ELSEVIER

The Journal of Systems and Software 50 (2000) 5–31

The Journal of Systems and Software

www.elsevier.com/locate/jss

An encompassing life cycle centric survey of software inspection

Oliver Laitenberger [a,*], Jean-Marc DeBaud [b,1]

[a] Fraunhofer Institute for Experimental Software Engineering, Sauerwiesen 6, 67661 Kaiserslautern, USA
[b] Lucent Software Product Line Laboratories, 263 Sherman Boulevard, Room 2R-324, Naperville, IL 60566-7050, USA

Received 15 December 1998; received in revised form 1 February 1999; accepted 10 March 1999

Abstract

This paper contributes an integrated survey of the work in the area of software inspection. It consists of two main sections. The first one introduces a detailed description of the core concepts and relationships that together define the field of software inspection. The second one elaborates a taxonomy that uses a generic development life-cycle to contextualize software inspection in detail.

After Fagan's seminal work presented in 1976, the body of work in software inspection has greatly increased and reached measured maturity. Yet, there is still no encompassing and systematic view of this research body driven from a life-cycle perspective. This perspective is important since inspection methods and refinements are most often aligned to particular life-cycle artifacts. It also provides practitioners with a roadmap available in their terms.

To provide a systematic and encompassing view of the research and practice body in software inspection, the contribution of this survey is, in a first step, to introduce in detail the core concepts and relationships that together embody the field of software inspection. This lays out the field key ideas and benefits and elicits a common vocabulary. There, we make a strong effort to unify the relevant vocabulary used in available literature sources. In a second step, we use this vocabulary to build a contextual map of the field in the form of a taxonomy indexed by the different development stages of a generic process. This contextual map can guide practitioners and focus their attention on the inspection work most relevant to the introduction or development of inspections at the level of their particular development stage; or to help motivate the use of software inspection earlier in their development cycle.

Our work provides three distinct, practical benefits: First, the index taxonomy can help practitioners identify inspection experience directly related to a particular life-cycle stage. Second, our work allows structuring of the large amount of published inspection work. Third, such taxonomy can help researchers compare and assess existing inspection methods and refinements to identify fruitful areas of future work.

1. Introduction

In the past two decades, software inspections have emerged as one of most effective quality assurance techniques in software engineering. The primary goal of an inspection is to detect defects before the testing phase begins; and hence strongly contribute to improve the overall quality of software with the corollary budget and time benefits (DeMarco, 1982; Yourdon, 1997). In this article, we consider inspection to be an approach involving a well-defined and disciplined process in which qualified personnel analyse a software product using a reading technique for the purpose of detecting defects. A defect is considered any deviation from predefined quality properties (this includes the functional ones). This definition of an inspection is broader in scope than the one originally provided by Fagan (1976) [2], which allows us to discuss each inspection variation in detail. However, the scope of this survey is limited in so far as we do not consider the work on other static analysis techniques, such as audits or walkthroughs, although these techniques may be equally or in some situations even more effective for achieving the stated goal. The limitation stems from the fact that these approaches are often less structured and vary a lot, which makes it difficult to compare them scientifically to software inspection. A more general discussion on the differences of these techniques is provided, for example, in Marciniak (1994).

* Corresponding author.
E-mail addresses: oliver.laitenberger@iese.fhg.de (O. Laitenberger), debaud@research.bell-labs.com (J.-M. DeBaud).

[1] This work was done while Dr. Jean-Marc DeBaud was at the Fraunhofer Institute for Experimental Software Engineering.

[2] We acknowledge that others may prefer the term 'Formal Technical Review (FTR)' or 'Review' to account for variations to the Fagan-style of inspection.

0164-1212/00/$ - see front matter
PII: S0164-1212(99)00073-4

After Fagan's seminal introduction of the generic notion of inspection to the software domain at IBM in the early 1970s (Fagan, 1976), a large body of contributions in the form of new methodologies and/or incremental improvements has been proposed promising to leverage and amplify inspection's benefits within software development and even maintenance projects. However, most of the published work has not been integrated into a broader context, that is, into a coherent body of knowledge taken from a life-cycle point of view, hence making the work difficult to reconcile and evaluate in specific life-cycle conditions. This may be one reason why practitioners still are asking questions of the following type: What are the key differences among the currently available inspection approaches? On which part of the life-cycle can these approaches be applied? What have been their documented effects at those stages? What category of effects can these approaches have on a project, or an organization? What are the qualitative and quantitative results to support the claims? What type of tools are available to support inspections? The lack of clear, consolidated answers to these questions might be a reason as to why, so far, inspections have not fully and effectively penetrated the software industry (Johnson, 1998b). Yet, the fact that at least some form of inspection has become a necessity for the CMM(TM) and ISO-9000 certification increases the pressure to answer the stated questions. We believe that at least some of them can be addressed by integrating existing research and practice work into a coherent body of knowledge viewed from a life-cycle angle.

Findings about inspections have not been easy to reconcile and consolidate due to the sheer volume of work already published. Hence, it is not surprising that the available surveys (Kim et al., 1995; Mac-donald and Miller, 1995; Porter et al., 1995a; Tjahjono, 1996; Wheeler et al., 1997) only cover the most relevant published research in their reviews.

Broadly speaking, existing surveys can be summarized as follows: Kim et al. (1995) present a framework for software development technical reviews including software inspection (Fagan, 1976), Freedman and Weinberg's technical review (Weinberg and Freedman, 1984), and Your don's structured walkthrough (Yourdon, 1989). They segment the framework according to aims and benefits of reviews, human elements, review process, review outputs, and other matters. Macdonald et al. (1996a) describe the scope of support for the currently available inspection process and review tools. Porter et al. (1995a) focus their attention on the organizational attributes of the software inspection process, such as the team size or the number of sessions, to understand how these attributes influence the costs and benefits of software inspection. Wheeler et al. (1997) discuss the software inspection process as a particular type of peer review process and elaborate the differences between software inspection, walkthroughs, and other peer review processes. Tjahjono (1996) presents a framework for formal technical reviews (FTR) including objective, collaboration, roles, synchronicity, technique, and entry/exit-criteria as dimensions. Tjahjono's framework aims at determining the similarities and differences between the review process of different FTR methods, as well as identifying potential review success factors. All of these surveys contribute to the knowledge of software inspection by identifying factors that may impact inspection success. However, none of them present its findings from a software life-cycle phases perspective. This makes it difficult for practitioners to determine which inspection method or refinement to choose, should they want to introduce inspection or improve on their current inspection approach.

To tackle this problem, it is the goal and hence the stated contribution of this article to portray the status of research *and* practice as published in available software inspection publications from a life-cycle angle and present the facts as reported in the literature. For this purpose, we performed an extensive literature survey including a wide source of publications. The survey consists of two principal sections. The first includes a taxonomy of the core concepts and relationships that together embody the notion of software inspection. This taxonomy is centered around five primary dimensions – technical, managerial, organizational, economics, and tools – with which we attempt to characterize the nature of software inspection. While these primary dimensions are most relevant for the major areas of software development, we elicited from the literature particular subdimensions that are principal for work in the software inspection area. In the second section, the survey introduces an idealized life-cycle taxonomy contextualizing software inspection to each main life-cycle development phase, taking into account their specific particularities. This can make it much easier for practitioners, in a given life-cycle phase, to get an overview of relevant inspection work including its empirical validation. Of course, considering the large volume of published work in the area of software inspection, it is impossible to integrate each and every article in this survey. Hence, we decided to include only significant contributions to the field, that is, we excluded, for example, most opinion papers.

Practitioners as well as researchers can profit from this survey in three different ways: First, the survey provides a road-map in the form of a contextualized, life-cycle taxonomy that allows the identification of available inspection methods and experience directly related to a particular life-cycle phase. This may be particularly interesting for practitioners since they often want to tackle quality deficiencies of concrete life-cycle artifacts with software inspection. Yet, they often do not know which method or refinements are available and

which ones to choose. Hence, this survey helps them focus quickly on the best-suited inspection approach adapted to their particular environment via the life-cycle driven taxonomy. Second, our work helps to structure the large amount of published inspection work. This structure allows us to present the gist of the inspection work performed so far and helps practitioners as well as researchers characterize the nature of new work in the inspection field. In a sense, this structure also helps define a common vocabulary that depicts the domain of software inspection. Third, our survey presents an overview of the current state of research as well as an analysis of today's knowledge in the field of software inspection. The condensed view on the published work allows us to distill a theory in the form of three causal models. These models, together with the road map, may be particularly interesting to researchers for identifying areas where little methodological and empirical work has been done so far.

We structured this survey as follows. Section 2 presents the study methodology including the approach we followed to identify and select the relevant software inspection literature. Section 3 describes the core concepts and relationships that together define the notion of software inspection. Section 4 details them. Section 5 introduces a generic software life-cycle model to provide the context for the situational inspection taxonomy. Section 6 presents the latter. Section 7 integrates the concepts and relationships into a theory to point out possible future research directions. Section 8 concludes.

2. Study methodology

Literature surveys have long played a central role in the accumulation of scientific knowledge. As science is a cumulative endeavor, any one theory or finding is suspect because of the large array of validity threats that must be ruled out. Moreover, all too often new techniques and methods are proposed and introduced, without building on the extensive body of knowledge that is incorporated in the ones already available. These problems can be somewhat alleviated by establishing the current facts using the mechanism of literature survey. The facts are the dependable relationships among the various concepts that occur despite any biases that may be present in particular studies because of the implicit theories behind the investigators' choice of observations, measures, and instruments. Hence, a literature survey makes the implicit theories explicit by identifying their commonalities and differences, often from a specific angle when the body of knowledge has become very rich. In some cases, a literature survey may even be an impetus for the unification of existing theories to induce a new, more general theory that can be empirically tested afterwards.

To achieve these goals, a survey must fulfill several principles: First, it must be well-contained, that is, encapsulate its work within a clearly defined scope where the benefits of doing so can be well-understood and accepted. Second, a survey must provide profound breadth and depth regarding the literature relevant to its defined scope. Finally, it must present a unified vocabulary reconciling the most important terms in a field.

The first principle is the hardest to fulfill and illustrates the fact that there cannot be one single method for developing a survey since it is so tightly coupled with the notion of scope. The scope is, in fact, what defines the gist of a survey and hence, depending on the particular interest of the authors, a survey can be geared in different directions. This is clearly illustrated by the different directions taken by the four surveys we mentioned in the section above. Each used a particular scope and rationales for its motivation. In our case, the scope has been defined as filtering the software inspection literature work from the angle of the life-cycle phases. We believe this perspective can be well understood and accepted because of the benefits outlined in the above sections.

To fulfill the second and third survey principles, finding and selecting the relevant literature is of utmost importance. We attempted to collect any publication fitting our definition of inspection which captures, we believe, the essence of other definitions. However, no single method for locating relevant literature is perfect (Cooper, 1982). Hence, we utilized a combination of methods to locate articles and papers on our subject.

We conducted researches of the following two inspection libraries: Bill Brykczynski's collection of inspection literature (Brykczynski and Wheeler, 1993; Wheeler et al., 1996) and the FTR Library (Johnson, 1998a). To be sure not to miss a paper recently published, we performed three additional steps in search of inspection articles: First, we employed a keyword search in the INSPECT database of the OCLC (1998) and the library of the Association of Computing Machinery (1998) using the keyword 'software inspection'. Second, we manually searched the following journals published between 1990 and July 1997: IEEE Transactions on Software Engineering, IEEE Software, Journal of Systems & Software, Communications of the ACM, and ACM Software Engineering Notes. Finally, we looked at the reference sections of books dealing with software inspection (Gilb and Graham, 1993; Strauss and Ebenau, 1993) and manually searched the library of the International Software Engineering Research Network (1998) and the proceedings of the NASA-Goddard Software Engineering Workshop from 1984 to 1997. Table 1 shows the results of our literature search. The reader must keep in mind that some articles are cross-referenced among several

Table 1
Summary of search results

Source	Number of articles
Literature in Wheeler et al. (1996)	147
FTR-library	204
OCLC database	55
ACM database	21
IEEE transactions on software engineering	10
IEEE software	9
Journal of Systems & Software	4
Communications of the ACM	4
ACM software engineering notes	3
Other (e.g., ISERN-reports)	22

libraries. We made the results of our literature search available on-line (Fraunhofer Institute for Experimental Software Engineering, 1998).

Considering the very large number of published articles available, it was impossible to give full attention to every article within this survey, although we carefully considered each and every one of them. We excluded articles based on the following rules: (a) the article is an opinion paper and, therefore, does not represent tangible inspection experiences, (b) it takes considerable effort (money or time) to get an article, (c) one or several authors published several papers about similar work in journals and conference proceedings – in this case, we considered the most relevant journal publication –, (d) an article does only provide a weak research or practical contribution, though we acknowledge the subjectivity of this criteria. However, we avoided the dangers of ignoring papers because they do not fit neatly into our taxonomy. When in doubt, we included them. Overall, we included a total of 99 articles and reports about software inspection in this survey.

Although we consider the selected sample of papers as representative of the work in the inspection area, we are aware that the published papers are only a biased sample of inspection work actually carried out in reality. There are two principal reasons for this, which we can only be aware of, without any hope of overcoming them:

1. The 'File drawer problem' – unpublished as well as unretrievable null results stored away by unknown researchers (Rosenthal, 1979). When inspections are unsuccessfully applied, they are most often not reported in the literature. In all the articles we reviewed, there is only one which shows that inspection did not have the expected benefits (Shirey, 1992). Yet, we believe that there might be more unsuccessful inspection trials.
2. The successful use of inspection might also be only sporadically reported since that may reveal defect information unpalatable to companies engaged in competitive industries (Ackerman et al., 1989).

3. Core inspection concepts and relationships

Based on the selected literature, we derived a taxonomy to articulate the core concepts and relationships of software inspection. This taxonomy is centered around five primary dimensions – technical, managerial, organizational, economics, and tools. With them, we attempted to characterize the nature of software inspection. For each primary dimension, we used a selection criterion in the form of a concrete goal to elicit from the relevant literature. Yet, though necessary, these five primary dimensions are not unique. They are relevant to the major areas of software development. Hence, we elicited from the literature particular sub-dimensions that we saw as fundamental to the nature and application of software inspection. Fig. 1 shows the elicited dimensions and sub-dimensions.

We briefly describe below each dimension and its associated primary goals. The major goal of the technical dimension is to characterize the different inspection methods so as to identify similarities and differences among them. For this, each inspection approach needs to be characterized in more detail according to the activities performed (process), the inspected software product (product), the different team roles as well as overall optimal size and selection (team roles, size, and selection), and the technique applied to detect defects in the software product (reading technique). The managerial dimension provides information on the effects inspections have on a project and vice versa. Managers are most often interested in the way inspections influence project effort (effort), project duration (duration), and product quality (quality). However, inspections might also have other effects a manager might be interested in, such as their contribution to team building or education in a particular project (others). The organizational dimension characterizes the effects inspections have on the whole organization and vice versa. For the organizational dimension, we elicited project structure (project structure), team (team), and environment (environment) as particular subdimensions. These subdimensions provide important information on the context in which inspections take place. The assessment dimension includes qualitative (qualitative assessment) and quantitative assessment (quantitative assessment) of inspections. This dimension allows one to make a comparison of cost/benefit ratios in a given situation to determine the economics fo a software inspection implementation. Finally, the tool dimension describes how inspections can be supported with tools. For this dimension, we elicited the purpose of the various tools (purpose) and investigated how they support a given inspection approach (supported inspection approach).

We have to state that the dimensions are not completely orthogonal, that is, one dimension

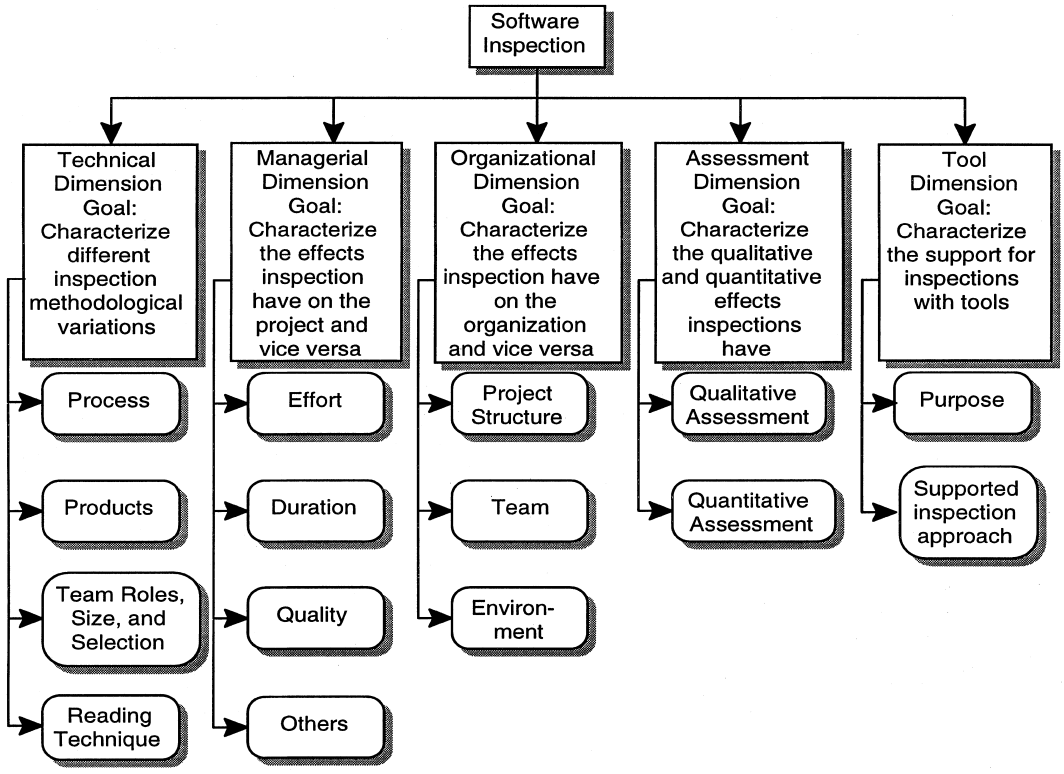

Fig. 1. Dimensions and subdimensions of the identified taxonomy.

may be related to another dimension, but this is unavoidable. For example, a manager might base his/her decision about introducing inspections on the cost/benefit ratio inspections had in previous projects. Yet, we have done our best to minimize such overlap.

We now proceed by discussing in details each of Fig. 1's dimensions and subdimensions using the relevant articles.

4. Inspection concepts and relationships

4.1. The technical dimension of software inspection

Inspections must be tailored to fit particular development situations. To do so, it is fundamental to characterize the technical dimension of current inspection methods and their refinements in order to grasp the similarities and differences among them. As depicted in Fig. 2, the technical dimension of our taxonomy includes as subdimensions the inspection process, inspected, the inspected product, the team roles participants have in an inspection as well as the team size, and the reading technique. Each of the subdimensions is discussed in more detail in this section. In total, we have identified 49 references relevant to this dimension.

4.1.1. The process dimension

To explain the various similarities and differences among the methods, a reference model for software inspection processes is needed. To define such a reference model, we adhered to the purpose of the various activities within an inspection rather than their organization. This allows us to provide an unbiased examination of the different approaches. We identified six major process phases: Planning, Overview, Defect Detection, Defect Collection, Defect Correction, and Follow-up. These phases can be found in most inspection methods or their refinements. However, the question of how each phase is organized and performed often makes the difference, that is, it characterizes one method for another.

4.1.1.1. Planning. The objective of the planning phase is to organize a particular inspection when materials to be inspected pass entry criteria, such as when source code successfully compiles without syntax errors. This phase includes the selection of inspection participants, their assignment to roles, the scheduling of the inspection meeting, and the distribution of the inspection material. In most papers, this phase is not described in much detail, except in Ackerman et al. (1989) and Fagan (1976). However, we consider planning important to mention as a separate phase because there must be a person within a project or organization who is

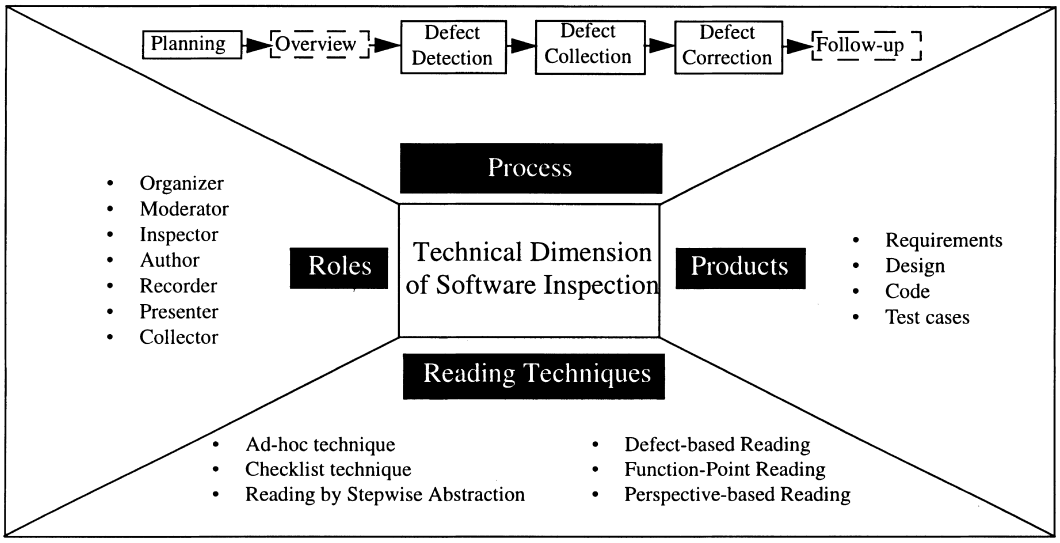

Fig. 2. Technical dimension of software inspection.

responsible for planning all inspection activities, even if such an individual plays numerous roles.

4.1.1.2. Overview. The overview phase consists of a first meeting in which the author explains the inspected product to other inspection participants. The main goal of the overview phase is to make the inspected product more lucid and, therefore, easier to understand and inspect for participants. Such a first meeting can be particularly valuable for the inspection of early artifacts, such as requirements or design documents, but also for complex source code. However, this meeting consumes effort and increases the duration of an inspection. Moreover, it may focus the attention of inspectors on particular issues, which prohibits an independent assessment of the inspected artifact. These limitations may be one reason why Fagan (1976) states that an overview meeting for code inspection is not necessary. This statement is somewhat supported by Gilb and Graham (1993). They call the overview meeting 'Kickoff Meeting' and point out that such a meeting can be held, if desired, but is not compulsory for every inspection cycle. However, other authors consider this phase essential for effectively performing the subsequent inspection phases. Ackerman et al. (1989), for example, argue that the overview brings all inspection participants to the point where they can easily read and analyse the inspected artifact. In fact, most published applications of inspections report performing an overview meeting (Crossman, 1991; Doolan, 1992; Fowler, 1986; Franz and Shih, 1994; Kelly et al., 1992; Kitchenham et al., 1986; Raz and Yaung, 1997; Reeve, 1991; Russell, 1991; Svendsen, 1992; Tripp et al., 1991; Wenneson, 1985). However, there are also examples that either did not perform one or did not report about one (Bourgeois, 1996; Knight and Myers, 1993).

We found two conditions under which an overview meeting is justified and beneficial. First, when the inspected artifact is complex and difficult to understand. In this case, explanations from the author about the inspected artifact facilitate the understanding of the inspected product for inspection participants. Second, when the inspected artifact belongs to a large software system. In this case, the author may explain the relationship between the inspected artifact and the whole software system to other participants. In both cases, explanations by the author may help other participants perform more effective inspection and save time in later inspection phases.

4.1.1.3. Defect detection. The defect detection phase can be considered the core of an inspection. The main goal of the defect detection phase is to scrutinize a software artifact to elicit defects. How to organize this phase is still debated in the literature. More specifically, the issue is whether defect detection is more an individual activity and hence should be performed individually, or whether defect detection is a group activity and should therefore be conducted as part of a group meeting, that is, an inspection meeting. Fagan (1976) reports that a group meeting provides a synergy effect, that is, most of the defects are detected because inspection participants meet and scrutinize the inspection artifact together. He makes the implicit assumption that interaction contributes something to an inspection that is more than the mere combination of individual results. Fagan refers to this effect as the 'phantom' inspector. However, others found little synergy in an inspection meeting. The most cited reference for this position is a paper by Votta (1993). His position is empirically supported in Sauer et al. (1996).

In the literature, we found a broad spectrum of opinions ranging between these two positions. In fact,

most of the papers consider defect detection as an individual or a mixture of individual and group activity rather than a pure group activity (Using our survey selection, we had the following tally: Individual: 19; Both: 15; Group: 13). In many cases, authors distinguish between a 'preparation' phase of an inspection, which is performed individually, and a 'meeting' phase of an inspection, which is performed within a group (Ackerman et al., 1989; Gilb and Graham, 1993; Strauss and Ebenau, 1993; Fagan, 1976). However, it often remains unclear whether the preparation phase is performed with the goal of detecting defects or just understanding the inspected artifact to detect defects later on in a meeting phase. For example, Ackerman et al. (1989) state that a preparation phase lets the inspectors thoroughly understand the inspected artifact. They do not explicitly state that the goal of the preparation phase is defect detection. Bisant and Lyle (1989) consider individual preparation as the vehicle for individual education. Other examples mention that the inspected artifact should be individually studied in detail throughout a preparation phase, but do not explicitly state education as a goal per se (Christenson et al., 1990; Doolan, 1992; Fowler, 1986; Letovsky et al., 1987).

Since the literature on software inspection does not provide a definite answer on which alternative to choose, we looked at some literature from the psychology of small group behavior (Dennis and Valacich, 1993; Levine and Moreland, 1990; Shaw, 1976). Psychologists found that an answer to the question whether individuals or groups are more effective, depends upon the past experience of the persons involved, the kind of task they are attempting to complete, the process that is being investigated, and the measure of effectiveness. Since at least some of these parameters vary in the context of a software inspection, we recommend organizing the defect detection activity as both individual and group activity with a strong emphasis on the former. Individual defect detection with the explicit goal of looking for defects that should be resolved before the document is approved ensures that inspectors are well-prepared for all following inspection steps. This may require extra effort on the inspectors' behalf since each of them has to understand and scrutinize the inspected document on an individual basis. However, the effort is justified because, if a group meeting is performed later on, each inspector can play an active role rather than hiding himself or herself in the group and, thus, make a significant contribution to the overall success of an inspection.

There has been noticeable growth in the research on how individual defect detection takes place and can be supported with adequate techniques (Basili et al., 1996; Basili, 1997; Porter et al., 1995b). We tackle this issue later in more detail when we discuss reading techniques to support defect detection.

4.1.1.4. Defect collection. In most published inspection processes, more than one person participates in an inspection and scrutinizes a software artifact for defects. Hence, the defects detected by each inspection participant must be collected and documented. Furthermore, a decision must be made whether a defect is really a defect. These are the main objectives of the defect collection phase. A follow-up objective may be to decide whether the inspected artifact needs to be reinspected. Since the defect collection phase is most often performed in a group meeting, the decision whether or not a defect is really a defect is in many cases a group decision. The same holds for the decision whether or not to perform a reinspection. To make the reinspection decision more objective, some authors suggest trying to apply statistical models for estimating the remaining number of defects in the software product after inspection (Eick et al., 1992; Wiel and Votta, 1993). If the estimate exceeds a certain threshold, the software product needs to be reinspected. However, in a recent study Briand et al. (1997) showed that statistical estimators are not very accurate for inspections with less than four inspection participants. Further research is necessary to validate this finding. In addition to the statistical estimation models, graphical defect content estimation approaches are currently being investigated (Wohlin and Runeson, 1998).

Since a group meeting is effort consuming and increases the development schedule, some authors suggest abandoning such a meeting for inspections. Instead, they offer the following alternatives (Votta, 1993): Managed meetings, depositions, and correspondence. Managed meetings are well-structured meetings with a limited number of participants. A deposition is a three-person meeting in which the author, a moderator, and an inspector collect the inspectors' findings and comments. Correspondence includes forms of communication where the inspections and author never actually meet (e.g., by using electronic mail). Some researchers have elaborated on these alternatives. Sauer et al. (1996), for example, provide some theoretical underpinning for depositions. They suggest that the most experienced inspectors collect the defects and decide upon whether these are real or not.

Overall, research does not seem to provide a conclusive answer to the question of whether inspection meetings pay-off or not. We recommend to practitioners that they start with the 'traditional' meeting-based approach and try later on whether non-meeting based approaches provide equivalent benefits. Regarding the benefits of group meetings, those also provide more intangible benefits such as dissemination of product information, development experiences, or enhancement of team spirit as reported in Franz and Shih (1994). Although difficult to measure, these benefits must be taken into account when a particular inspection approach is

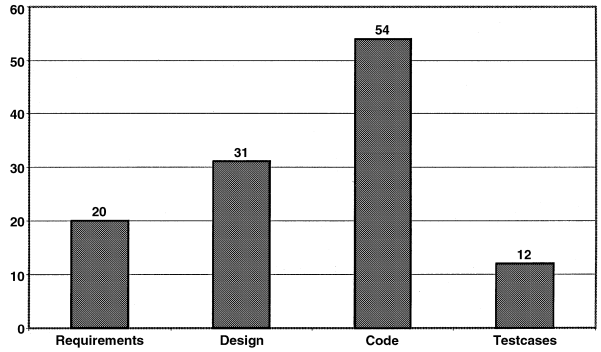

Fig. 3. Distribution of the use of software inspection on various product types.

evaluated in addition to the number of defects it helps detect and remove.

On the other hand, these meetings are not problem-solving sessions. Neither personal conflicts among people or departments nor radically alternate solutions – complete rewrite or redesign – of the inspected artifact should be discussed there.

4.1.1.5. Defect correction. Throughout the defect correction phase, the author reworks and resolves defects found (Fagan, 1976) or rationalizes their existence (Shirey, 1992). For this he or she edits the material and deals with each reported defect. There is not much discussion in the literature about this activity.

4.1.1.6. Follow-up. The objective of the follow-up phase is to check whether the author has resolved all defects. For this, one of the inspection participants verifies the defect resolution. Doolan reports that the moderator checks that the author has taken some remedial action for each defect detected (Doolan, 1992). However, others do not report a follow-up phase (Myers, 1978; Russell, 1991; Shirey, 1992). They either did not perform one or did not consider it important. Furthermore, many consider the follow-up phase optional like the overview phase.

4.1.2. The product dimension

The product dimension refers to the type of product that is usually inspected. Boehm (1981) stated that one of the most prevalent and costly mistakes made in software projects today is deferring the activity of detecting and correcting software problems until late in the project. This statement supports the use of software inspection for early life-cycle documents. However, a look at the literature reveals that in most cases inspection was applied to code documents. Fig. 3 depicts how inspection was applied to various software products [3] phasewise. Although code inspection improves the quality and provides savings, the savings are higher for early life-cycle artifacts as shown in a recent study (Briand et al., 1998), which integrates published inspection results into a coherent cost/benefit-model. The results of the study reveal that the introduction of code inspection saves 39% of defect costs compared to testing alone. The introduction of design inspection saves 44% of defect costs compared to testing alone.

4.1.3. The team role and size

Practitioners usually have the following three questions about software inspections: (1) what roles are involved in an inspection, (2) how many people are assigned to each role, and (3) how to select people for each role. For the first question, a number of specific roles are assigned to inspection participants. Hence, each inspection participant has a clear and specific responsibility. The roles and their responsibilities are described in Ackerman et al. (1989), Fagan (1976) and Russell (1991). There is not much disagreement regarding the definition of inspection roles. In the following, we describe each of these roles in more detail:

- *Organizer*: The organizer plans all inspection activities within a project or even across projects.
- *Moderator*: The moderator ensures that inspection procedures are followed and that team members perform their responsibilities for each phase. He or she moderates the inspection meeting if there is one. In this case, the moderator is the key person in a successful inspection as he or she manages the inspection team and must offer leadership. Special training for this role is suggested.
- *Inspector*: Inspectors are responsible for detecting defects in the target software product. Usually all team members can be assumed to be inspectors, regardless of their specific role.
- *Reader/Presenter*: If an inspection meeting is performed, the reader will lead the team through the material in a complete and logical fashion. The material should be paraphrased at a suitable rate for detailed examination. Paraphrasing means that the reader should explain and interpret the material rather than reading it literally.
- *Author*: The author has developed the inspected product and is responsible for the correction of defects during rework. During an inspection meeting, he or she addresses specific questions the reader is not able to answer. The author must not serve as moderator, reader, or recorder.
- *Recorder*: The recorder is responsible for logging all defects in an inspection defect list during the inspection meeting.
- *Collector*: The collector consolidates the defects found by the inspectors if there is no inspection meeting.

[3] We should note that some articles describe software inspection for several products. This explains why the total number of references is 114, although we only included 99 articles in this survey.

To answer the second question, that is, how to assign resources to these roles in an optimal manner, the reported numbers in the literature are not uniform. Fagan (1976) recommends to keep the inspection team small, that is, four people. Bisant and Lyle (1989) have found performance advantages in an experiment with two persons: one inspector and the author, who can also be regarded as an inspector. Weller (1993) presents some data from a field study using three to four inspectors. Madachy et al. (1993) presents data showing that the optimal size is between three and five people. Bourgeois (1996) corroborates these results in a different study. The experimental results of Porter et al. (1997) suggest that reducing the number of inspectors from 4 to 2 may significantly reduce effort without increasing inspection interval or reducing effectiveness.

We assume that there is no definite answer to this question and that an answer heavily depends on the type of product and the environment in which an inspection is performed. However, we recommend starting with three to four people: One author, one or two inspectors, and one moderator (also playing the role of the presenter and scribe). After a few inspections, the benefits of adding an additional inspector can be empirically evaluated.

The final question is how to select members of an inspection team. Primary candidates for the role of inspectors are personnel involved in product development (Fagan, 1986). Outside inspectors may be brought in when they have a particular expertise that would add to the inspection (National Aeronautics and Space Administration, 1993). Inspectors should have good experience and knowledge (Fagan, 1986; Blakely and Boles, 1991; Strauss and Ebenau, 1993). However, the selection of inspectors according to experience and knowledge has two major implications. First, inspection results heavily depend upon human factors. This often limits the pool of relevant inspectors to a few developers working on a similar or interfacing products (Ackerman et al., 1989). Second, personnel with little experience are not chosen as inspectors although they may learn and, thus, profit a lot from inspection. Defect detection, that is, reading techniques, which we discuss later on in more detail, may alleviate these problems.

It is sometimes recommended that managers should neither participate nor attend inspections (National Aeronautics and Space Administration, 1993; Kelly et al., 1992). This stems from the fact that inspections should be used to assess the quality of the software product, not the quality of the people who create the product (Fagan, 1986). Using inspection results to evaluate people may result in less than honest and thorough inspections results since inspectors may be reluctant to identify defects if finding them will result in a poor performance evaluation for a colleague.

4.1.4. The reading technique dimension

Recent empirical studies seem to demonstrate that defect detection is more an individual than a group activity as assumed by many inspection methods and refinements (Land et al., 1997; Porter and Johnson, 1997; Votta, 1993). Moreover, these empirical studies show that a particular organization of the inspection process does not explain most of the variation in inspection results. Rather, one expects that inspection results depend on inspection participants themselves (Porter and Votta, 1997) and their strategies for understanding the inspected artifacts (Rifkin and Deimel, 1994). Therefore, supporting inspection participants, that is, inspectors, with particular techniques that help them detect defects in software products, may increase the effectiveness of an inspection team most. We refer to such techniques as *reading techniques*.

A reading technique can be defined as a series of steps or procedures whose purpose is for an inspector to acquire a deep understanding of the inspected software product. The comprehension of inspected software products is a prerequisite for detecting subtle and/or complex defects, those often causing the most problems if detected in later life-cycle phases. In a sense, a reading technique can be regarded as a mechanism for the individual inspector to detect defects in the inspected product. Of course, whether inspectors take advantage of this mechanism is up to them.

Even though reading is one of the key activities for individual defect detection (Basili, 1997), few documented reading techniques are currently available to support the activity. We found that *ad hoc reading* and *checklist-based reading* are probably the most popular reading techniques used today for defect detection in inspections (Fagan, 1976; Gilb and Graham, 1993).

Ad hoc reading, by nature, offers very little reading support at all since a software product is simply given to inspectors without any direction or guidelines on how to proceed through it and what to look for. However, ad-hoc does not mean that inspection participants do not scrutinize the inspected product systematically. The word 'ad-hoc' only refers to the fact that no technical support is given to them for the problem of how to detect defects in a software artifact. In this case, defect detection fully depends on the skill, the knowledge, and the experience of an inspector. Training sessions in program comprehension as presented in Rifkin and Deimel (1994) may help subjects develop some of these capabilities to alleviate the lack of reading support. Although an ad-hoc reading approach was only mentioned a few times in the literature (Shirey, 1992; Doolan, 1992), we found many articles in which little was mentioned about how an inspector should proceed in order to detect defects. Hence, we assumed that in most of these cases no particular reading technique was provided because otherwise it would have been stated.

Checklists offer stronger, boilerplate support in the form of questions that inspectors are to answer while reading the document. Checklists are advocated in more than twenty five articles. See, for example, Ackerman et al. (1989), Fagan (1976), Fagan (1986), Humphrey (1995) and Tervonen (1996). Gilb and Graham (1993) also advocate the use of checklists must ultimately be derived from the rules of the process which themselves are being checked by inspection. Although reading support in the form of a list of questions is better than none (such as ad-hoc), checklist-based reading has several weaknesses as denoted in the literature. First, the questions are often general and not sufficiently tailored to a particular development environment. Thus, the checklist provides little support to help an inspector understand the inspected artifact. That can be vital to detect application logic defects. Second, concrete instructions on how to use a checklist are often missing, that is, it is often unclear when and based on what information an inspector is to answer a particular checklist question. Finally, the questions of a checklist are often limited to the detection of defects that belong to particular defect types. Since the defect types are based on past defect information (Chernak, 1996), inspectors may not focus on defect types not previously detected and, therefore, may miss whole classes of defects.

Techniques providing more structured and precise reading instructions include both a reading technique denoted as '*Reading by Stepwise Abstraction*' for code documents advocated by the Cleanroom community (Dyer, 1992a,b; Linger et al., 1979), as well as a technique suggested by Parnas et. al. called *Active Design Review* (Parnas and Weiss, 1985; Parnas, 1987) for the inspection of design documents. Reading by Stepwise Abstraction requires an inspector to read a sequence of statements in the code and to abstract the function these statements compute. An inspector repeats this procedure until the final function of the inspected code artifact has been abstracted and can be compared with the specification. Active Design Reviews, which is a suggested variation to the conventional inspection methodology, assign clear responsibilities to inspectors of a team and require each of them to take an active role in an inspection of design artifacts. In doing so, an inspector is required to make assertions about parts of the design artifact rather than simply point out defects.

A more recent development in the area of reading techniques for individual defect detection in software inspection is *Scenario-based reading* (Basili, 1997). The gist of the Scenario-based reading idea is the use of the notion of scenarios that provide custom guidance for inspectors on how to detect defects. A scenario may be a set of questions or a more detailed description for an inspector on how to perform the document review. Principally, a scenario limits the attention of an inspector to the detection of particular defects as defined by the custom guidance. Since each inspector may use a different scenario, and each scenario focuses on different defect types, it is expected that the inspection team, together, becomes more effective. Hence, it is clear that the effectiveness of a scenario-based reading technique depends on the content and design of the scenarios. So far, researchers have suggested three different approaches for developing scenarios and, therefore, three different scenario-based reading techniques: *Defect-based Reading* (Porter et al., 1995b) for inspecting requirements documents, a scenario-based reading technique based on function points for inspecting requirements documents (Cheng and Jeffrey, 1996), and *Perspective-based Reading* for inspecting requirements documents (Basili et al., 1996) or code documents (Laitenberger and DeBaud, 1997).

The main idea behind Defect-based Reading is for different inspectors to focus on different defect classes while scrutinizing a requirements documents (Porter et al., 1995b). For each defect class, there is a scenario consisting of a set of questions an inspector has to answer while reading. Answering the questions helps an inspector primarily detect defects of that particular class. The defect-based reading technique has been validated in a controlled experiment with students as subjects. The major finding was that inspectors applying Defect-based Reading are detecting more defects than inspectors applying either Ad-hoc or checklist-based reading.

Cheng and Jeffrey (1996) have chosen a slightly different approach to define scenarios for defect detection in requirements documents. This approach is based on Function Point Analysis (FPA). FPA defines a software system in terms of its inputs, files, inquiries, and outputs. The scenarios, that is, the Function Point Scenarios, are developed around these items. A Function Point Scenario consists of questions and directs the focus of an inspector to a specific function-point item within the inspected requirements document. The researchers carried out an experiment to investigate the effectiveness of this approach compared to an Ad-hoc approach. The experimental results show that, on average, inspectors following the Ad-hoc approach found more defects than inspectors following the function-point scenarios. However, it seems that experience is a confounding factor that biased the results of the experiment.

The main idea behind the perspective-based reading technique is that a software product should be inspected from the perspective of different stakeholders (Basili et al., 1996; Laitenberger and DeBaud, 1997). The rationale is that there is no single monolithic definition of software quality, and little general agreement about how to define any of the key quality properties, such as correctness, maintainability, or testability. Therefore, inspectors of an inspection team have to check software quality as well as the software quality factors of a

software artifact from different perspectives. The perspectives mainly depend upon the roles people have within the software development or maintenance process. For each perspective, either one or multiple scenarios are defined, consisting of repeatable activities an inspector has to perform, and questions an inspector has to answer. The activities are typical for the role within the software development or maintenance process, and help an inspector increase his or her understanding of the software product from the particular perspective. For example, designing test cases is a typical activity performed by a tester. Therefore, an inspector reading from the perspective of a tester may have to think about designing test cases to gain an understanding of the software product from the tester's point of view. Once understanding is achieved, questions about an activity or questions about the result of an activity can help an inspector identify defects.

Reading a document from different perspectives is not a completely new idea. It was seeded in early articles on software inspection, but never worked out in detail. Fagan (1976) reports that a piece of code should be inspected by the real tester. Fowler (1986) suggests that each inspection participant should take a particular point of view when examining the work product. Graden et al. (1986) state that each inspector must denote the perspective (customer, requirements, design, test, maintenance) by which they have evaluated the deliverable. So far, the perspective-based reading technique has been applied for inspecting requirements (Basili et al., 1996) and code documents (Laitenberger and DeBaud, 1997).

General prescriptions about which reading technique to use in which circumstances can rarely be given. However, in order to compare them, we set up the following criteria: Application Context, Usability, Repeatability, Adaptability, Coverage, and Overlap. The criteria are to provide answers to the following questions:

1. *Application context*: To which software products can a reading technique be applied and to which software products has a reading technique already been applied?
2. *Usability*: Does a reading technique provide prescriptive guidelines on how to scrutinize a software product for defects?
3. *Repeatability*: Are the results of an inspector's work repeatable, that is, are the results such as the detected defects, independent of the person looking for defects?
4. *Adaptability*: Is a reading technique adaptable to particular aspects, e.g., notation of the document, or typical defect profiles in an environment?
5. *Coverage*: Are all required quality properties of the software product, such as correctness or completeness, verified in an inspection?
6. *Overlap*: Does the reading technique focus each inspector to check the same quality properties or do different inspectors check different quality properties?
7. *Validation*: How was the reading technique validated, that is, how broadly has it been applied so far?

Table 2 characterizes each reading technique according to these criteria. We use question marks for cases for which no clear answer can be provided.

4.2. The managerial dimension of software inspection

One of the most important criteria for choosing a particular inspection approach is the effort a particular inspection method or refinement consumes. Effort is an issue project managers are mainly interested in. Hence, we refer to this dimension as the managerial dimension. To make a sound evaluation, that is, to determine whether it is worth spending effort on inspection, one must also consider how inspections affect the quality of the software product as well as the cost and the duration of the project in which they are applied. We discuss a sample of 24 articles in the context of these three subdimensions.

4.2.1. Quality

Some authors state that inspections can reduce the number of defects reaching testing by ten times (Freedman and Weinberg, 1990). However, these statements are often based on personal opinion rather than on collected inspection data. Hence, we focus our discussion about quality on examples of published inspection data taken from the literature. We emphasize that many of the data reported in the literature are not presented in a manner that allows straightforward comparison and analysis as pointed out by Briand et al. (1998).

Fagan (1976) presents data from a development project at Aetna Life and Casualty. An application program of eight modules (4439 non-commentary source statements) was written in Cobol by two programmers. Design and code inspections were introduced into the development process. After 6 months of actual usage, 46 defects had been detected during development and usage of the program. Fagan reports that 38 defects had been detected by design and code inspections together, yielding a defect detection effectiveness for inspections of 82%. In this case, the defect detection effectiveness was defined as the ratio of defects found and the total number of defects in the inspected software product. The remaining 8 defects had been found during unit test and preparation for acceptance test. In another article, Fagan (1986) publishes data from a project at IBM Respond, UK. A program of 6271 LOC in PL/1 was developed by seven programmers. Over the life cycle of the product, 93% of all defects were detected by inspections. He also mentions

Table 2
Characterization of reading techniques

Reading techniques	Characteristic						
	Application context	Usability	Repeatability	Adaptability	Coverage	Overlap	Validation
Ad-hoc	All products; all products	No	No	No	Low	High	Industrial practice
Checklists	All products; all products	No	No	Yes	Case dependent	High	Industrial practice
Reading by stepwise abstraction	All products allowing abstraction; functional code	Yes	Yes	No	High	High	Applied in clean room projects (Linger et al., 1979)
Active design reviews	Design; design	Yes	Yes	Yes	?	?	Experimental validation (Parnas, 1987)
Defect-based reading	All products; requirements	Yes	Case dependent	Yes	High	?	Experimental validation (Porter et al., 1995b)
Reading based on function points	All products; requirements	Yes	Case dependent	Yes	?	?	Experimental validation (Cheng and Jeffrey, 1996)
Perspective-based reading	All products; requirements, code	Yes	Yes	Yes	High	?	Experimental validation (Basili et al., 1996; Laitenberger and DeBaud, 1997)

two projects of the Standard Bank of South Africa (143 KLOC) and American Express (13 KLOC of system code), each with a defect detection effectiveness for inspections of over 50% without using trained inspection moderators.

Weller (1992) presents data from a project at Bull HN Information Systems which replaced inefficient C-code for a control microprocessor with Forth. After system tests had been completed, code inspection effectiveness was around 70%. Grady and van Slack (1994) report on experiences from achieving widespread inspection use at HP. In one of the company's divisions, inspections (focusing on code) typically found 60–70% of the defects. Shirey (1992) states that defect detection effectiveness of inspections is typically reported to range from 60% to 70%. Barnard and Price (1994) cite several references and report a defect detection effectiveness for code inspections varying from 30% to 75%. In their environment at AT&T Bell Laboratories, the authors achieved a defect detection effectiveness for code inspections of more than 70%. McGibbon (1996) presents data from Cardiac Pacemakers, where inspections are used to improve the quality of life-critical software. They observed that inspections removed 70–90% of all faults detected during development. Collofello and Woodfield (1989) evaluated reliability-assurance techniques in a case study – a large real-time software project that consisted of about 700 000 lines of code developed by over 400 developers. The respective defect detection effectiveness is reported to be 54% for design inspections, 64% for code inspections, and 38% for testing. More recently, Raz and Yaung (1997) presented the results of an analysis of defect-escape data from design inspection in two maintenance releases of a large software product. They found that the less effective inspections were those with the largest time investment, the likelihood of defect escapes being clearly affected by the way in which the time was invested and by the size of the work product inspected. Kitchenham et al. (1986) report on experience at ICL, where 57.7% of defects were found by software inspections. The total proportion of development effort devoted to inspections was only 6%. Gilb and Graham (1993) include experience data from various sources in their discussion of the benefits and costs of inspections. IBM Rochester Labs publish values of 60% for source code inspections, 80% for inspections of pseudocode, and 88% for inspections of module and interface specifications. Grady (1994) performs a cost/benefit analysis for different techniques, among them design and code inspections. He states that the average percentage of defects found for design inspections is 55%, and 60% for code inspections. Franz and Shih (1994) present data from code inspection of a sales and inventory tracking systems project at HP. This was a batch system written in COBOL. Their data indicate that inspections had 19% effectiveness for defects that could also be found

during testing. Myers (1978) performed an experiment to compare program testing to code walkthroughs and inspections. This research is based on work performed earlier by Hetzel (1976). The subjects were 59 highly experienced data processing professionals testing and inspecting a PL/I program. Myers reports an average effectiveness value of 38% for inspections. This controlled experiment was replicated several times (Basili and Selby, 1987; Kamsties and Lott, 1995; Myers, 1978; Wood et al., 1997) with similar results.

4.2.2. Cost

It is necessary for a project manager to have a precise understanding of the cost associated with inspections. Since inspection is a human-based activity, inspection costs are determined by human effort. The most important question addressed in literature is whether an inspection effort is worth making when compared to the effort for other defect detection activities, such as testing. Most of the literature present solid data supporting the claim that the costs for detecting and removing defects during inspections is much lower than detecting and removing the same defects in later phases. For instance, the Jet Propulsion Laboratory (JPL) found the ratio of the cost of fixing defects during inspections to fixing them during formal testing ranged from 1:10 to 1:34 (Kelly et al., 1992), at the IBM Santa Teresa Lab the ratio was 1:20 (Remus, 1984), and at the IBM Rochester Lab it was 1:13 (Kan, 1995).

We must say that authors often relate the costs to either the size of the inspected product or the number of defects found. Ackerman et al. (1989) present data on different projects as a sample of values from the literature and from private reports:

- The development group for a small warehouse-inventory system used inspections on detailed design and code. For detailed design, they reported 3.6 h of individual preparation per thousand lines, 3.6 h of meeting time per thousand lines, 1.0 h per defect found, and 4.8 h per major defect found (major defects are those that will affect execution). For source code, the results were 7.9 h of preparation per thousand lines, 4.4 h of meetings per thousand lines, and 1.2 h per defect found.
- A major government-systems developer reported the following results from inspection of more than 562 000 lines of detailed design and 249 000 lines of source code: For detailed design, 5.76 h of individual preparation per thousand lines, 4.54 h of meetings per thousand lines, and 0.58 h per defect found. For code, 4.91 h of individual preparation per thousand lines, 3.32 h of meetings per thousand lines, and 0.67 h per defect found.
- Two quality engineers from a major government-systems contractor reported 3–5 staff-hours per major defect detected by inspections, showing a surprising consistency over different applications and programming languages.
- A banking computer-services firm found that it took 4.5 h to eliminate a defect by unit testing compared to 2.2 h by inspection (these were probably source code inspections).
- An operating-system development organization for a large mainframe manufacturer reported that the average effort involved in finding a design defect by inspections is 1.4 staff-hours compared to 8.5 staff-hours of effort to find a defect by testing.

Weller (1993) reports data from a project that performed a conversion of C-code to Fortran for several timing-critical routines. While testing the rewritten code, it took 6 h per failure. It was known from a pilot project in the organization that they had been finding defects in inspections at a cost of 1.43 h per defect. Thus, the team stopped testing and inspected the rewritten code, detecting defects at a cost of less than 1 h per defect.

Collofello and Woodfield (1989) estimate some factors for which they had insufficient data. They performed a survey among many of the 400 members of a large real-time software project who were asked to estimate the effort needed to detect and correct a defect for different techniques. The results were 7.5 h for a design error, 6.3 h for a code error, both detected by inspections, 11.6 h for an error found during testing, and 13.5 h for an error discovered in the field.

Franz and Shih (1994) data indicate that the average effort per defect for code inspections was 1 h and for testing 6 h. In presenting the results of analyzing inspections data at JPL, Kelly et al. (1992) report that it takes up to 17 h to fix defects during formal testing, based on a project at JPL. They also report approximately 1.75 h to find and fix defects during design inspections, and approximately 1.46 h during code inspections.

There are also examples that present findings from applying inspections only as a quality assurance activity. Kitchenham et al. (1986), for instance, report on experience at ICL where the cost of finding a defect in design inspections was 1.58 h.

Gilb and Graham (1993) include experience data from various sources in their discussion of the benefits and costs of inspections. A senior software engineer describes how software inspections started at Applicon. In the first year, 9 code inspections and 39 document inspections (documents other than code) were conducted and an average effort of 0.8 h was spent to find and fix a major problem. After the second year, a total of 63 code inspections and 100 document inspections had been conducted and the average effort to find and fix a major problem was 0.9 h.

Bourgeois (1996) reports experience from a large maintenance program within Lock-heed Martin Western Development Labs where software inspections

replaced structured walkthroughs in a number of projects. The analysed program was staffed by more than 75 engineers who maintain and enhance over 2 million lines of code. The average effort for 23 software inspections (6 participants) was 1.3 staff-hours per defect found and 2.7 staff-hours per defect found and fixed. Bourgeois also presents data from Jet Propulsion Laboratory which is used as an industry standard. There, the average effort for 171 software inspections (5 inspection participants) was 1.1 staff-hours per defect found and 1.4–1.8 staff-hours per defect found and fixed.

Because inspection is a human-intensive activity and, therefore, effort consuming, managers are often critical or even reluctant to use them the first time. Part of the problem is the perception that software inspections cost more than they are worth. However, available quantitative evidence as presented above indicates that inspections have had significant positive impact on the quality of the developed software and that inspections are more cost-effective than other defect detection activities, such as testing. Furthermore, it is important to keep in mind that besides quality improvement and cost savings realized by finding and fixing defects before they reach the customer, other benefits are often associated with performing inspections. These benefits, such as learning, are often difficult to measure, but they also have an impact on quality, productivity, and the success of a software development project.

4.2.3. Duration

Inspections do not only consume effort, but they also have an impact on the product's development cycle time. Inspection activities are scheduled in a way in which all people involved can participate and fulfill their roles. Thus, the interval for the completion of all activities will range from at least a few days up to a few weeks. During this period, other work that relies on the inspected software product may be delayed. Hence, duration might be a crucial aspect for a project manager if time to market is a critical issue during development. However, only few articles present information on the global inspection duration.

Votta discusses the effects of time loss due to scheduling contention. He reports that inspection meetings account for 10% of the development interval (Votta, 1993). Due to the delays, he advises substituting inspection meetings with other forms of defect collection.

4.3. The organizational dimension of software inspection

Fowler (1986) states that the introduction of inspection is more than giving individuals the set of skills on how to perform inspections: It also introduces a new process within an organization. Hence, it affects the whole organization, that is, the team, the project structure, and the environment. We identified six references relevant to this dimension.

4.3.1. Team

An important factor regarding software inspection is the human factor. Software inspection is driven by its participants, i.e, the members of a project team. Hence, the success or failure of software inspection as a tool for quality improvement and cost reduction heavily depends on human factors. If team members are unwilling to perform inspections, all efforts will be deemed to fail. Franz and Shih (1994) point out that attitude about defects is the key to effective inspections. Once the inevitability of defects is accepted, team members often welcome inspections as a defect detection method. To overcome objections, Russell reports on an advertising campaign to persuade project teams that inspections really do work (Russell, 1991). An advice which we often found in the literature was to exclude management from inspections (Franz and Shih, 1994; Kelly et al., 1992). This is suggested to avoid any misconception that inspection results are used for personnel evaluation. Furthermore, training is deemed essential (Ackerman et al., 1989; Fowler, 1986). Training allows project members to build their own opinion on how inspections work and how crucial defect data are within an environment for triggering further empirically justified process improvements.

4.3.2. Project structure

Inspection per se is a human-based activity. Especially when meetings are performed, authors are confronted with the defects they created. This can easily result in personal conflicts, particularly in project environments with a strict hierarchy. Hence, one must consider the project structure to anticipate the conflict potential among participants. Depending on this potential for conflict, one must decide whether an inspection moderator belongs to the development team or must come from an independent department. This is vital in cases in which inspection is applied between subgroups of one project. Personal conflicts within an inspection result in demotivation for performing inspection at all.

4.3.3. Environment

Introducing inspections is a technology transfer initiative. Hence, issues revolve around the need to deal with a software development organization, not just in terms of its workers but also in terms of its culture, management, budget, quality, and productivity goals. All these aspects can be subsumed in the subdimension environment of an organization. Fowler (1986) states that preparing the organization for using inspections dovetails with adapting the inspections to the local technical issues. Furthermore, the new process must be

carefully designed to serve in the organization's environment and culture. Based on their inspection experiences at Hewlett-Packard, Grady and van Slack (1994) suggest a four-stage process for inspection technology transfer: Experimental stage, initial guideline stage, widespread belief and adoption stage, and standardization stage. The experimental stage comprises the first inspection activities within an organization, and is often limited to a particular project of an organization. Based on the experiences in this project, first guidelines can be developed. This is the starting point for the initial guideline stage. In this stage, the inspection approach is defined in more detail and training material is created. The widespread belief and adoption stage takes advantage of the available experiences and training material to adopt inspection to several projects. Finally, the standardization stage helps build an infrastructure structure strong enough to achieve and hold inspection competence. This approach follows a typical new technology transfer model.

4.4. The assessment dimension of software inspection

When assessing whether inspections provide any benefits, we differentiate between qualitative versus quantitative assessment. While qualitative assessment is often based on the subjective opinion of inspection participants, quantitative assessment is based on data collected in inspection and subsequent defect detection activities, such as testing. In contrast to the managerial dimension, the assessment describes how to evaluate inspections rather than the results of the evaluation. We identified 20 references relevant to this dimension.

4.4.1. Qualitative assessment

Qualitative assessment is based on subjective judgement of inspection benefits rather than on real inspection data. Weller states inspection participation results in a better understanding of the software development process (Weller, 1993) and the developed product. This is supported in Doolan (1992). Furthermore, inspections contribute to increased team work because they allow the team to see each other's strengths and weaknesses (Crossman, 1991; Doolan, 1992; Franz and Shih, 1994; Jackson and Hoffman, 1994; MacLeod, 1993). They also provide a good forum for learning, that is, educate the team (Bisant and Lyle, 1989; Crossman, 1991; Doolan, 1992; Franz and Shih, 1994; Jackson and Hoffman, 1994; MacLeod, 1993; Tripp et al., 1991). One explanation is that team members become familiar with the whole system, not just with the part on which each one of them is working. Finally, inspections contribute to social integration (Svendsen, 1992). Apart from the effects on the development team, inspections affect each participant individually. One observation is that inspection participants develop software products more carefully (Doolan, 1992; Fagan, 1986; Tripp et al., 1991; Weller, 1993). Most of these advantages are systematically used within the Personal Software Process, advocated by Humphrey (1995).

4.4.2. Quantitative assessment

Quantitative assessment is often based on the data collected in an inspection or in the project in which inspections were applied. It requires an evaluation model (Briand et al., 1996) that describes how to combine the collected data in order to come up with a valid conclusion. Various models have been described in the literature. One can discern between models that do not consider costs and models that do consider costs. Models that do not consider the cost for performing inspections are presented in Fagan (1976), Jones (1996), Remus (1984), Collofello and Woodfield (1989) and Raz and Yaung (1997). These models basically relate the number of defects found in inspection to the total number of defects in a software artifact (if available). Models that do consider cost are presented by Grady and van Slack (1994), Collofello and Woodfield (1989), Franz and Shih (1994) and Kusumoto (1993). Most of these models are built on the concept of comparing inspection as a defect detection technique against testing activities. However, various assumptions are made for the various models. Hence, before applying them, one must carefully check whether the assumptions are fulfilled. An overview of the different models as well as a more mathematical description of each can be found in Briand et al. (1998).

4.5. The tool dimension of software inspection

Currently, few tools supporting inspections are available. Some of them were developed by researchers to investigate software (often source code) inspection and none of the academic tools has reached commercial status yet. There may be some commercial tools available that we were not aware of, since they have not been discussed in the inspection literature. We analysed, discussed, and classified the following ten inspection tools: (1) PAE (Program Assurance Environment) (Belli and Crisan, 1996), which can be seen as an extended debugger and represents an exception in the list of tools. (2) InspecQ (Knight and Myers, 1993) concentrates on the support of the Phased Inspection process model developed by Knight and Meyers (3) ICILE (Brothers et al., 1990) supports the defect detection phase as well as the defect collection phase in a face-to-face meeting. (4) Scrutiny (Gintell et al., 1995) and (5) CSI (Mashayekhi et al., 1993), support synchronous, distributed meetings to enable the inspection process for geographically separated development teams. (6) CSRS (Johnson and Tjahjono, 1997), (7) InspectA (Knight and Myers, 1993), (8) Hypercode (Perpich et al., 1997), and

(9) ASIA (Perry et al., 1996) remove the conventional defect collection phase and replace it with a public discussion phase were participants vote on defect-annotations. (10) ASSIST (Macdonald and Miller, 1995) uses its own process modelling language and executes any desired inspection process model. All tools provide more or less comfortable document handling facilities for browsing documents on-line.

To compare the various tools, we developed Table 3 according to the various phases of the inspection process. We focused on whether a tool provides facilities to control and measure the inspection process, and on the infrastructure on which the tool is running (a cross 'x' indicates support and a minus '–' no support). Of course, for source code products various compilers are available that can perform type and syntactical checking. This may remove some burden from inspectors. Furthermore, support tools, such as Lint for C, may help detect further classes of defects. However, the use of these tools is limited to particular development situations and may only lighten the inspection burden.

5. A generic life-cycle model for software development

So far, we have defined and detailed the various characterization dimensions of software inspection. Yet, software inspection can take place at different levels within the development life-cycle of software products. As we saw earlier in this paper, an important inspection customization factor is the stage within the life cycle from which the software product is originating. Hence, we believe it is important to present the inspection body of knowledge from a life-cycle point of view. To do this, we first introduce a generic life-cycle model to serve as reference for this angle.

We used as our model a simplified version of the *Vorgehensmodell* (V-Model) (Bröhl and Dröschel, 1995) to discuss inspection variations according to the identified products. Fig. 4 presents its main products and the main relationships among them. The products are generic enough and are found, at least in some form, in most, if not all, development process models. Hence, we consider this model appropriate for our purpose. This is an important observation because it allows the results from this work to be applied to most software development environments. Of course, some tailoring and/or modification of products from the development life-cycle might be required to accommodate naming conventions and project organization issues.

The V-model is not a process model per se, but rather a product model since it does not define the sequence of development steps that must be followed to create the generic software development products. Hence, it is applicable with all process models for which products are developed in sequence, in parallel, or incrementally. The point is that the logical relationship between development products should be maintained.

The following generic software development products are defined as those for which inspections can be conducted:

1. *Problem description*: This document is created by the customer to describe the problem for which a solution is being sought. The description might not be restricted to the software aspects of the problem but might also address a broader system context beyond the software components of that system.
2. *Customer requirements*: This document is created essentially by the customer, though the requirements engineer may assist. The document recasts the problem description in terms of requirements that must be satisfied by a software solution and generally addresses more than just software requirements. The combination of the two documents is frequently used by the customer as a statement of work to potential vendors for bidding on a development project.
3. *Developer requirements*: This document is created by the requirements engineer and defines the requirements for the proposed software solution to the customer's requirements. Hence, it describes precisely *what* the software system should do. The document should address all customer requirements and also introduce requirements unique to the particular software solution. This document is the formal response to the customer's requirements and serves as the technical basis for a contractual arrangement between the customer and vendor. This document sometimes may evolve into a specification document.
4. *Architecture*: This document is created by the system architect and describes a system design to implement the developer requirements. Hence, it describes the issue of *how* the system is to be structured (with associated rationale) so as to provide a solution. The architecture generally identifies the component parts, software and otherwise, and how they fit together and interact to provide the customer-required capability.
5. *Design*: This document is created by the designer and describes how the different components should be designed to fit and realize the architecture specification. It generally identifies their interface, structure, and how they fit together and interact with other components to provide the customer-required capability.
6. *Implementation*: These documents are created by the component programmer and includes the software code and ancillary support documentation.
7. *Test Cases*: The unit, integration, system and acceptance test cases are generated in accordance with the previously prepared documents, by the unit tester, integrator, and system tester, respectively. These tests allow them to validate the specific behavior of the executable modules, the executable system, the usable system, and the used system against the architecture,

Table 3
Overview of inspection tools

	PAE	ICICLE	Scrutiny	CSRS	InspecQ	ASSIST	CSI/ CAIS	InspectA	Hyper-code	ASIA
Reference	Belli and Crisan (1996)	Brothers et al. (1990)	Gintell et al. (1995)	Johnson and Tjahjono (1993)	Knight and Myers (1993)	Macdonald (1997)	Mashayekhi et al. (1993)	Murphy and Miller (1997)	Perpich et al. (1997)	Stein et al. (1997)
Planning support	–	–	–	x	x	x	–	x	x	–
Defect detection support	x	x	x	x	x	x	x	x	x	x
Automated defect detection	x	x	–	–	x	–	–	–	–	–
Annotation support	–	x	x	x	x	x	x	x	x	x
Document handling support	C-Code	C-Code	Code	Code/Text	C-Code (Ada)	Code	Code	Code	Code/Text	Code/ Text/ Graph
Reading technique	Checklist	Checklist	–	Checklist	Checklist	–	Checklist	Checklist	–	–
Defect collection support	–	x	x	x	–	x	x	x	x	x
(Synch/Asynch)	–/–	S/–	S/–	–/A	–/–	S/A	S/A	–/A	S/A	–/A
(Local/Distributed)	–/–	L/–	L/D	–/D	–/–	L/D	L/D	–/D	L/D	–/D
Defect correction support	–	–	–	–	–	–	–	(x)	(x)	x
Inspection process control possible	–	–	–	x	x	x	–	x	x	x
(Active/Passive/ None)	N	N	N	A/P	A	A/P	A/P	A/P	A/P	P
Process measurement support	–	x	x	x	–	x	x	x	x	x
Timestamp (effort)	–	–	–	x	–	–	x	–	–	x
Defect Statistics	–	x	x	x	–	x	x	x	x	x
others	–	–	ISO	–	–	–	History	–	ISO	x
Supported infrastructure	UNIX	X-win	ConvB	Ergret	?	LAN	Suite	E-mail	Web	Web

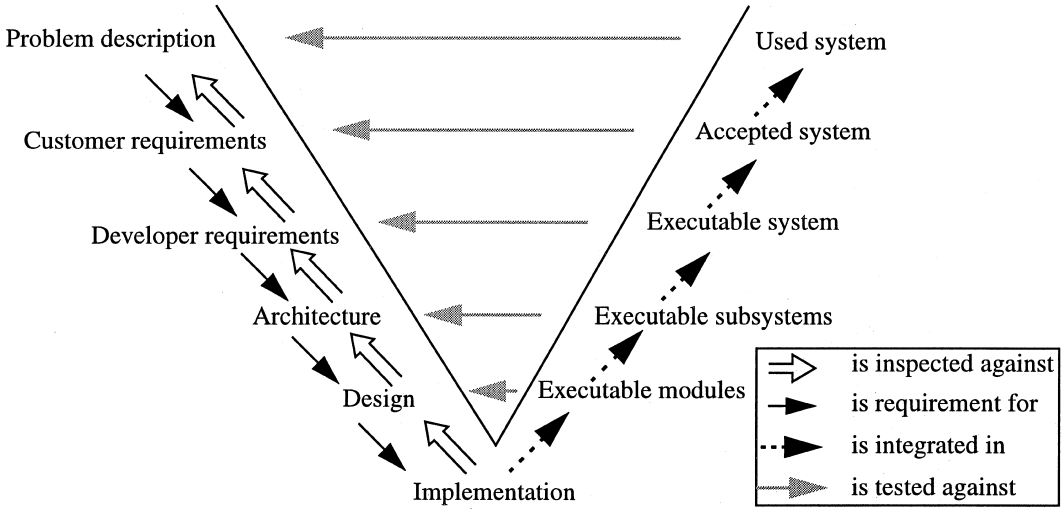

Fig. 4. A generic software development model.

developer requirements, customer requirements, and the problem description, respectively. The documentation of the test cases for each level is usually attached to the level-specific document (e.g., the acceptance test cases are attached to the customer requirements document).

6. A life-cycle taxonomy for software inspection

Most inspection variations take a one-size-fits-all approach (Johnson, 1998b): the same variation is assumed to work equally well regardless of which life-cycle product is inspected. However, we realized in the literature that some variations are tightly coupled to the inspected product type. Hence, we present in this section a life-cycle taxonomy for software inspection which describes the 'conventional' inspection approach as well as suggested variations according to the different life-cycle products. In our discussion, we focus more on the technical and assessment dimension of software inspections than on the managerial, organizational, or tool dimension. To facilitate our discussion, we first present an overview of articles in the context of our generic software development model and then, continue by discussing in detail the presented inspection variations according to the different life-cycle products.

6.1. Overview

Fig. 5 presents an overview of articles describing inspection variations for different life-cycle products. In addition to the articles that only discuss the inspection of a specific life-cycle product, we also included some articles describing the inspection of several different products. For each product, we start by describing and summarizing vital issues of the conventional inspection approach, which we described in detail in Section 3, and we continue with presenting other inspection variations.

6.2. Inspection of problem description, customer requirements, and developer requirements

6.2.1. 'Conventional' inspection approach

The conventional inspection approach we presented in Section 3 can be easily adapted for inspecting early life-cycle artifacts. Examples can be found in Ackerman et al. (1989), Doolan (1992), Fowler (1986), Graden et al. (1986) and Shirey (1992). However, inspecting early life-cycle artifacts is not commonly practiced in industry (Shirey, 1992). Two main reasons appear to be responsible. First, early life-cycle artifacts are often not as precise as the life-cycle artifacts developed later on (Cheng and Jeffrey, 1996). Therefore, understanding the semantics of the artifact may be more difficult for inspectors, which then makes defect detection in these software products more challenging than defect detection in later life-cycle artifacts. This is particularly the case if the key function and quality properties that the inspected artifact is to fulfill are ill-defined. Second, the inspected artifact may be the first written document in the software development project. This is, by nature, the case for a problem description. Hence, inspectors cannot compare or leverage the inspected artifact, e.g., the problem description, with another artifact previously developed. In this case, the inspection result heavily depends on the reading technique and the level of skill, knowledge, and experience of the inspectors. This problem is sometimes alleviated by providing reading techniques. The reading techniques offered to inspectors for defect detection in early life-cycle products are Ad-hoc (Doolan, 1992), checklist-based reading

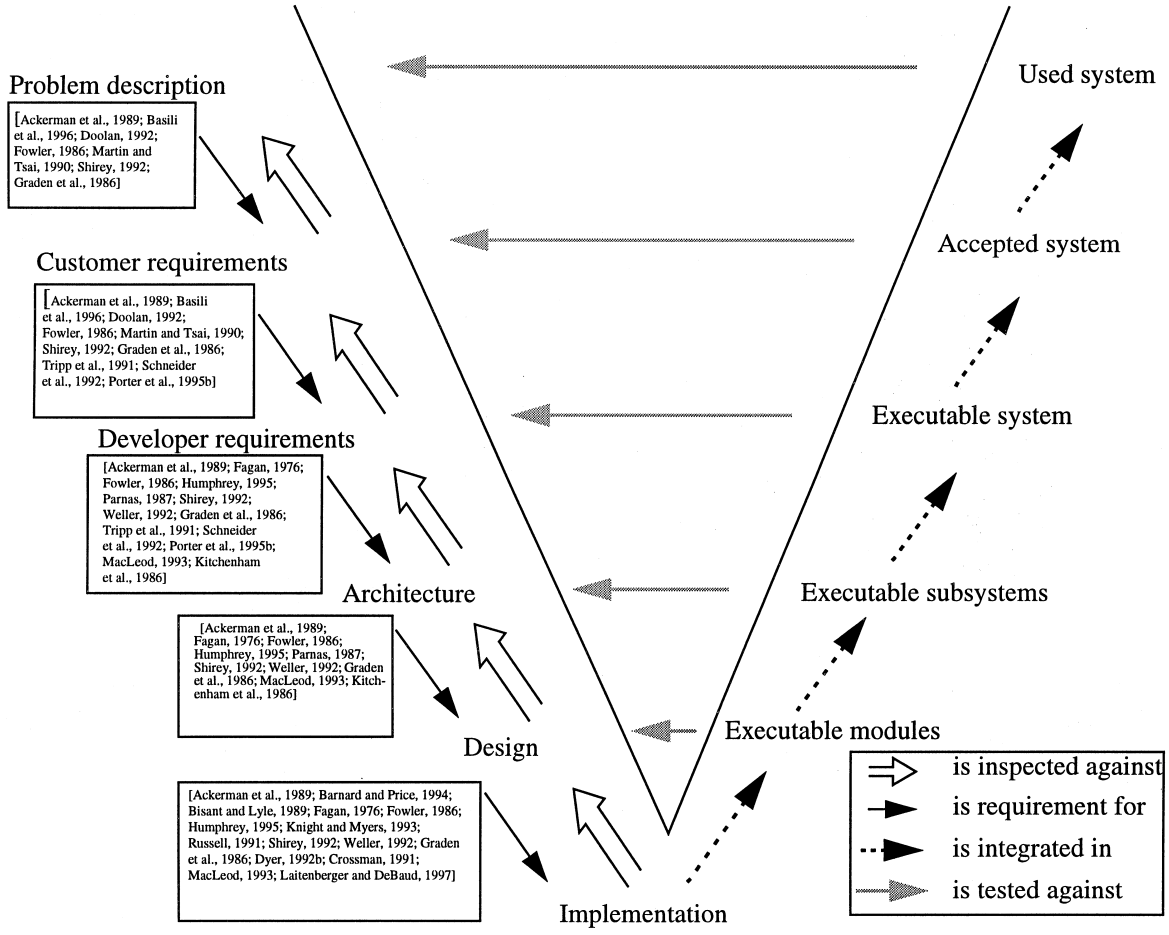

Fig. 5. Inspection variations according to the generic software development model.

(Gilb and Graham, 1993), Defect-based Reading (Porter et al., 1995b), and Perspective-based Reading (Basili et al., 1996). In many cases, the lack of reading support is compensated by increasing the number of inspectors (Bourgeois, 1996; Doolan, 1992), so that, on average, the number of inspectors is higher for early life-cycle products than for products developed later on. Of course, this increases inspection cost, which may prevent managers from organizing the inspection of these products.

6.2.2. Suggested variation: N-fold inspection

Martin et al. proposed the *N-fold inspection method* (Martin and Tsai, 1990; Schneider et al., 1992). This inspection method is based on the hypotheses that a single inspection team can find only a fraction of the defects in a software product and that multiple teams will not significantly duplicate each others efforts. In an *N*-fold inspection, *N* teams each carry out parallel independent inspections of the same software artifact. In a sense, *N*-fold inspection scales up some ideas of scenario-based reading techniques, which are applied in the conventional inspection approach on an individual level, to a team level. The inspection participants of each independent inspection follow the various inspection steps of a conventional inspection as outlined in Section 2, that is, individual defect detection with an Ad-hoc reading technique and defect collection in a meeting. The *N*-Fold inspection approach ends with a final step in which the results of each inspection team are merged into one defect list. It has been hypothesized that *N* different teams will detect more defects than a single large inspection team. In fact, there already exists empirical evidence which confirms this hypothesis (Tripp et al., 1991). However, if *N* independent teams inspect one particular document, inspection cost will be high. This limits this inspection approach to the inspection of early life-cycle artifacts for which very high quality really does matter, such as the aircraft industry, or safety critical systems (Tripp et al., 1991).

6.3. Inspection of architecture and design artifacts

6.3.1. Conventional inspection approach

The conventional inspection approach for inspecting architecture and design documents is described, for

instance, in Ackerman et al. (1989), Fagan (1976), Fowler (1986), Graden et al. (1986), Humphrey (1995), Kitchenham et al. (1986), MacLeod (1993), Shirey (1992) and Weller (1992). The reading techniques available for defect detecting in architecture and design artifacts are Ad-hoc and checklist-based reading.

6.3.2. Suggested variation: Active Design Reviews

Parnas (1987) and Parnas and Weiss (1985) suggest an inspection method denoted as *Active Design Reviews* (ADR) for inspecting design documents. The authors believe that in conventional design inspections, inspectors are given too much information to examine, and that they must participate in large meetings which only allow for limited interaction between inspectors and author. To tackle theses issues, inspectors are chosen based on their specific level of expertise skills and assigned to ensure thorough coverage of design documents. Only two roles are defined within the ADR process. An inspector has the expected responsibility of finding defects, while the designer is the author of the design being scrutinized. There is no indication of who is responsible for setting up and coordinating the review. The ADR process consists of three steps. It begins with an overview step, where the designer presents an overview of the design and meeting times are set. The next step is the defect detection step for which the author provides questionnaires to guide the inspectors. The questions are designed such that they can only be answered by careful study of the design document, that is, inspectors have to elaborate the answer instead of stating yes/no. Some of the questions reinforce an active inspection role by making assertions about design decisions. For example, he or she may be asked to write a program segment to implement a particular design in a low-level design document being inspected. The final step is defect collection, which is performed in inspection meetings. However, each inspection meeting is broken up into several smaller, specialized meetings, each of which concentrates on one quality property of the artifact. An example is checking consistency between assumptions and functions, that is, determining whether assumptions are consistent and detailed enough to ensure that functions can be correctly implemented and used.

ADR is an important inspection variation because ADR inspectors are guided by a series of questions posed by the author(s) of the design in order to encourage a thorough defect detection step. Thus, inspectors get reading support when scrutinizing a design document. Although little empirical evidence shows the effectiveness of this approach, other researchers based their inspection variations upon these ideas (Cheng and Jeffrey, 1996; Knight and Myers, 1991).

6.4. Inspection of implementations

6.4.1. Conventional inspection approach

Software inspections have most often been applied to implementations, that is, code artifacts. Examples can be found in Ackerman et al. (1989), Barnard and Price (1994), Crossman (1991), Fagan (1976), Fowler (1986), Graden et al. (1986), Humphrey (1995), MacLeod (1993), Shirey (1992), and Weller (1992). The currently available reading techniques for defect detection in implementations are Ad-hoc (Ackerman et al., 1989), Checklist-based reading (Fagan, 1976), Reading by Stepwise Abstraction (Dyer, 1992a), and Perspective-based reading (Laitenberger and DeBaud, 1997). The current state of the practice is to use checklists for defect detection in implementations. Despite the large portion of work in the area of software inspection, the number of available reading techniques even for implementations is rather low. A possible reason is that, in the past, too much attention has been paid to the inspection process and group issues and too little to the individuals carrying out the reading, that is, the defect detection activity in the privacy of their own offices. It is only recently that this issue has been tackled.

An important aspect regarding implementation is the influence of the chosen programming language. So far, most articles present the inspection of functional code, such as Pascal, Fortran, or C-code. Inspection and more specifically, defect detection in object-oriented code may impose additional problems, such as inheritance, dynamic binding, or polymorphism (Hatton, 1998; Macdonald et al., 1996b). Object-oriented concepts therefore raise new questions for inspection, such as how to ensure the quality of the inheritance structure? More importantly, how to ensure the quality of an artifact by a static analysis approach, such as inspection, when dynamic binding is applied? These are only some of the questions that must be tackled in the context of object-oriented development methods. For others, we refer to Jones (1994). However, these questions highlight the fact that inspection of early products may become even more important, regardless of the development process.

6.4.2. Suggested variation-1: phased inspection

Knight and Myers (1991, 1993) suggested the Phased Inspection method. The main idea behind Phased inspection is for each inspection phase to be divided into several mini-inspections or phases. Mini-inspections are conducted by one or more inspectors and are aimed at detecting defects of one particular class or type. This is the most important difference to conventional inspections which check for many classes or types of defects in a single examination. If there is more than one inspector, they will meet just to reconcile their defect list. The phases are done in sequence, that is, inspection does not

progress to the next phase until rework has been completed on the previous phase.

Although Knight and Myers state that phased inspections are intended to be used on any work product, they only present some empirical evidence of the effectiveness of this approach for the code inspections. However, Porter et al. (1997) argue based on the results of their experiments, that multiple session inspections, that is, mini-inspections, with repair in between are not more effective for defect detection but are more costly than conventional inspections. This may be one explanation why we did not find extensive use of the phased inspection approach in practice.

6.4.3. Suggested variation-2: Verification-based Inspection

Verification-based Inspection is an inspection variation used in conjunction with the Cleanroom software development method. Although this method requires the author(s) to perform various inspections of work products, the inspection process itself is not well described in the literature. We found that it consists of at least one step, in which individual inspectors examine the work product using a reading technique denoted as Reading by Stepwise Abstraction. This reading technique is limited to code artifact, though it provides a more formal approach for inspectors to check the functional correctness (Dyer, 1992b). We found little information on the inspection process after the individual defect detection step. However, the Cleanroom approach is one of the few development approaches in which defect detection and inspection activities are tightly integrated in and coupled with development activities.The Cleanroom approach and its integrated inspection approach has been applied in several development projects (Basili, 1997; Dyer, 1992a; Deck, 1994).

6.5. Inspection of testcases

The importance of inspecting testcases is pointed out several times in the literature (Ackerman et al., 1989; Graden et al., 1986; Shirey, 1992). It stems from the fact that the participation of developers in the inspection of testcases alerts them to user expectations before the software product is developed. However, for the inspection of testcases, no inspection variations different from the conventional approaches have been described.

7. Future research direction

One of the most challenging and significant avenues of research in the software engineering discipline is the investigation of how to assure software quality, reduce development cost, and keep software projects within schedule. Software inspection is a practical approach to help tackle all three issues. However, there still exist challenging questions that need to be addressed by researchers in the future. Examples are: (1) What is the most cost-effective inspection variation? (2) When to stop inspection? (3) How does the number and experience of inspectors influence software inspection? (4) How does software inspection scale up (e.g., how to introduce inspections in projects in which changes are made to a large system that has not been inspected so far)? (5) How to provide adequate reading techniques for inspectors? (6) How to support software inspection with tools? (7) How do software inspections depend on the type of software artifact? For instance, are inspections of functional software artifacts different from the inspection of object-oriented artifacts, and if so, what are the consequences? (8) How much to inspect?

Although each of these questions in isolation provides a fruitful area for future inspection research, we encourage the research community to tackle them in the context of a larger framework or theory. There, the underlying research goal is to provide guidance for practitioners on the most successful inspection approach in the context of the practitioner's software development situation. [4] The most successful inspection approach is the one that helps to find most of the defects in the inspected artifact, has an optimal cost/benefit ratio, and can be performed within the specified time frame. These three characteristics can be aligned to the quality/effort/duration-subdimensions of our taxonomy. Although there might be a limitless number of factors that induce variations on the subdimensions, our survey allows us to distill what we believe are the most important influential factors. This puts us into a position where we can apply a causal modeling approach to theory construction (Blalock, 1979). For each subdimension, we describe a causal model and its graphical representation in the form of a path diagram (Pedhazur, 1982) describing the relationships between the subdimension and the influential factors. The set of causal models defines the theory. Such a theory is beneficial for three reasons. First, practitioners as well as researchers gain insight into the major factors influencing software inspection. Second, a theory offers the possibility for a researcher to integrate his or her own work into a broader context and to highlight his or her methodological or empirical contribution to the inspection field in a systematic manner. Finally, in the long run, the accumulation of knowledge in the context of a theory makes software inspection an even more effective approach for overcoming software quality deficiencies and cost overruns. In the following, we present and discuss three causal models, which

[4] This does not imply that there is *one* best-fitting approach for *all* development situations.

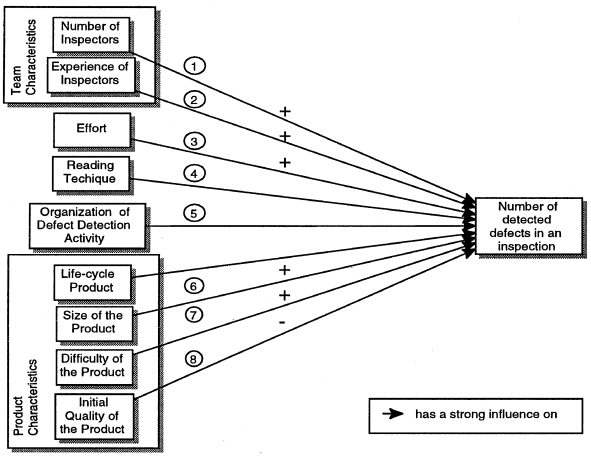

Fig. 6. Path diagram for explaining the number of defects detected.

should be regarded as a starting point for further refinement and elaboration.

7.1. A causal model for explaining inspection quality

A high quality inspection must ensure that most of the detectable defects in a software product are, indeed, detected. Therefore, we are interested in the factors that have an impact on the number of defects detected. Fig. 6 depicts the ones we isolated from the literature. There, the principal factors are the team characteristics, effort, the reading technique, the organization of the defect detection activity, and product characteristics. The major team characteristics are the number of inspectors and their experience. The major product characteristics are the type of product that is inspected, the difficulty of the product, such as its complexity, the size of the product, and its initial quality.

In Fig. 6 and in the two Figures that follow, an arrow linking a given pair of variables (X, Y) indicates that there is assumed to be a direct causal link between these variables. [5] A '+' sign above an arrow must be interpreted as statements of the form 'an increase in X will produce (cause) an increase in Y'. A '−' sign indicates statements of the form 'an increase in X will produce a decrease in Y'. In addition, we give each relationship a number so that we can later on refer to the articles dealing with the relationship.

The principal factors impact the number of defects detected in an inspection in the following manner:

- Increasing the number of inspectors is expected to increase the number of defects detected in an inspection. However, there will be a ceiling effect after which adding an additional inspector does not necessarily pay off in more detected defects. The optimal number of inspectors needs to be determined empirically as, for example, presented in Madachy et al. (1993) or Bourgeois (1996).
- Using very experienced inspectors is expected to increase the number of detected defects in an inspection. This stems from the fact that if an inspector is well versed in the application domain, he or she already knows many potential pitfalls and problem spots.
- Spending more effort for defect detection is expected to increase the number of defects detected in an inspection.
- The difficulty of a product is related to the defect-proneness. This means that a more difficult software product contains more defects. Difficulty may be, for example, defined as the complexity of the inspected product (McCabe, 1976). This relationship, therefore, translates to the following expectation: The more difficult the inspected product is, the more defects are expected to be detected in an inspection.
- The larger the size of an inspected product, the more defects are detected in an inspection (assuming a constant defect density in the inspected product).
- The higher the initial quality of the inspected document, the lower the number of detected defects.

The factors 'life-cycle product', 'reading technique' and 'organization of the defect detection activity' are not that easy to quantify, and we refer to previous parts of this survey for a detailed discussion. Moreover, we have to mention that other factors, such as tool support, may have an impact as well. Although these factors also need to be investigated, we did not include them here because we focused on the most prevalent factors.

According to the life-cycle structure of our survey, Table 4 presents the articles in which the expectations were mentioned and, in a few cases, empirically investigated.

7.2. A causal model for explaining inspection effort

As depicted in Fig. 7, the principal factors determining inspection effort are team and product characteristics. The major team characteristics that impact inspection effort are the number of persons (not only inspectors) involved in an inspection and their experience. The major product characteristics are the type, the difficulty, and the size of the product.

The principal factors impact inspection effort in the following manner.

- Increasing the number of people increases inspection effort.
- The more experienced the inspectors, the less effort they consume for defect detection and, thus, for

[5] We do not assume that the relationships are necessarily linear and additive. Furthermore, the causal models are a simplification in the sense that we neglect interactions among the different influential factors. Such an interaction may be, for example, the checking rate (Gilb and Graham, 1993), which is defined as the ratio of the size and the defect detection effort.

Table 4
Articles describing the relationship between influential factors and the number of defects detected

	Number of inspectors	Experience of inspectors	Effort	Reading technique	Organization of defect detection	Size of the product	Difficulty of product	Initial quality of product
Customer and developer requirements		(Fowler, 1986)		(Porter et al., 1995b; Basili et al., 1996; Cheng and Jeffrey, 1996	(Tripp et al., 1991)			
Architecture and design	(Fagan, 1986)	(Parnas, 1987)	(Christenson et al., 1990; Raz and Yaung, 1997)	(Parnas, 1987)		(Parnas, 1987)	(Christenson et al., 1990; Raz and Yaung, 1997; Weller, 1993)	(Humphrey, 1995)
Implementation	(Fagan, 1986; Blakely and Boles, 1991; Strauss and Ebenau, 1993; Weller, 1993)	(Barnard and Price, 1994; Porter et al., 1997; Laitenberger and DeBaud, 1997)	(Barnard and Price, 1994; Bourgeois, 1996; Christenson et al., 1990; Weller, 1993; Franz and Shih, 1994),	(Linger et al., 1979; Dyer, 1992b; Laitenberger and DeBaud, 1997)	(Fagan, 1986; Votta, 1993; Porter and Johnson, 1997; Land et al., 1997)	(Porter et al., 1998)	(Barnard and Price, 1994; Christenson et al., 1990)	(Humphrey, 1995)
Test cases								

the overall inspection. However, one must be aware that the more experienced inspectors are often the more expensive ones, so that the overall inspection costs are not necessarily decreased commensurately.
- The more difficult (e.g., complex) a product, the more effort is required for inspecting it.
- The larger the size of the inspected product, the more effort is required for its inspection.

Moreover, the inspection effort is determined by which life-cycle product is inspected. Here again, other factors, such as reading technique, may influence inspection effort.

According to the life-cycle structure of our survey, Table 5 presents the articles in which the relationships were discussed and examined.

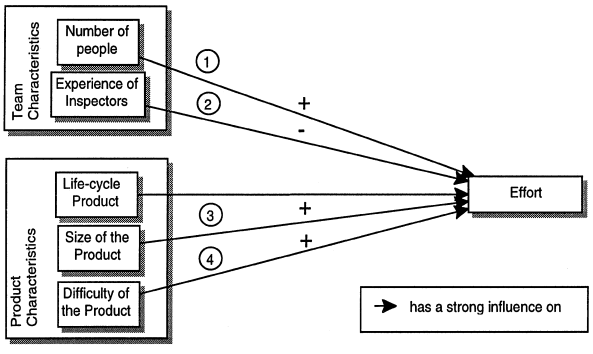

Fig. 7. Path diagram for explaining inspection effort.

7.3. A causal model for explaining inspection duration

As depicted in Fig. 8, the most important factors determining inspection duration are the team characteristics and the organization of the inspection process. The team characteristics involve the number of people and the number of teams. All these factors are hypothesized to have a positive relationship with duration, although few solid data is currently available. The scarcity of work regarding inspection duration is the reason why we do not present a table of relevant articles. The articles that discuss inspection duration are Bourgeois (1996), Porter et al. (1997) and Votta (1993).

7.4. Discussion

Transferring software inspection into development organizations as well as bridging the gap between the state-of-the-art and the state-of-the-practice clearly requires concerted efforts by both researchers and practitioners. One major obstacle to operationalize the transition seems to be a scarcity of experimental work that is sufficiently solid and well analysed to justify the risks entailed for transition to industrial practice. Experimental approaches, as, for example, presented in Jalote and Haragopal (1998), play a vital role in convincing inspection participants as well as their supervisors that software inspections are beneficial and allow a smooth transition from research to practice. Furthermore, the data collected in these experiments help determine key success factors for software inspection and

Table 5
Articles describing the relationship between influential factors and inspection effort

	Number of people	Experience of inspectors	Size of the product	Difficulty of the product
Customer and developer requirements				
Architecture and design	(Fagan, 1976; Parnas, 1987)			(Raz and Yaung, 1997)
Implementation	(Bisant and Lyle, 1989; Fagan, 1976; Weller, 1993)		(Porter et al., 1997)	(Bourgeois, 1996)
Test cases				

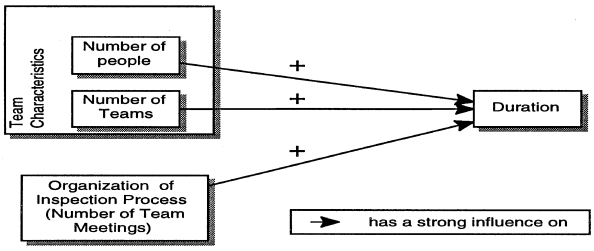

Fig. 8. Path diagram for explaining inspection duration.

help establish the relationships among them. However, sound experimentation requires viable theories or models to understand as well as to predict factors that bias software inspection. So far, few models or theories have been presented for understanding or prediction, despite many numerical studies presented in literature. This is particularly the case for the inspection of life-cycle products other than code. Hence, we made an initial step towards theory construction by presenting three causal models. Such a theory points out promising areas for future research and provides a starting point for systematically accumulating knowledge in the inspection field. Both researchers and practitioners need to refine the theory, study the functional form of relationships, and investigate interactions among the different factors. Regarding code inspections, some researcher have already followed this process (Porter et al., 1998; Seaman and Basili, 1998).

8. Conclusion

In this paper, we presented an encompassing, life-cycle centric survey of work in the area of software inspection. The survey consisted of two main sections: The first introduced a detailed description of the core concepts and relationships that together define the field of software inspection. The second elaborated a taxonomy that uses a generic development life-cycle to contextualize software inspection in detail.

This type of survey is beneficial to both practitioners and researchers: First, it provides a roadmap in the form of a contextualized, life-cycle taxonomy that allows the identification of available inspection methods and experience directly related to a particular life-cycle phase. This may be particularly interesting for practitioners, since they often want to tackle the quality deficiencies of concrete life-cycle products with software inspection. Yet, they often do not know which method or refinement to choose. Hence, this survey helps to quickly focus on the best-suited inspection approach adapted to a particular environment. Second, our work helps structure the large amount of published inspection work. This structure allows us to present the gist of the inspection work so far performed and helps practitioners as well as researchers characterize the nature of new work in the inspection field. In a sense, this structure also helps define a common vocabulary that depicts the software inspection field area. Third, our survey presents an overview of the current state of research as well as an analysis of today's knowledge in the field of software inspection. It integrates the knowledge into a theory that, together with the road map, can be particularly interesting for researchers to identify areas where little work has been done so far.

We have to state that each survey has its limitations. At the time of publication, this can only be a snapshot of the work that is currently in progress. Furthermore, a survey usually represents only a fraction of articles that are available on a subject. However, in this case we analysed more than 400 references. We made the references of papers available via the World Wide Web (Fraunhofer Institute for Experimental Software Engineering, 1998) and encourage other researchers to send us their inspection articles or references to integrate them into our bibliography.

Acknowledgements

We thank the anonymous reviewers for their helpful improvement suggestions. We are also grateful to Tom Gilb, Khaled El Emam and Sonnhild Namingha for their comments on this paper.

References

Ackerman, A.F., Buchwald, L.S., Lewsky, F.H., 1989. Software inspections: an effective verification process. IEEE Software 6 (3), 31–36.

Association of Computing Machinery, 1998. The ACM Digital Library. http://www.acm.org/dl/.

Barnard, J., Price, A., 1994. Managing code inspection information. IEEE Software 11 (2), 59–69.

Basili, V., Green, S., Laitenberger, O., Lanubile, F., Shull, F., Sorumgard, S., Zelkowitz, M., 1996. The empirical investigation of perspective-based reading. J. Empirical Software Eng. 2 (1), 133–164.

Basili, V.R., 1997. Evolving and packaging reading technologies. J. Systems Software 38 (1).

Basili, V.R., Selby, R.W., 1987. Comparing the effectiveness of software testing techniques. IEEE Trans. Software Eng. 13 (12), 1278–1296.

Belli, F., Crisan, R., 1996. Towards automation of checklist-based code-reviews. In: Proceedings of the Eighth International Symposium on Software Reliability Engineering.

Bisant, D.B., Lyle, J.R., 1989. A two-person inspection method to improve programming productivity. IEEE Trans. Software Eng. 15 (10), 1294–1304.

Blakely, F.W., Boles, M.E., 1991. A case study of code inspections. Hewlett–Packard J. 42 (4), 58–63.

Blalock, H.M., 1979. Theory Construction. Prentice-Hall, Englewood Cliffs, NJ.

Boehm, B.W., 1981. Software Engineering Economics. Advances in Computing Science and Technology. Prentice-Hall, Englewood Cliffs.

Bourgeois, K.V., 1996. Process insights from a large-scale software inspections data analysis. Cross Talk, J. Defense Software Eng., 17–23.

Briand, L., El-Emam, K., Freimut, B., Laitenberger, O., 1997. Quantitative evaluation of capture-recapture models to control software inspections. In: Proceedings of the Ninth International Symposium on Software Reliability Engineering.

Briand, L., El-Emam, K., Fussbroich, T., Laitenberger, O., 1998. Using simulation to build inspection efficiency Benchmarks for development projects. In: Proceedings of the 20th International Conference on Software Engineering, pp. 340–349.

Briand, L.C., Differding, C.M., Rombach, H.D., 1996. Practical guidelines for measurement based process improvement. Software Process 2 (4), 253–280.

Bröhl, A.-P., Dröschel, W., 1995. Das V-Modell. Oldenburg, Munich.

Brothers, L., Sembugamoorthy, V., Muller, M., 1990. ICICLE: Groupware for code inspection. In: Proceedings of the ACM Conference on Computer Supported Cooperative Work, pp. 169–181.

Brykczynski, B., Wheeler, D.A., 1993. An annotated bibliography on software inspections. ACM SIGSOFT Software Eng. Notes 18 (1), 81–88.

Cheng, B., Jeffrey, R., 1996. Comparing inspection strategies for software requirements specifications. In: Proceedings of the 1996 Australian Software Engineering Conference, pp. 203–211.

Chernak, Y., 1996. A statistical approach to the inspection checklist formal synthesis and improvement. IEEE Trans. Software Eng. 22 (12), 866–874.

Christenson, D.A., Steel, H.T., Lamperez, A.J., 1990. Statistical quality control applied to code inspections. IEEE J. Selected Areas Commun. 8 (2), 196–200.

Collofello, J.S., Woodfield, S.N., 1989. Evaluating the effectiveness of reliability-assurance techniques. J. Systems Software 9, 191–195.

Cooper, H.M., 1982. Scientific guidelines for conducting integrative research reviews. Rev. Educational Res. 52 (2), 291–302.

Crossman, T.D., 1991. A method of controlling quality of applications software. South African Comput. J. 5, 70–74.

Deck, M., 1994. Cleanroom Software Engineering to reduce Software Cost. Technical report. Cleanroom Software Engineering Associates, 6894 Flagstaff Rd. Boulder, CO 80302.

DeMarco, T., 1982. Controlling Software Projects. Yourdon Press, New York.

Dennis, A., Valacich, J., 1993. Computer brainstorms: more heads are better than one. J. Appl. Social Psychology 78 (4), 531–537.

Doolan, E.P., 1992. Experience with Fagan's inspection method. Software-Practice Experience 22 (3), 173–182.

Dyer, M., 1992a. The Cleanroom Approach to Quality Software Development. Wiley, Chichester.

Dyer, M., 1992b. Verification-based inspection. In: Proceedings of the 26th Annual Hawaii International Conference on System Sciences, pp. 418–427.

Eick, S.G., Loader, C.R., Long, M.D., Votta, L.G., VanderWiel, S., 1992. Estimating software fault content before coding. In: Proceedings of the 14th International Conference on Software Engineering, pp. 59–65.

Fagan, M.E., 1976. Design and code inspections to reduce errors in program development. IBM Sys. J. 15 (3), 182–211.

Fagan, M.E., 1986. Advances in software inspections. IEEE Trans. Software Eng. 12 (7), 744–751.

Fowler, P.J., 1986. In-process inspections of workproducts at AT&T. AT&T Technical J. 65 (2), 102–112.

Franz, L.A., Shih, J.C., 1994. Estimating the value of inspections and early testing for software projects. CS-TR-6 Hewlett-Packard J.

Fraunhofer Institute for Experimental Software Engineering, 1998. An Inspection Bibliography. http:// www.iese.fhg.de/ISE/Inspbib/inspection.html.

Freedman, D.P., Weinberg, G.M., 1990. Handbook of Walkthroughs, Inspections, and Technical Reviews, 3rd ed. Dorset House Publishing, New York.

Gilb, T., Graham, D., 1993. Software Inspection. Addison-Wesley, Reading, MA.

Gintell, J., Houde, M., McKenney, R., 1995. Lessons learned by building and using scrutiny, a collaborative software inspection system. In: Proceedings of the Seventh International Workshop on Computer-Aided Software Engineering, pp. 350–357.

Graden, M.E., Horsley, P.S., Pingel, T.C., 1986. The effects of software inspections on a major telecommunications-project. AT&T Technical J. 65 (3), 32–40.

Grady, R.B., 1994. Successfully applying software metrics. IEEE Comput. 27 (9), 18–25.

Grady, R.B., vanSlack, T., 1994. Key lessons in achieving widespread inspection use. IEEE Software 11 (4), 46–57.

Hatton, L., 1998. Does OO Sync with how we think?. IEEE Software 15 (3), 46–54.

Hetzel, W.C., 1976. An Experimental Analysis of Program Verification Methods. PhD thesis, University of North Carolina at Chapel Hill, Department of Computer Science.

Humphrey, W.H., 1995. A Discipline for Software Engineering. Addison-Wesley, Reading, MA.

International Software Engineering Research Network, 1998. Bibliography of the International Software Engineering Research Network. http://www.iese.fhg.de/ISERN/pub/isern_biblio_tech.html.

Jackson, A., Hoffman, D., 1994. Inspecting module interface specifications. Software Testing, Verification Reliability 4 (2), 101–117.

Jalote, P., Haragopal, M., 1998. Overcoming the NAH syndrome for inspection deployment. In: Proceedings of the 20th International Conference on Software Engineering, pp. 371–378.

Johnson, P., 1998a. The WWW Formal Technical Review Archive. http://zero.ics.hawaii.edu/johnson/ FTR.

Johnson, P.M., 1998b. Reengineering inspection. Commun. ACM 41 (2), 49–52.

Johnson, P.M., Tjahjono, D., 1993. Improving software quality through computer supported collaborative review. In: Proceedings of the 19th International Conference on Software Engineering, pp 61–76.

Johnson, P.M., Tjahjono, D. (Ed.), 1997. Assessing Software Review Meetings: A Controlled Experimental Study Using CSRS, 118–127. ACM Press, New York.

Jones, C., 1994. Gaps in the object-oriented paradigm. IEEE Comput. 27 (6), 90–91.

Jones, C., 1996. Software defect-removal efficiency. IEEE Comput. 29 (4), 94–95.

Kamsties, E., Lott, C.M., 1995. An empirical evaluation of three defect-detection techniques. In: Schäfer, W., Botella, P. (Eds.), Proceedings of the Fifth European Software Engineering Conference, pp. 362–383. Lecture Notes in Computer Science Nr. 989, Springer, Berlin.

Kan, S.H., 1995. Metrics and Models in Software Quality Engineering. Addison-Wesley, Reading, MA.

Kelly, J.C., Sherif, J.S., Hops, J., 1992. An analysis of defect densities found during software inspections. J. Sys. Software 17, 111–117.

Kim, L.P.W., Sauer, C., Jeffery, R., 1995. A framework for software development technical reviews. Software Quality and Productivity: Theory, Practice, Education and Training.

Kitchenham, B., Kitchenham, A., Fellows, J., 1986. The effects of inspections on software quality and productivity. Technical Report 1, ICL Technical J.

Knight, J.C., Myers, E.A., 1991. Phased inspections and their implementation. ACM SIGSOFT Software Engineering Notes 16 (3), 29–35.

Knight, J.C., Myers, E.A., 1993. An improved inspection technique. Commun. ACM 36 (11), 51–61.

Kusumoto, S., 1993. Quantitative Evaluation of Software Reviews and Testing Processes. PhD thesis, Faculty of the Engineering Science of Osake University.

Laitenberger, O., DeBaud, J.-M., 1997. Perspective-based reading of code documents at robert bosch GmbH. Information Software Technol. 39, 781–791.

Land, L.P.W., Sauer, C., Jeffery, R., 1997. Validating the defect detection performance advantage of group designs for software reviews: report of a laboratory experiment using program code. In: Jazayeri, M., Schauer, H. (Eds.), Lecture Notes in Computer Science No 1301, Proceedings of the Sixth European Software Engineering Conference, pp. 294–309.

Letovsky, S., Pinto, J., Lampert, R., Soloway, E., 1987. A cognitive analysis of a code inspection (Ed.), Empirical Studies Programming, 231–247.

Levine, J.M., Moreland, R.L., 1990. Progress in small group research. Annual Rev. Psychology 41, 585–634.

Linger, R.C., Mills, H.D., Witt, B.I., 1979. Structured Programming: Theory and Practice. Addison-Wesley, Reading, MA.

Macdonald, F., 1997. Assist v1.1 User Manual. Technical Report RR-96-199 [EFoCS-22-96]. Empirical Foundations of Computer Science, (EFoCS), University of Strathclyde, UK.

Macdonald, F., Miller, J., 1995. Modelling Software Inspection Methods for the Application of Tool Support. Technical Report RR-95-196 [EFoCS-22-96]. Empirical Foundations of Computer Science, (EFoCS), University of Strathclyde, UK.

Macdonald, F., Miller, J., Brooks, A., Roper, M., Wood, M., 1996b. Applying inspection to object oriented software. Software Testing, Verification Reliability 6, 61–82.

Macdonald, F., Miller, J., Brooks, A., Roper, M., Wood, M., 1996a. Automating the software inspection process. Automated Software Eng. 3 (193), 193–218.

MacLeod, J.M., 1993. Implementing and sustaining a software inspection program in an R&D environment. Hewlett–Packard J.

Madachy, R., Little, L., Fan, S., 1993. Analysis of a successful inspection program. In: Procceding of the 18th Annual NASA Software Eng. Laboratory Workshop, pp. 176–198.

Marciniak, J.J., 1994. Reviews and audits. In: Marciniak, J.J. (Ed.), Encyclopedia of Software Engineering, vol. 2. Wiley, Chichester, pp. 1084–1090..

Martin, J., Tsai, W.T., 1990. N-fold inspection: a requirements analysis technique. Commun. ACM 33 (2), 225–232.

Mashayekhi, V., Drake, J.M., Tsai, W.T., Riedl, J., 1993. Distributed, collaborative software inspection. IEEE Software 10, 66–75.

McCabe, T.J., 1976. A complexity measure. IEEE Trans. Software Eng. 2 (4), 308–320.

McGibbon, T., 1996. A Business Case for Software Process Improvement. Technical Report F30602-92-C-0158. Data & Analysis Center for Software (DACS). URL: http://www.dacs.com/techs/roi.soar/soar.html.

Murphy, P., Miller, J., 1997. A process for asynchronous software inspection. In: Proceedings of the Eighth International Workshop on Software Technology and Engineering Practice, pp. 96–104.

Myers, G.J., 1978. A controlled experiment in program testing and code walkthroughs/inspections. Commun. ACM 21 (9), 760–768.

National Aeronautics and Space Administration, 1993. Software Formal Inspection Guidebook. Technical Report NASA-GB-A302. National Aeronautics and Space Administration. http://satc.gs-fc.nasa.gov/fi/fipage.html.

OCLC, 1998. Online Computer Library Center. http://www.oclc.org/oclc/menu/home1.html.

Parnas, D.L., 1987. Active design reviews: principles and practice. J. Sys. Software 7, 259–265.

Parnas, D.L., Weiss, D., 1985. Active design reviews: principles and practices. In: Proceedings of the Eighth International Conference on Software Engineering, pp. 132–136. Also Available as NRL Report 8927, 18 November 1985.

Pedhazur, E.J., 1982. Multiple Regression in Behavioral Research, 2nd ed. Harcourt Brace College, New York.

Perpich, J., Perry, D., Porter, A., Votta, L., Wade, M., 1997. Anywhere, anytime code inspections: using the web to remove inspection bottlenecks in large-scale software development. In: Proceedings of the 19th International Conference on Software Engineering, pp. 14–21.

Perry, D.E., Porter, A., Votta, L.G., Wade, M.W., 1996. Evaluating workflow and process automation in wide-area software development. In: Montangero, C. (Ed.), Proceedings of the Fifth European Workshop on Software Process Technology, Lecture Notes in Computer Science Nr. 1149. Springer, Berlin, Heidelberg, pp. 188–193.

Porter, A.A., Johnson, P.M., 1997. Assessing software review meetings: results of a comparative analysis of two experimental studies. IEEE Trans. Software Eng. 23 (3), 129–144.

Porter, A.A., Siy, H., Mockus, A., Votta, L., 1998. Understanding the sources of variation in software inspections. ACM Trans. Software Eng. Methodology 7 (1), 41–79.

Porter, A.A., Siy, H., Votta, L.G., 1995a. A Review of Software Inspections. Technical Report CS-TR-3552, UMIACS-TR-95-104. Department of Computer Science, University of Maryland, College Park, Maryland 20742.

Porter, A.A., Siy, H.P., Toman, C.A., Votta, L.G., 1997. An experiment to assess the cost-benefits of code inspections in large scale software development. IEEE Trans. Software Eng. 23 (6), 329–346.

Porter, A.A., Votta, L.G., 1997. What makes inspections work. IEEE Software, 99–102.

Porter, A.A., Votta, L.G., Basili, V.R., 1995b. Comparing detection methods for software requirements inspections: a replicated experiment. IEEE Trans. Software Eng. 21 (6), 563–575.

Raz, T., Yaung, A.T., 1997. Factors affecting design inspection effectiveness in software development. Information Software Technol. 39, 297–305.

Reeve, J.T., 1991. Applying the fagan inspection technique. Quality Forum 17 (1), 40–47.

Remus, H., 1984. Integrated software validation in the view of inspections/reviews. Software Validation, 57–65.

Rifkin, S., Deimel, L., 1994. Applying program comprehension techniques to improve software inspection. In: Proceedings of the 19th Annual NASA Software Eng. Laboratory Workshop, NASA.

Rosenthal, R., 1979. The 'file drawer problem' and tolerance for null results. Psychological Bulletin 86 (3), 638–641.

Russell, G.W., 1991. Experience with inspection in ultralarge-scale developments. IEEE Software 8 (1), 25–31.

Sauer, C., Jeffery, R., Lau, L., Yetton, P., 1996. A behaviourally motivated programme for empirical research into software development technical review. Technical Report 96/5, Centre for Advanced Empirical Software Research, Sydney, Australia.

Schneider, G.M., Martin, J., Tsai, W.T., 1992. An experimental study of fault detection in user requirements documents. ACM Trans. Software Eng. Methodology 1 (2), 188–204.

Seaman, C.B., Basili, V.R., 1998. Communication and organization: an empirical study of discussion in inspection meetings. IEEE Trans. Software Eng. 24 (6), 559–572.

Shaw, M.E., 1976. Group Dynamics: The Psychology of Small Group Behaviour. McGraw-Hill, New York.

Shirey, G.C., 1992. How inspections fail? In: Proceedings of the Ninth International Conference on Testing Computer Software, pp. 151–159.

Stein, M., Riedl, J., Harner, S., Mashayekhi, V., 1997. A case study of distributed, asynchronous software inspection. In: Proceedings of the 19th International Conference on Software Engineering. IEEE Computer Society Press, pp. 107–117.

Strauss, S.H., Ebenau, R.G., 1993. Software Inspection Process. McGraw Hill Systems Design & Implementation Series.

Svendsen, F.N., 1992. Experience with inspection in the maintenance of software. In: Proceedings of the Second European Conference on Software Quality Assurance.

Tervonen, I., 1996. Support for quality-based design and inspection. IEEE Software 13 (1), 44–54.

Tjahjono, D., 1996. Exploring the effectiveness of formal technical review factor with CSRS, a collaborative software review system. PhD thesis, Department of Information and Computer Science, University of Hawaii.

Tripp, L.L., Stuck, W.F., Pflug, B.K., 1991. The application of multiple team inspections on a safety-critical software standard. In: Proceedings of the Fourth Software Engineering Standards Application Workshop. IEEE Computer Society Press, pp. 106–111.

Votta, L.G., 1993. Does every inspection need a meeting?. ACM Software Eng. Notes 18 (5), 107–114.

Weinberg, G.M., Freedman, D.P., 1984. Reviews, walkthroughs, and inspections. IEEE Trans. Software Eng. 12 (1), 68–72.

Weller, E.F., 1992. Experiences with inspections at Bull HN information system. In: Proceedings of the Fourth Annual Software Quality Workshop.

Weller, E.F., 1993. Lessons from three years of inspection data. IEEE Software 10 (5), 38–45.

Wenneson, G., 1985. Quality assurance software inspections at NASA Ames: metrics for feedback and modification. In: Proceedings of the 10th Annual Software Engineering Workshop.

Wheeler, D.A., Brykczinski, B., Meeson, R.N., 1996. Software Inspection – An Industrial Best Practice. IEEE Computer Society Press.

Wheeler, D.A., Brykczynski, B., Jr., R.N.M., 19 97. Software peer reviews. In: Thayer, R.H. (Ed.), Software Engineering Project Management. IEEE Computer Society, Silver Spring, MD.

Wiel, S.A.V., Votta, L.G., 1993. Assessing software designs using capture-recapture methods. IEEE Trans. Software Eng. 19 (11), 1045–1054.

Wohlin, C., Runeson, P., 1998. Defect content estimations from review data. In: Proceedings of the 20th International Conference on Software Engineering, pp. 400–409.

Wood, M., Roper, M., Brooks, A., Miller, J., 1997. Comparing and combining software defect detection techniques: a replicated empirical study. In: Jazayeri, M., Schauer, H. (Eds.), Lecture Notes in Computer Science No 1301. Proceeding of the Sixth European Software Engineering Conference, pp. 262–277.

Yourdon, E., 1989. Structured Walkthroughs, 4th ed. Prentice-Hall, New York.

Yourdon, E., 1997. Death March. Prentice-Hall, Englewood Cliffs.

Oliver Laitenberger received the degree Diplom-Informatiker (M.S.) in Computer Science with a minor in Economics from the University of Kaiserlautern, Germany, in 1996. He is currently a researcher in the department of 'Quality of Software Development' at the Fraunhofer Institute for Experimental Software Engineering (IESE) in Kaiserslautern. His main research interests concern the area of reviews and inspections as well as their empirical evaluation.

Jean-Marc DeBaud received a Ph.D. degree in Computer Science from Georgia Institute of Technology in 1996. Afterwards, he worked as a head of the 'Product Line department (PLA)' at the Fraunhofer Institute for Experimental Software Engineering (IESE) in Kaiserslautern. Currently, he pursues industrial interests.