A Survey of Software Inspection Checklists

Bill Brykczynski
Institute for Defense Analyses
1801 N. Beauregard St.
Alexandria, VA 22311-1772
bryk@ida.org

Abstract

Software inspection processes call for a checklist to provide reviewers with hints and recommendations for finding defects during the examination of a workproduct. Many checklists have been published since Michael Fagan created the inspection process in the mid-1970's. This paper surveys 117 checklists from 24 sources. Different categories of checklist items are discussed and examples are provided of good checklist items as well as those that should be avoided.

Keywords: Checklist, checklist item, software inspection, review.

1. Introduction

The software inspection process is generally considered a software engineering "best practice" [Wheeler 1996]. Inspection processes usually call for a reviewer to use a checklist in support of the examination of a workproduct [Fagan 1976, Gilb 1993]. The primary purpose of the checklist is to provide the reviewer with hints and recommendations for finding defects. Heuristics that are commonly suggested for creating an effective inspection checklist include:

1. Checklists should be regularly updated based on defect analysis. By updating checklists regularly, reviewers may be more likely to read and use them. If the checklist items are updated in response to frequently occurring defects, then it's more likely they will help the reviewer in finding additional defects.
2. Checklists should not be longer than a single page. A reviewer is less inclined to flip through pages of a checklist while examining, say, a code listing. A single-page checklist can be placed on a desk and read in close proximity to the product being examined.
3. Checklist items should be phrased in the form of a question (e.g., Has each variable been properly initialized before it is first used?). This heuristic is rather dubious, though, because all question-based checklist items could be re-phrased as imperative sentences (e.g., Verify that each variable as been properly initialized before it is first used.).
4. Checklist items should not be too general (e.g., Are all requirements complete, consistent, and unambiguous?).
5. Checklist items should not be used for conventions better enforced through other means (e.g., by the use of automated tools, entry/exit criteria before the inspection meeting).

The inspection checklist has received increased attention in the software engineering literature in the past five years. Fagan [1976] first described the inspection checklist and how it should be based on frequently occurring defects. In their book on the inspection process, Gilb and Graham [1993] discuss a number of issues relating to checklist construction, including the use of actual defects to populate checklist items. Porter et al. [1994, 1995] conducted experiments evaluating reviewer effectiveness using different types of checklists. Other researchers have suggested new and improved inspection processes that involve checklist techniques [Parnas 1987, Knight 1993]. Humphrey [1995, 1997] describes how to create personal checklists to be used by developers in reviewing their workproducts.

This paper surveys 117 checklists from 24 sources. These checklists are intended for use by reviewers as part of an inspection process, although several are for other forms of formal technical review [Yourdon 1989]. A summary of each checklist is provided, including the number of checklist items it contains and the workproduct it is intended for. This paper also examines different categories of checklist items and provides examples of good checklist items as well as those that should be avoided.

2. Checklist item categories

Many checklists have items that ask whether or not the workproduct being reviewed conforms to higher-order documents. Thus, a checklist for design may ask "Does the design fully implement its requirements?" and a checklist for code may ask "Does the code fully implement the design?" This kind of check is generally the first analysis performed by a reviewer and should be an implicit part of any review.

Most checklist items are intended to help a reviewer find defects. However, an inspection might have goals other than defect detection. Below are example checklist items that support goals other than defect detection:

1. "Is any part of the code a possible candidate for reuse?" [Johnson 1995]. This checklist item indicates the inspection is also being used for populating a reuse library.
2. "Are comments used appropriately?" [Johnson 1995]. This checklist item suggests that improving the maintainability of the code is a goal of the inspection.
3. "Are necessary buy-vs.-build decisions included?" [McConnell 1993]. This checklist item for an architecture review indicates the inspection is also being used to ensure proper cost decisions have been made.

Only two of the checklists in this survey actually suggest that a check-mark be placed next to the checklist item [Hollocker 1990, Humphrey 1997]. For the most part, there is little verification that a reviewer actually performed any analysis relating to a checklist item.

2.1 Checklist items for non-code workproducts

Checklists are created for review of workproducts within specific phases of software development. Table 1 lists the surveyed checklists by these phases. Checklists for workproducts other than code (e.g., requirements and design specifications) are typically lists of

Checklist Type		Source
Requirements		[Ackerman 1989], [Basili 1998], [Dunn 1984], [Freedman 1982], [Hollocker 1990], [Humphrey 1989], [Johnson 1995], [McConnell 1993], [NASA 1993], [Porter 1995], [SPAWAR 1997]
Design		[Basili 1998], [Dunn 1984], [Fagan 1976], [Freedman 1982], [Hollocker 1990], [Humphrey 1989], [Humphrey 1995], [Johnson 1995], [Kohli 1975], [Kohli 1976], [Maguire 1993], [McConnell 1993], [NASA 1993], [SPAWAR 1997]
Code	General	[Dunn 1984], [Fagan 1976], [Freedman 1982], [Hollocker 1990], [Humphrey 1989], [Jackson 1994], [Johnson 1995], [Kohli 1976], [McConnell 1993], [Myers 1979]
	Ada	[Humphrey 1997], [SPAWAR 1997]
	Assembler	[Ascoly 1976]
	C, C++	[Baldwin 1992], [Dichter 1992], [Humphrey 1995], [Maguire 1993], [Marick 1995], [NASA 1993], [SPAWAR 1997]
	Cobol	[Ascoly 1976]
	Fortran	[Ascoly 1976], [NASA 1993], [SPAWAR 1997]
	PL/I	[Ascoly 1976]
Testing		[Basili 1998], [Hollocker 1990], [Johnson 1995], [Larson 1975], [Maguire 1993], [McConnell 1993], [NASA 1993], [SPAWAR 1997]
Documentation		[Freedman 1982], [Hollocker 1990], [Humphrey 1989], [SPAWAR 1997]
Process		[Freedman 1982], [McConnell 1993], [SPAWAR 1997]

Table 1. Inspection checklists by type

issues that should be analyzed to ensure consistency, correctness, and completeness. These checklists tend to be more general in nature than code checklists because the workproduct being reviewed is typically written in descriptive text, not a programming language. The following are example requirement and design specification checklist items:

1. "Are all the inputs to the system specified, including their source, accuracy, range of values, and frequency?" [McConnell 1993]
2. "Are all assumptions, limitations, and constraints identified? Are they all acceptable?" [Dunn 1984]
3. "Are all possible states or cases considered?" [SPAWAR 1997]

2.2 Checklist items for generally accepted programming practices

Many checklist items warn of common mistakes or risky programming behavior that is language independent. The following examples are generally accepted programming practice checklist items:

1. "Has each field been initialized properly before it is first used?" [Ascoly 1976]
2. "Are any filenames or pathnames embedded?" [Dichter 1992]
3. "Is the module independent of other modules?" [McConnell 1993]
4. "Are named constants named for the abstract entities they represent rather than the numbers they refer to?" [McConnell 1993]
5. "Does the routine protect itself from bad input data?" [McConnell 1993]
6. "Is it possible for the divisor in a division operation to be zero?" [Myers 1979]
7. "Are there any 'off by one' errors (e.g., one too many or too few iterations)?" [Myers 1979]

2.3 Checklist items for a particular language

Some checklist items warn of highly error-prone areas for particular languages or indicate the likely presence of a defect.

1. Ada: "Does code redefine meaning of any identifier denoting an attribute of the entities declared in the STANDARD package?" [SPAWAR 1997]
2. Assembler: "Have registers been saved on entry and restored on exit? Have stacks been properly initialized?" [Dunn 1984]
3. C: "Are unsigned values tested greater than or equal to zero? *if (myUnsignedVar >= 0)* will always evaluate true." [Baldwin 1992]
4. C: "Are there any common logical errors (== vs. =, misplaced semicolons, missing braces)?" [Dichter 1992]
5. C: "Ensure the { } are proper and matched." [Humphrey 1995]
6. C: "Is the argument to `sizeof` an incorrect type? A common error is using `sizeof(p)` instead of `sizeof(*p)`." [Marick 1995]
7. Fortran: "Determine if the DO variable is expected to be used upon exit of DO loop. The DO variable is not defined at exit." [Ascoly 1976]. Note that this checklist item may be outdated, as newer versions of Fortran (e.g., Fortran 90 and 95) allow the DO index to be used after completion of the DO loop.
8. PL/I: "Are there any mixed-mode computations? An example is the addition of a floating-point variable to an integer variable. Such occurrences are not necessarily errors, but they should be explored carefully to ensure that the language's conversion rules are understood. This is extremely important in a language with complicated conversion rules (e.g., PL/1). For instance, the following PL/1 program fragment:

```
DECLARE A BIT (1);
A=1;
```

leaves A with the bit value 0, not 1." [Myers 1979]

2.4 Checklist items for style issues

A number of general purpose programming style guidelines are available [Kernighan 1978, Maguire 1993, McConnell 1993]. Programming language guidelines are also available for particular languages such as Ada [SPC 1989], C and C++ [Koenig 1989, Holub

1995]. Style guides for writing code for safety critical systems have also been written [NRC 1996, Hatton 1998].These guides can provide many useful suggestions for populating preliminary checklists. These style items may not indicate an actual fault in the program, but could help to avoid fault injection during downstream maintenance. The following are example style guidelines used as checklist items:

1. C: "Are all constant names uppercase?" [NASA 1993].
2. C: "Does the value of the variable never change? *int months_in_year = 12;* should be *const unsigned months_in_year = 12;*" [Baldwin 1992]
3. Ada: "Is each task name a noun phrase describing the function of the task?" [SPAWAR 1997]
4. COBOL: "Are all WORKING-STORAGE items that are used as constants designated as such?" [Ascoly 1976]

2.5 Checklist items tailored to an individual or project

Some checklist items involve mistakes that an individual programmer routinely makes. For example, a novice Ada programmer may find he tends to use integer data types when enumerated data types are more appropriate. A C programmer may repeatedly find herself incorrectly using a "do-while" loop instead of a "while" loop. Sometimes these mistakes derive from misunderstandings involving the project domain rather than programming language constructs. For example, a programmer may have a basic misunderstanding about how scheduling works in a real-time system, and thus repeatedly makes mistakes in code dealing with the scheduler.

Humphrey's [1995] Personal Software Process (PSP) prescribes the use of a checklist tailored to the past mistakes made by an individual. The PSP checklist is used by the developer of the workproduct, not by other reviewers of the workproduct.

Other checklist items point to problems that a large part of the project development team is encountering. For example, a design change may not have been properly communicated to the programmers, resulting in many deficiencies found during inspection. Or, part of the system may be highly complex, and interactions with it may be highly defect prone for all module writers.

2.6 Checklist items that require non-trivial analysis effort

Some checklists have been developed that require the reviewer to take a more active role in finding defects. The reviewer must perform some analysis before addressing the checklist items. The most frequently cited example of this kind of checklist is Parnas and Weiss [1987] on active design reviews. This review process assigns each reviewer a clear area of responsibility and each review has a specific purpose and expertise requirement. Table 2 provides an example of four types of reviews for a module concerned with peripheral devices. Questionnaires are used to allow reviewers to make assertions about design decisions. Figure 1 provides an example part of a questionnaire for the review types found in Table 2. Designers then read the questionnaires and meet with reviewers to resolve issues. These questionnaires are essentially checklists that require the reviewer to perform analysis before being able to address the checklist items. Other examples of checklists that involve active reviewer effort include Knight [1993] and Porter [1995].

Consistency Between Assumptions and Functions

The assumptions should be compared to the function and event descriptions to detect whether a) they are consistent, and b) the assumptions contain enough information to ensure that the functions can be implemented and the events can be detected. If an access function cannot be implemented unless the device has properties that are not in the assumption list, there is a design error, i.e., either a gap in the assumption list or a function that cannot be implemented for some replacement device. The device interface specifications should be reviewed for this criterion by avionics programmer reviewers. After studying the assumptions, the design issues, and the functions and events, they should perform the following reviews:

Review 1

For each of the access functions, the reviewer should answer the following questions:

1. Which assumptions tell you that this function can be implemented as described?
2. Under what conditions may this function be applied? Which assumptions describe those conditions?
3. Is the behavior of this function, i.e., its effects on other functions, described in the assumptions?

Figure 1. Example active design review questionnaire

Type	Description
Assumption Validity	For each device, check that all assumptions made are valid for any device that can reasonably be expected to replace the current device.
Assumption Sufficiency	For each device, check that the assumption lists contain all the assumptions needed by the user programs to make effective and efficient use of the device.
Consistency Between Assumptions and Functions	For each module, compare the assumptions to the function and event descriptions to detect whether a) they are consistent, and b) the assumptions contain enough information to ensure that the functions can be implemented, the events can be detected, and the module can be used as intended.
Access Function Adequacy	For each device, check that user programs can use the device efficiently and meet all requirements by using only the access functions provided in the abstract interface.

Table 2. Example types of active design reviews

3. Checklist items to avoid

This section of the paper discusses several types of checklist items that should be avoided.

3.1 Checks that should be done automatically

Automated tools can be used to check for the potential presence of certain defects. The most well-known tool is *lint* for C [Johnson 1977, Darwin 1988]. A number of *lint*-like tools are available for C and C++ (e.g., ParaSoft's Insure++) and Java [Geis 1998]. The following are example checklist items better left to automated tools:

1. "Are nested IFs indented properly?" [Ascoly 1976]. This check should be performed by a pretty printer.
2. C: "Are functions called with the correct number and type of parameters?" [Dichter 1992]. This defect is checked, for example, by *lint*. The programmer should examine the *lint* output to determine if any flags indicate the presence of a defect. (*lint*'s high signal-to-noise ratio is a separate issue). Alternatively, if programming in ANSI C, prototypes could be used to avoid this defect.
3. "Do actual and formal interface parameter lists match?" [Dunn 1984]. Some languages, such as Ada, provide this check via the compiler while others (e.g., early versions of C) do not.

3.2 Outdated checklist items

Some checklist items are generally outdated in contemporary software development. The following are example checklist items that were useful during their time, but are unneeded and defect-prone today:

1. "Is logic coded in the fewest and most efficient statements?" [Ascoly 1976]. While there are still circumstances where this checklist item is applicable, developers today often do not need to focus on code optimization. This approach led, in part, to the Year 2000 problem.
2. "Where applicable, can the value of a variable go outside its meaningful range? For example, statements assigning a value to the variable PROBABILITY might be checked to ensure that the assigned value will always be positive and not greater than 1.0." [Myers 1979]. This checklist item is outdated for programming languages that provide strong support for data types (e.g., Ada). With strong typing, the compiler would perform the above check via a range constraint. However, the programmer would need to code such a range constraint, which makes for a good checklist item.

3.3 Checklist items better suited as entry/exit criterion

Some checklist items are better used as entry or exit criterion prior to the inspection. It may be more effective to have a single person verify certain properties or actions rather than having an entire inspection team perform this check. The following are example checklist items that should be checked prior to the inspection:

1. "Is the compilation (or assembly) listing free of fault messages?" [Dunn 1984]. This is better checked by the author and verified by the moderator before sending the material to the inspection team.
2. "Is the output from the requirements language processor complete and fault-free?" [Dunn 1984]. Similar to the above checklist item, this is better checked by the author and verified by the moderator.
3. "Do all non-void functions have a return value? ... [This item is] sufficiently checked by *lint*." [Dichter 1992]. The programmer should run *lint* and examine the output to ensure that this type of defect does not exist. Review of *lint* output should be an entry criterion to the code inspection.

3.4 Checklist items that are too general

Some checklist items are too general to be of much help.

1. "Are software requirements clear and ambiguous?" [SPAWAR 1997]
2. "The code is maintainable." [Hollocker 1990]
3. "Are there any 'go to' statements?" [SPAWAR 1997]
4. "Is the definition of success included? Of failure?" [McConnell 1993]. This checklist item is intended for requirements specification. While it can be worthwhile to think about this issue in the context of a general purpose review, it is too general for individual reviewers to use as a checklist item for an inspection.
5. "Are the goals of the system defined?" [Porter 1995] and [SPAWAR 1997]. Similar to the above checklist item, this is a very general checklist item more suitable for a requirements walkthrough than an inspection.

4. Summary and observations

Table 3 cites each of the checklists surveyed and characterizes them in terms of the number of checklists items, the workproduct the checklist is intended for, and provides a brief comment relating to the checklist. Some of these checklists are more suited towards general purpose walkthroughs, while others are intended for the person creating the product (e.g., desk-checking).

In examining these checklists, several observations can be made. None of the checklists should be used "as is." Project staff should invest the effort necessary to analyze the types of errors being made and develop tailored checklists for helping reviewers increase their defect detection effectiveness. These checklists can, however, provide helpful ideas for populating a project checklist.

Many of the checklists are too lengthy to be used by a reviewer as part of an inspection process. Half of the surveyed checklists included twenty or more checklists items. Checklists should generally be limited to a page in length.

Feedback on checklist effectiveness can help to determine if the reviewers are using the checklist or how well it is working. Several review methods encourage this feedback [Gilb 1993, Humphrey 1995, Parnas 1987], but there have been few reports published describing checklist experiences with these methods.

The observations noted above have been widely discussed in the software inspection literature. However, there has been little industrial experimentation investigating methods for improving checklist effectiveness. The inspection checklist appears to be a fertile area for future software engineering research.

Source	Items	Intended Workproduct for Checklist	Comments
[Ackerman 1989]	23	Requirements	Checklist items are divided into 3 categories: completeness, ambiguity, and consistency. Items consist of general questions such as "What is the total input space?" and "What are the types of runs?"
[Ascoly 1976]	80	Code (Cobol)	Quite a large collection of checklist items for contemporary programming languages and environments of the 1970's. Many of the Cobol items pertain to the format of comments. Document is a bit difficult to obtain.
	23	Code (Fortran)	
	93	Code (PL/I)	
	20	Code (Assembler)	
[Baldwin 1992]	72	Code (C++)	This document is part inspection checklist, part style guide, and part guide to defensive programming. It has many examples of defect-prone code fragments accompanied by an explanation and suggested code fragments.
[Basili 1998]	6	Requirements	A set of lab materials intended to be used by customers of a product as part of their assessment of a document, not those who create the document.
	6	Design	
	5	Testing	
[Dichter 1992]	17	Code (C)	Four items are identified as being sufficiently checked by *lint*. Others include defect-prone aspects of C and style issues.
[Dunn 1984]	20	Requirements	While some of the items are too general for an inspection checklist, others are more specific ("Are imported data tested for validity?").
	19	Design (Top-Level)	
	22	Design (Detailed)	
	33	Code (General)	
[Fagan 1976]	19	Design	All of these checklist items are derived from [Kohli 1976].
	12	Code (Assembly and General)	
[Freedman 1982]	11	Requirements	Most of these checklist items are too general to be used for an inspection and are more suitable for general purpose walkthroughs. For example, "What have you forgotten?", "What has been done wrong?", and "Did you dot the i's?" are design checklist items.
	10	Design (Preliminary)	
	24	Design (Design Misfit)	
	14	Code (General)	
	20	Code (General: Side Effects)	
	9	Code (General: Data Side Effects)	
	11	Documentation (Side Effects)	
	80	Documentation	
	27	Process (Inspection Recorders)	
	6	Misc. (Side Effects)	
[Hollocker 1990]	19	Requirements	This is a good source for a wide range of checklist items. Many items are general but can easily be adapted for specific use.
	17	Design (Document)	
	22	Design (Architecture)	
	22	Design (Detailed)	
	18	Code (General)	
	23	Testing (Test Plan)	
	12	Testing (Test Specifications)	
	18	Testing (Test Reports and Records)	
	14	Documentation	
[Humphrey 1989]	11	Requirements	The checklists in this book are slightly edited versions of those found in [Freedman 1982].
	24	Design	
	14	Code (General)	
	88	Documentation	

Table 3. Summary of checklists

Source	Items	Intended Workproduct for Checklist	Comments
[Humphrey 1995]	26	Design	The checklists were created by Humphrey based on personal design and code defect analysis. This type of checklist is to be used by developers in reviewing their workproducts; they are not intended to be used by other reviewers. The C++ checklist items are a mix of general items ("Ensure that the code conforms to the coding standards") and defect-prone aspects of C ("Verify the proper use of ==").
	21	Code (C++)	
[Humphrey 1997]	24	Code (Ada)	The C++ checklist is re-published from [Humphrey 1995]. See [Humphrey 1995] for a description of PSP checklists.
	21	Code (C++)	
[Jackson 1994]	23	Code (General)	A checklist for review of a module's design and implementation.
[Johnson 1995]	9	Requirements	Most of these checklist items are too general to be used for an inspection ("Do any of the requirements conflict with one another" and "Are comments used appropriately?").
	10	Design	
	13	Code (General)	
	8	Testing	
[Kohli 1975]	35	Design (High Level)	Mostly outdated. Geared towards IBM mainframe implementations.
[Kohli 1976]	65	Design (Detailed)	Mostly outdated. Geared towards IBM mainframe implementations.
	12	Code (General)	
[Larson 1975]	13	Testing (Test Plan)	This checklist is still applicable after 24 years.
[Maguire 1993]	8	Design	These checklists are intended more for developers than for reviewers. However, many of the checklist items could be easily adapted for an inspection checklist.
	9	Code (C)	
	15	Testing	
[Marick 1995]	47	Code (C)	Includes narrative to explain defect-prone aspects of C, and includes bad/good code fragments to demonstrate many of the checklist items.
[McConnell 1993]	27	Requirements	These checklists cover the range of reviews for a software development project and are primarily intended for developers rather than for reviewers. However, many of the checklist items could be easily adapted to an inspection checklist. This is the most extensive set of checklists included in this survey.
	26	Design (Architecture)	
	14	Code (General: Constructing a Routine)	
	22	Code (General: High-Quality Routines)	
	9	Code (General: Module Quality)	
	21	Design (High-Level Design)	
	11	Code (General: Data Creation)	
	31	Code (General: Naming Data)	
	14	Code (General: Considerations in Using Data)	
	38	Code (General: Fundamental Data)	
	8	Code (General: Organizing Straight-Line Code)	
	16	Code (General: Conditionals)	
	19	Code (General: Loops)	
	12	Code (General: Unusual Control Structures)	
	10	Code (General: Control-Structure Issues)	
	27	Code (General: Layout)	
	27	Code (General: Self-Documenting Code)	
	31	Code (General: Commenting Techniques)	
	14	Testing (Test Cases)	
	34	Testing (Debugging)	
	6	Testing (Incremental Integration Strategy)	
	11	Process (Evolutionary Delivery)	
	9	Process (Making Changes)	
	12	Process (Configuration Management)	
	8	Process (A Quality-Assurance Program)	
	11	Process (Effective Inspections)	

Table 3. Summary of checklists (continued)

Source	Items	Intended Workproduct for Checklist	Comments
[Myers 1979]	11	Code (General: Data Reference)	The checklist items are generally programming language independent.
	6	Code (General: Data Declaration)	
	10	Code (General: Computation)	
	8	Code (General: Comparison)	
	8	Code (General: Control-Flow)	
	11	Code (General: Interface)	
	8	Code (General: Input/Output)	
	5	Code (General: Other Checks)	
[Porter 1995]	0	Ad Hoc	The requirements checklist with 29 items is primarily derived from other published industry checklists. The checklist with 18 items was used in support of scenario-based checklist research, and focuses on data type consistency, incorrect functionality, and ambiguities or missing functionality.
	29	Requirements	
	18	Requirements (Scenario)	
[Yourdon 1989]	0	None	No checklists, but provides general guidelines for requirements, design, and code walkthroughs.
[NASA 1993]	34	Requirements (Functional) (JPL)	Many of the checklists are derived from an earlier NASA JPL document.
	73	Requirements (Software) (JPL)	
	41	Design (Architecture)	
	47	Design (Architecture Design) (JPL)	
	64	Design (Functional) (JPL)	
	37	Design (Detailed)	
	56	Design (Detailed) (JPL)	
	55	Code (C)	
	49	Code (C) (JPL)	
	90	Code (Fortran) (JPL)	
	40	Testing (Test Plan) (JPL)	
	31	Testing (Test Procedure) (JPL)	
[SPAWAR 1997]	31	Requirements (System)	Many of the checklist items are derived from [NASA 1993].
	25	Requirements (Software)	
	25	Design (Software Preliminary)	
	23	Design (Software Detailed)	
	113	Code (Ada)	
	50	Code (C)	
	80	Code (Fortran)	
	31	Testing (Test Plan)	
	30	Testing (Test Cases and Procedures)	
	20	Documentation (User)	
	15	Process (Software Development Plan)	

Table 3. Summary of checklists (continued)

Bibliography

[Ackerman 1989]
Ackerman, A. Frank, Lynne S. Buchwald, and Frank H. Lewski. Software Inspections: An Effective Verification Process. *IEEE Software*, Vol. 6, No. 3, May 1989, pp. 31-36.

[Ascoly 1976]
Ascoly, Joseph, Michael J. Cafferty, Stephen J. Gruen, and O. Robert Kohli. Code Inspection Specification. IBM Corp., Kingston, NY, Technical Report TR21.630, 1976.

[Baldwin 1992]
Baldwin, John T. An Abbreviated C++ Code Inspection Checklist. Available on-line at *http://www.ics.hawaii.edu/~johnson/FTR/Bib/Baldwin92.html*, Oct. 27, 1992.

[Basili 1998]
Basili, Victor, Scott Green, Oliver Laitenberger, Filippo Lanubile, Forrest Shiull, Sivert Sørumgård, and Marvin Zelkowitz. Lab Package for the Empirical Investigation of Perspective-Based Reading. Available on-line at *http://www.cs.umd.edu/projects/SoftEng/ESEG/manual/pbr_package/manual.html*.

[Darwin 1988]
Darwin, Ian F. *Checking C Programs with Lint*. O'Reilly, 1988.

[Dichter 1992]
Dichter, Carl R. Two Sets of Eyes: How Code Inspections Improve Software Quality and Save Money. *Unix Review,* Vol. 10, No. 2, Jan. 1992, pp. 18–23.

[Dunn 1984]
Dunn, Robert H. *Software Defect Removal.* McGraw-Hill Book Company, 1984.

[Fagan 1976]
Fagan, Michael E. Design and Code Inspections to Reduce Errors in Program Development. *IBM Systems Journal,* Vol. 15, No. 3, 1976, pp. 182-211.

[Freedman 1982]
Freedman, Daniel P. and Gerald M. Weinberg. *Handbook of Walkthroughs, Inspections, and Technical Reviews: Evaluating Programs, Projects, and Products.* 3rd ed., Boston, Little, Brown & Co., M.A., 1982.

[Geis 1998]
Geis, Jennifer M. JavaWizard: Investigating Defect Detection and Analysis. Master's Thesis, Information and Computer Sciences, University of Hawaii, May 1998.

[Gilb 1993]
Gilb, Tom and Dorothy Graham. *Software Inspection.* Addison-Wesley, Reading, MA, 1993.

[Hatton 1998]
Hatton, Les. *Safer C: Developing Software for High-Integrity and Safety-Critical Systems.* McGraw-Hill, 1998.

[Hollocker 1990]
Hollocker, Charles P. *Software Reviews and Audits Handbook.* John Wiley & Sons, N.Y., 1990.

[Holub 1995]
Holub, Allen I. *Enough Rope to Shoot Yourself in the Foot: Rules for C and C++ Programming.* McGraw-Hill, 1995.

[Humphrey 1989]
Humphrey, Watts S. *Managing the Software Process.* Reading, MA: Addison-Wesley Publishing Co. 1989.

[Humphrey 1995]
Humphrey, Watts S. *A Discipline for Software Engineering.* Reading, MA: Addison-Wesley Publishing Company, 1995.

[Humphrey 1997]
Humphrey, Watts S. *Introduction to the Personal Software Process.* Reading, MA: Addison-Wesley Publishing Company, 1997.

[Jackson 1994]
Jackson, Ann and Daniel Hoffman. Inspecting Module Interface Specifications. *Software Testing, Verification, & Reliability,* Vol. 4, No. 2, Jun. 1994, pp. 101–117.

[Johnson 1977]
Johnson, S.C. *Lint: A C Program Checker.* Technical Memorandum 77-1273-14, Sep. 16, 1977.

[Johnson 1995]
Johnson, Jay, Rod Skoglund, and Joe Wisniewski. *Program Smarter, Not Harder: Get Mission-Critical Projects Right the First Time.* McGraw-Hill, Inc. 1995.

[Kernighan 1978]
Kernighan, Brian W. and P.J. Plauger. *Elements of Programming Style.* 2nd edition, New York: McGraw-Hill, 1978.

[Knight 1993]
Knight, John C. and E. Ann Myers. An Improved Inspection Technique. *Communications of the ACM,* Vol. 36, No. 11, Nov. 1993, pp. 51-61.

[Koenig 1989]
Koenig, Andrew R. *C Traps and Pitfalls.* MA: Addison-Wesley Publishing Company, 1989.

[Kohli 1975]
Kohli, O. Robert. High-Level Design Inspection Specification. Tech. Report TR 21.601, IBM Corp., Kingston, N.Y., Jul. 21, 1975.

[Kohli 1976]
Kohli, O. Robert and Ronald A. Radice. Low-Level Design Inspection Specification. Tech. Report TR 21.629, IBM Corp., Kingston, N.Y., Apr. 1976.

[Larson 1975]
Larson, Rodney R. Test Plan and Test Case Inspection Specification. Tech. Report TR21.585, IBM Corp., Kingston, N.Y., Apr. 4, 1975.

[Maguire 1993]
Maguire, Steve. *Writing Solid Code.* Microsoft Press, 1993.

[Marick 1995]
Marick, Brian. *The Craft of Software Testing.* Prentice Hall, 1995.

[McConnell 1993]
McConnell, Steve. *Code Complete.* Microsoft Press, 1993.

[Myers 1979]
Myers, Glenford J. *The Art of Software Testing.* Wiley, 1979.

[NASA 1993]
National Aeronautics and Space Administration. Software Formal Inspections Standard, NASA-STD-2202-9, Apr. 1993.

[NRC 1996]
U.S. Nuclear Regulatory Commission. *Review Guidelines on Software Languages for Use in Nuclear Power Plant Safety Systems.* NUREG/CR-6463, Jun. 1996.

[Parnas 1987]
Parnas, David L. and David M. Weiss. Active Design Reviews: Principles and Practices. *Journal of Systems and Software,* No. 7, 1987, pp. 259-265.

[Porter 1994]
Porter, Adam A. and Lawrence G. Votta. An Experiment to Assess Different Defect Detection Methods for Software Requirements Inspections. In *Sixteenth International Conference on Software Engineering,* Sorrento, Italy, May 1994, pp. 103-112.

[Porter 1995]
Porter, Adam A., Lawrence G. Votta, Jr., and Victor R. Basili. Comparing Detection Methods for Software Requirements Inspections: A Replicated Experiment. *IEEE Transactions on Software Engineering,* Vol. 21, No. 6, Jun. 1995, pp. 563-575.

[SPAWAR 1997]
Space and Naval Warfare Systems Center. *Formal Inspection Process, Version 2.2.* Sep. 29, 1997. Available on-line at http://sepo.nosc.mil/FIProc.zip.

[SPC 1989]
Software Productivity Consortium. *Ada Quality and Style: Guidelines for Professional Programmers.* Van Nostrand Reinhold, New York, 1989.

[Wheeler 1996]
Wheeler, David A., Bill Brykczynski, and Reginald Meeson, Jr., editors. *Software Inspection: An Industry Best Practice.* IEEE Computer Society Press, 1996.

[Yourdon 1989]
Yourdon, Edward. *Structured Walkthroughs.* 4th Edition. Prentice-Hall, Englewood Cliffs, N.J., 1989.