corrections are made in each phase in order for the process to be completed.

There are six main steps in an inspection Fagan (1976):

- Planning: When a particular deliverable or working product is created, an inspection team is designated to review it and a moderator is indicated. The moderator will check if the provided material is enough for an inspection. The moderator and the developer sign rules as to how the process will take place, and then the material is distributed and the schedule defined.
- Presentation: In this step the deliverable author(s) will present the main characteristics of the deliverable, its scope and objectives and answer any questions inspectors might have. If the inspection staff know what the project is about and have an understanding of the deliverable, this step can be skipped.
- Preparation: Inspectors study individually to understand the deliverable and its context. To do so, reading procedures can be provided. Also, checklists are often used to help in the detection of mistakes.

- Meeting: In this phase, inspectors review the project together. All reviewers should come prepared, completely aware of how the deliverable should work. A person is chosen to conduct the meeting, known as the scriber. Potential defects are discussed, but not their solutions. This process should not last more than two hours, since after this time the inspectors' concentration and analysis capacity usually decreases drastically. In case there is the need for a meeting to last more than two hours, it is suggested the inspection work continues the next day.

- Rework: The author corrects the mistakes found by the inspectors and confirmed by the moderator. The moderator can consider some defects as false positives: they have been marked as defects, but during the inspection meeting they have been considered as correct information.

- Follow-up: The material corrected by the authors is resubmitted to the moderator. He reevaluates the quality, or calls for a new inspection if the rate of defects found on the first inspection is greater than 5% of project defect average (This 5% rate is a number suggested by Fagan [1976], but it should not be taken as a hard rule. The moderator has the freedom to choose whether s/he is going to ask for a new meeting).

In addition to these steps, the inspection should be undertaken using the following general guidelines:

- Objectives

  Inspection should be focused on finding defects in the analyzed deliverable or working products. Discussions on the best method to correct the problem or how the product should have been developed should not happen during inspection; since they make it less objective and can prolong the discussion on problems outside the purview of the inspection team.

- Inspection Staff

  The inspection team should be composed of a small group. Fagan (1976) suggests that a group of 6 people would be ideal, but groups of 4 or 5 people are also very common. This team must have a moderator, a person who will act as a group "reporter", and other technical people who can take part in the inspection. Very large groups tend not to be productive. Fagan (1976) suggests that the group of inspectors should be diverse, that is, it should include at least one quality control person, developer, and programmer; the exact numbers should vary according to what is being analyzed. Recently, Basili and his colleagues proposed a perspective-based team composed of the following inspecting roles: designer, tester, and customer. For example, if an analysis document is being inspected, there should be more analysts than programmers, but there should be at least one programmer in the group. Experience from many different enterprises that use this technique shows that this suggestion increases productivity and group vision.

  In general, managers should not participate in the inspection group, so that the group is not inhibited to expose the defects in a colleague's work. The most effective way the manager can help in this process is by motivating his employees to develop a good project, and motivating the inspection process inside the firm. However, if the group decides the manager's presence will help and others will not feel pressured or inhibited, there are no barriers to his or her participation.

- Rules

The project's success is based on a set of rules that must be followed as strictly as possible. These rules should be for the moderator, the reporter, for the author and the rest of the inspectors. An important rule for the author is that he should never be the moderator or the reporter. In reality, although this does not always happen, the author should not participate in the inspection process.

Qualified moderators are essential for the success of the inspection process. The moderator should coordinate the time the inspectors will have for analysis, should evaluate the staff's progress, define and not allow members to deviate their attention from the main focus, and deal with the author's corrections. The moderator should be someone experienced in the inspection process, which knows well the rules for each of the other roles. This is the only way the moderator will be able to assure these rules are followed.

Moderators should take the time to help and, most of all, to motivate the inspection team, since this motivation is what will guarantee that defects are effectively found. To do so, it is necessary that the moderator has total control over the technique and knows how to deal with the people involved in the process. It is necessary that the moderator first get other inspectors accustomed to working as a group and only afterwards demands a high productivity level. Motivating people to the project that will be developed and making them realize the importance of what is being done conditions this behavior.

- Deliverables

  Inspection can be used in many different phases of software development, from requirements to implementation. Early inspection are performed, better it will be.

- Outputs

  The two main outputs to the inspection are:

  1. A list of defects that need to be corrected;

  2. A report describing the execution of the inspection process, who participated in it, when it was done and what were the group conclusions concerning the inspected deliverable(s);

The inspection process should not be something done spontaneously. It must be planned and refined by an enterprise or a group that has control over the technique and the responsibility for its application. This group's job is:

- To learn about the inspection process;
- To determine in which phase of the project the inspection will be used;
- To document the inspection proceedings;
- To organize the inspection process, prepare documentation on it and offer training to the people who are going to work with this technique;
- To collect inspection data for the record of this process;
- To give answers based on the information record;

- To analyze the record comparing results across many projects and making the necessary recommendations based on this data.

Defining the types of defect expected to be found is an important part of inspection support. Such expectations should be used to focus the documentation, training, and job aides such as checklists.

6 Reading-Driven Inspection Techniques

According to Basili et al. (1996), a reading technique is a series of steps for the individual analysis of a software product to achieve the understanding needed for a particular task. Basili and his colleagues have showed that reading techniques increase the effectiveness of individual reviewers by providing guidelines that they can use, during the detection phase of a software inspection, to examine (or "read") a given software document and identify defects. Techniques attempt, thus, to capture knowledge about best practices for defect detection into a procedure that can be followed.

In this section, we present two reading techniques that directly support the production of quality software designs: Perspective-Based Reading (PBR), which ensures that the deliverables used as input to the design phase are of high quality, and Object-Oriented Reading Techniques (OORT's), which evaluate the quality of the design deliverables themselves.

6.1 A Requirements Inspection Technique: Perspective-Based Reading (PBR)

A set of inspection techniques known as Perspective-Based Reading (PBR) was created for the domain of requirements inspections [Shull et al, 2000]. PBR is designed to help reviewers answer the following questions about the requirements they are inspecting:

- How do I know what information in these requirements is important to be verified?
- Once I have found the important information, how do I find defects in that information?

PBR exploits the observation that different information in the requirements is more or less important for the different uses of the document. That is, the ultimate purpose of a requirement document is to be used by a number of different people to support tasks throughout the development lifecycle. Conceivably, each of those persons finds different aspects of the requirements important for accomplishing his or her task. If we could ask all of the different people who use the requirements to review it from their own point of view, then we would expect that all together they would have reviewed the whole document (since any information in the document is presumably there to help somebody do his or her job).

Thus, in a PBR inspection each reviewer on a team is asked to take the perspective of a specific user of the requirements being reviewed. His or her responsibility is to create a high-level version of the work products that a user of the requirements would have to create as part of his or her normal work activities. For example, in a simple model of the software lifecycle we could expect the requirements document to have three main uses in the software lifecycle:

- As a description of the needs of the customer: The requirements describe the set of functionality and performance constraints that must be met by the final system.

- As a basis for the design of the system: The system designer has to create a design that can achieve the functionality described by the requirements, within the allowed constraints.

- As a point of comparison for system test: The system's test plan has to ensure that the functionality and performance requirements have been correctly implemented.

In such an environment, a PBR inspection of the requirements would ensure that each reviewer evaluated the document from one of those perspectives, creating some model of the requirements to help focus their inspection: an enumeration of the functionality described by the requirements, a high-level design of the system, and a test plan for the system, respectively. The objective is not to duplicate work done at other points of the software development process, but to create representations that can be used as a basis for the later creation of more specific work products and that can reveal how well the requirements can support the necessary tasks.

Once reviewers have created relevant representations of the requirements, they still need to determine what defects may exist. To facilitate that task, the PBR techniques provide a set of questions tailored to each step of the procedure for creating the representation. As the reviewer goes through the steps of constructing the representation, he or she is asked to answer a series of questions about the work being done. There is one question for every applicable type of defect. When the requirements do not provide enough information to answer the questions, this is usually a good indication that they do not provide enough information to support the user of the requirements, either. This situation should lead to one or more defects being reported so that they can be fixed before the requirements need to be used to support that task later in the product lifecycle.

6.2 Design Inspection Techniques: Object-Oriented Reading Techniques (OORT's)

In PBR, reviewers are asked to develop abstractions, from different points of view, of the system described by the requirements because requirements notations do not always facilitate the identification of important information and location of defects by an inspector. For an OO design, in contrast, the abstractions of important information already exist: the information has already been described in a number of separate models or diagrams (e.g. state machines, class diagrams) as discussed at the end of the previous section.

However, the information in the abstractions has to be checked for defects, and reading techniques can still supply a benefit by providing a procedure for individual inspection of the different diagrams, although unique properties of the OO paradigm must be addressed. In an object-oriented design we have graphical representations of the domain concepts instead of the natural-language representation found in the requirements document. Another feature of object-oriented designs that has to be accounted for is the fact that while the different documents within the design all represent the system, they present different views of the information.

The PBR techniques for requirements are concerned mainly with checking the correctness of the document itself (making sure the document was internally consistent and clearly expressed, and whether the contents did not contradict any domain knowledge). A major difference in the OORT's is that for checking the correctness of a design, the reading process must be twofold. As in requirements inspection, the correctness and consistency of the design diagrams themselves must of course be verified (through "horizontal reading") to ensure a consistent document. But a frame of reference is necessary in order to assess design correctness.

Thus it is also necessary to verify the consistency between design artifacts and the system requirements (through "vertical reading"), to ensure that the system design is correct with respect to the functional requirements.

The OORT's [Travassos et al, 1999] consist of a family of techniques in which a separate technique has been defined for each set of diagrams that could usefully be compared against each other. For example, sequence diagrams need to be compared to state machines to detect whether, for a specific object, there are events, constraints or data (described in the state machine) that could change the way that messages are sent to it (as specified in the sequence diagram). The advantage of this approach is that a project engaged in design inspections can select from this family only the subset of techniques that correspond to the subset of artifacts they are using, or that are particularly important for a given project.

In order to provide specific reading procedures for the different kinds of UML diagrams, Travassos and his colleagues (2000) have created two different set of reading techniques: the vertical reading and the horizontal reading. In the vertical reading, the reviewer is asked to compare specification models (requirement specification and use cases) against design models (Object-Class, Sequence and state diagrams). In the horizontal reading, the reviewer is asked to compare both design models against design model and specification requirement against use cases. Figure 6 shows the different UML artifacts and the techniques that are available to review them. A detailed description about the development of OORT's including the version 2 of all the techniques can be found in (http://fc-md.umd.edu/reading/reading.html). A complete description about a design process exploring both PBR and OORT's techniques can be found in [Travassos et al, 2001].

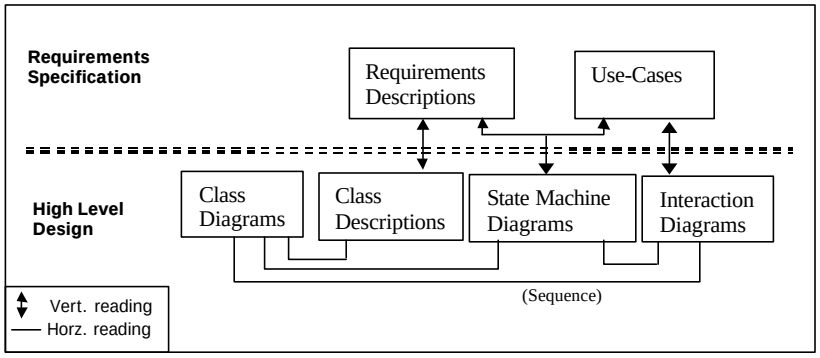

Figure 6 – Reading Techniques for OO Design

7 Case Study

In this section, we will describe a case study where OORT's techniques were applied. This inspection was performed in professional environment at Oracle Brazil.

7.1 Project description:

This system is responsible for the control of the taxes by the Mato Grosso State's Secretary of Finance. The objective is to receive and allow the contributor to make commercial tax declaration over merchants and services through the Internet. It will also be necessary to treat data received by

them. The contributors without access to the Internet can make their declarations through magnetic disk.

7.2 Inspectors team:

A team of 5 people from the Oracle Brazil, Enterprise Solution, Software Quality Engineering Group performed this inspection job. All five participants have knowledge on UML (in different levels). A tutorial on inspection was made previous to the inspection job.

7.3 Material

After the material developed by the client was sent to Oracle for quality assurance purposes, a meeting to present the problem was scheduled. After this meeting, the auditors, who already had previous knowledge of the technique, received the material (which were use cases, classes description, classes diagram, and sequences diagrams) and the inspection material (inspection forms and defect reports cards). From all the reviewers, one, who had greater control over the technique, stayed put in order to solve any questions that could have come up. (These questions would have to be pertinent to the inspection technique, not to the system being analyzed).

7.3.1 Inspection Data:

After data analysis, the following results were obtained:

Maximum number of defects by inspector:	57
Minimum number of defects by inspector:	12
Average of defects:	35

Comments: The person who found only 12 defects was not too familiarized with the inspection technique. Therefore, this person had a greater difficulty to find the defects. The person who found 57 mistakes is a person familiarized the inspection technique, having already applied this technique in other projects.

Maximum number of false positives:	3
Minimum number of false positives:	0
Average of false positives:	0.8

Comments: False positives are data that are only apparently wrong. They alter something that is not wrong, generating a defect. The low number of false positives shows that the technique was well applied, and their presence shows how the moderator is important, since he is the one who will take all the false positives away.

Maximum time spent in the inspection process:	8 hours
Maximum time by defect:	40 minutes
Minimum time spent in the inspection process:	3 hours
Minimum time by defect:	2.10 minutes
Average of time spent in the inspection process:	5.8 hours
Average time by defect:	15 minutes

Comments: The great variation of time in the inspection is due to the familiarity each one has with the technique, as well as the variation of time by the defect is due to the knowledge on the inspection technique and the UML.

In relation to the tables below, it is important to mention the normalization of the collected data. The amount of defects showed in each table is the actual amount of defects, that is, the defects found by the two inspectors are being counted only once instead of twice, according to the formula below:

$$D_i = \{ d_1, \ldots, d_n \} ; E = \cup D_i; QTE = |E|,$$

where:

- d_j is a defect;
- D_i denotes the set of defects found by the inspector i,
- and E is the union of all defects, so, let us suppose two defects, d_i and d_j, belonging to E, if $d_i = d_j$ then i=j.

Type of defect:

Type	Omission	Incorrect Fact	Ambiguity	Inconsistency
# of Defects	50	7	6	8

Guideline used to detect the defect

Place	Sequence Diagram (SD)	Class Description (CD)	Use Cases (UC)	Sequence Diagram vs. Use Cases	Sequence Diagram vs. Class Description
# of Defects	4	13	15	32	2

Defect localization

Place	Use Cases	Sequence Diagram	Class Description
# of Defects	20	45	14

8 Conclusion

In this report, we have showed that software inspection has proved to be both effective and efficient. Many defects could be found at early phases of the project life cycle. The effort of finding such defects was minimal.

As commented, checklists and reading procedures are at the heard of a successful inspection process. We have provided specific reading procedures and checklists for UML based deliverables. We also provided a case study in which these procedures have applied with considerable success.

We have also point out the importance of an expert moderator. In moments of conflicts and doubts among inspectors, the moderator is of fundamental importance for the solving of the impasse.

Moreover, the moderator is the one who consolidates data. Consolidation is fundamental, since the developer will have a more precise and direct result, and metrics and the record will be done after it.

It is also important to have in mind that inspection, and any other form of revision, is not a miraculous formula to solve problems. It is one more step in the process and it needs to be well monitored, have its results criticized and evaluated, or else, many doubtful results will be taken as truth, generating problems to the project and discrediting the technique.

9 Bibliography

1. Ackeman, A. Frank & Buchwald, Lynne S. & Lewski, Frank H. *Software Inspections: An Effective Verification Process* IEEE software, Vol. 6 Number 3 May 1989 p. 31-36
2. Basili, Victor R. & Selby, Richard W. *Comparing the Effectiveness of Software Techniques Strategies,* IEEE software, Vol. 13 Number 12 Dec 1987 p. 278-296
3. Basili, V. R., Green S., Laitenberger, O., Lanubile, F., Shull, F., Sorumgard, S., Zelkowitz, M. V.. The Empirical Investigation of Perspective-Based Reading, Empirical Software Engineering Journal, I, 133-164, 1996
4. Blakely, F.W & Boles, M.E. *A Case Study of Code Inspections.* Hewlett-Packard Vol. 42, Number 4 Oct 1991, p. 58-63
5. El Eman, K. and Madhavji, N. H, (eds.), Elements of Software Process Assessment and Improvement. 1999. IEEE Press
6. Carver, J. Impact of Background and Experience on Software Inspections. University of Maryland, Computer Science Dep., College Park, MD, Ph.D. Thesis Proposal. 1999.
7. Deutsch, Michael S. & Willis, Ronald R. *software Quality Engineering a total technical and management approach.* Prentice Hall, 1st edition, 1988.
8. Doolan, E.P. *Experience with Fagan´s Inspection Method* Software – Practice and Experience Vol. 22, Number 2 Feb 1992, p. 173-182
6. Fagan, Michael. E. *Design and code inspections to reduce errors in program development IBM System Magazin.* 15(3): 182-211, 1976.
7. Fagan, Michael E. *Advances in software Inspections.* IEEE TSE. Vol. 12, Number 7, 1986, p. 744-751.
8. Fowler, Priscilla J. *In-Process Inspections of Workproducts at AT&T* AT&T Technical J. Vol. 65, Number 2 Mar/Apr 1986 p. 102-112
9. Humphrey, W.S., *Managing the Software Process,* Reading Mass, 1989
10. Michaelis *Modern dictionary of the Portuguese Language,* Reader's Digest Brazil, 1998.
11. Pressman, Roger S. *Software Engineering A Practitioner's Approach* McGraw Hill, 4th edition, 1997.
12. Russell, Glen W. *Experience with inspection in ultra large-scale developments.* IEEE software Vol. 8, Number 1, 1991, p.25-31.
13. Russell, Glen W. & Research, Bell Northern *Experience with Inspection in Ultra large-Scale Developments* IEEE software, Vol. 8 Number 1, Jan 1991 p. 25-31
14. Shull, F.; Rus, I.; and Basili, V.R. How Perspective-Based Reading Can Improve Requirements Inspections. *IEEE Computer* 33, 7 (July 2000), 73-79
14. Travassos, G. H., Shull, F., Fredericks, M., Basili, V. R. Detecting Defects in Object Oriented Designs: Using Reading Techniques to increase Software Quality. ACM SIGPLAN Notices, USA, v.34, n.10, p.47-56, 1999. ISSN 0362-1340
15. Travassos, G. H., Shull, F., Carver, J. A Family of Reading Techniques for OO Design Inspections**.** In: WQS'2000 - WORKSHOP QUALIDADE DE SOFTWARE, 2000, Joao Pessoa. Proceedings of SBES'2000 - Workshops. Joao Pessoa: Brazilian Computer Society, 2000. v.1. p.225-237
16. Travassos, G. H., Shull, F., Carver, J. Working with UML: A Software Design Process Based on Inspections for the Unified Modeling Language. In: Advances in Computers. 2001, v.54. pp. 35-98. ISBN 0-12-012154-9
17. Weller, Edward F. *Lessons from three years of inspection.* IEEE software Vol. 10, Number 5, 1993, p.38-45.
18. Wheeler, D. A.; Brykczynski, B. and Meeson, R. N. (eds.). *Software inspection: An Industry Best Practice.* IEEE Computer Society; ISBN: 081867340. 1996.