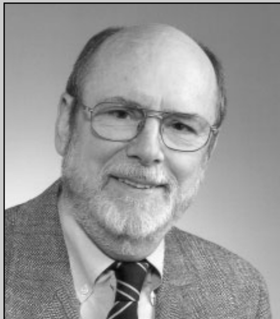
Put Better Programs into Production in Less Time with Code Reviews: What They Are, How to Conduct Them, and Why

David F. Jenkins



David Jenkins became an SAP consultant in 1995 after 29 years as a Consulting Systems Representative at IBM. He currently develops and teaches classes in beginning and advanced ABAP programming, ABAP performance and tuning, and Java, and consults with clients on all aspects of ABAP development.

(complete bio appears on page 24)

> *Any […] fool can write code that a computer can understand, the trick is to write code that humans can understand.*
> *– Martin Fowler*

One of the major strengths of SAP product offerings is the availability of turnkey application solutions "out-of-the-box." Nevertheless, the reality is that many of those solutions require additional functionality, which is often provided through the use of custom programming — in SAP environments, this is most often done in ABAP and, more recently, also in Java. It behooves us, then, as developers of SAP software, to find and adopt ways of reducing both the time and cost required to produce such software. One method of achieving such reductions is through the use of *code reviews*.

In this article, I'll provide a brief introduction to the concept of code reviews — what they are, how they're conducted, and what benefits you can expect to receive from their use — along with some tips for conducting code reviews in your own development environment, including an extensive sample checklist you can use as a template. Whether you are a developer or a manager, you will find some information in this article that you can use to enhance the delivery of error-free programs on time and within cost constraints.

Code Reviews: An Overview

There is no substitute for a close examination of others' code — no code should be released into production until someone other than the

author has read and understood it. The generic term "code review" is used to describe the process of having a third party evaluate your program at various points in the development cycle, prior to placing it into productive use. The objectives of such a review include:

- Defect discovery
- Conformance to specifications
- Identification of opportunities for process improvement
- Knowledge transfer
- Exploration of alternative strategies
- Assessment of follow-up on previously identified problems
- Identification of risks

Code reviews can take any number of forms, and the terms used to describe those forms are numerous. However, the most commonly used names describing types of code reviews are *inspections*, *team reviews*, and *walkthroughs*, which I will briefly introduce here.

Inspections

According to Wiegers,[1] an inspection is a formal process for examining work product to assure that it satisfies its functional specifications and customer needs, and that all pertinent standards, regulations, procedures, etc., have been followed. Major outputs of the inspection process are metrics intended to quantify defects discovered during inspection.

Inspections include well-defined stages, such as planning, overview, preparation, meetings, and follow-up. The inspection meeting usually comprises the culmination of all inspection activities (other than follow-up) and generally follows a formal procedure, with attendees taking on formal roles. These roles often include:

- **Author:** Authors typically do not serve as meeting moderator, reader, or recorder (see below).
- **Moderator:** Moderators work with authors to plan the inspection and lead the rest of the inspection team.
- **Reader:** Readers present the material undergoing inspection at the formal inspection meeting.
- **Recorder:** Recorders log comments, action items, recommendations, etc., that arise during the inspection meeting.
- **Other:** Others may be invited to participate in the inspection meeting, but are not assigned specific duties.

Code inspections are generally recognized as being the most productive for uncovering defects. Because of the formality of the inspection process and the possible complexity of problem solutions, I have observed that inspection meetings are often limited to defect discovery, and that the identification of problem solutions is accomplished outside of the inspection process.

Team Reviews

Team reviews involve small groups that review a product to see that it is ready for productive use and that it satisfies specifications. The essential difference between an inspection and a team review is often defined by the level of formality with which the review is conducted.

In an inspection, the leader of the inspecting group is typically *not* the author of the target program, and meeting participants may be assigned to critique certain parts of the program being scrutinized. A team review, on the other hand, is often led by the program author, and all participants may be asked to report on any issues they have uncovered during independent reviews of the entire program.

[1] In addition to my own observations and experiences, this article makes use of information gathered from a number of sources. Publications of authors mentioned throughout this article are listed in the "References" sidebar on page 23.

 www.SAPpro.com ©2003 SAP Professional Journal. Reproduction prohibited. All rights reserved.

Team reviews tend to be less formal than inspections, and hence less productive. Van Veenendaal reports that team reviews may uncover as few as two-thirds the number of errors that are revealed by inspections. Team reviews may be more suitable when the product itself does not warrant the formality of a full-fledged inspection, or formal inspections don't conveniently mesh with the organizational culture. It is not uncommon for organizations just beginning to institute review processes to take a "toe-in-the-water" approach and to start with team reviews as an introductory step.

Walkthroughs

Walkthroughs are conducted even less formally than team reviews. At meetings (often moderated by the program author), the author may describe both the business or technical problem the program or system was defined to solve and the program itself. Comments from attendees are solicited, and the thrust of the walkthrough process may range from a solicitation for ideas on how to solve a particular problem, to provision for a vehicle to educate team members about a project or product.

One of the biggest drawbacks to the walkthrough approach is that because they are conducted without a formal procedure and by various individuals, they will probably lack consistency across the organization — they may be casual in one department and disciplined in another, for instance, leading to varying levels of detail. On the other hand, they may provide a useful vehicle for knowledge transfer — because a walkthrough essentially involves a presenter and an audience, more folks can be in attendance, and it may be possible to use the meeting to educate others on the product under review.

Additional Types of Reviews

There are yet other levels of granularity to reviews: in *pair programming*, two programmers may work on a single program, at a single workstation, and hence the efforts of each are under continuous review by the other.[2] To the extent that two programmers work on any given product, that product will undergo continuous review.

Desk checks and *passarounds* involve the distribution of code and supporting materials to one or more individuals to solicit comments regarding all aspects of the product. Desk-checking by peers is one of the least expensive forms of review, since it involves only one reviewer's time. It is most applicable when you are working under severe time constraints, the risk is low, and the peers doing the desk checks have the level of skill necessary to identify defects without additional input from others.

In general, I'll use the term "code review" in this article to refer to any of a number of activities, all broadly intended to subject programs to some sort of formal review process by third parties in order to meet the objectives enumerated earlier (identify defects, risks, non-conformance, inefficiencies, and alternative strategies; enable knowledge transfer; and follow up on previous issues).

Why Conduct Code Reviews?

Code reviews are often opposed because of the mistaken belief that they will slow development. As Wiegers points out, it's not reviews that slow projects — it's defects. Earlier defect detection and remediation will reduce the amount of rework required on the project, and shifting the defect detection to earlier stages of product development can provide a huge payoff. A number of studies (Jones, Cooper and Mullen, Haley, and others) have found that rework can account for as much as 60% of total project development effort. From my own experience, by far the majority of post-implementation

[2] Pair programming is an integral component of a development approach called *extreme programming* (commonly referred to as "XP"). XP is extreme in the sense that it takes 12 well-known software development "best practices" to logical extremes. A complete discussion of XP is beyond the scope of this article. See the "References" sidebar on page 23 for a pointer to an excellent source of more information on this subject by Kent Beck.

work is concerned with defect remediation, as opposed to implementation of additional functionality. The costs of major SAP implementations are often measured in terms of tens, and as much as hundreds, of millions of dollars — any reduction in the rework portion of the associated development effort can have a significant impact on the total project cost. Here are some benefits that have accrued from the use of inspections:

- Gilb and Graham report that the cost of maintaining a portfolio of 400 inspected programs was one-tenth the cost (per line of code) for 400 similar uninspected programs.
- At AT&T Bell Labs, inspections reduced the cost of error remediation by ten-fold (Humphrey).
- IBM saved 20 hours of testing and 82 hours of rework per hour of inspection (Holland).
- Bell Northern Labs found that detecting defects through inspection was two to four times faster than detecting them through testing.

  Other quantified benefits include the following:

- Hewlett-Packard reported an ROI of *1,000%* on their inspection process, and reduced time-to-market in some cases by almost two months.
- During five years of code inspections, Primark Investment Management saw a five-fold decrease in product errors reported per customer per year.
- Litton Data Systems found that a 3% investment in inspections (as a percent of total project effort) reduced the number of errors found during integration and testing phases by 30%.

These are impressive numbers — the leverage that code reviews provide to lower total development and installation costs is obviously significant.

One might argue, however, that ABAP is a “high-level” language, and hence less complex, and therefore needs less reviewing to assure product completeness and accuracy. (Capers Jones has published a table that provides a comparison of over 500 languages on the basis of *language level*. The sidebar on the next page shows a sampling of levels for various languages — note that ABAP enjoys a relatively higher level than the other languages shown.) I would argue, however, that the higher the level of the language, the more significant the consequences may be of a single defect. It is even more important in the case of a language such as ABAP, then, to assure that installed code is as free of defects as possible.

On the other hand, Java does not share the high language level of ABAP. To that extent, it will take significantly more statements to code equivalent functionality, and hence the coding may provide increased opportunity for defects due to the increased number of lines of code. Therefore in this case, too, rigorous review will be required.

So, regardless of language or language level, it is important to apply a review protocol to your custom development.

As noted earlier, inspections are typified by a higher level of formality than other modes of code review and hence they afford the greatest possibility for gathering and analyzing data regarding review effects on the total project. Nevertheless, other review types also provide potential for significant, albeit perhaps somewhat less striking, results.

Given that reviews provide the potential for remarkable development cost reductions, how are they implemented? Before embarking on a code review, there are two aspects that must be addressed: *what* will be reviewed, and *how* the reviews will be conducted. In the next two sections, I’ll briefly cover each of these subjects.

The Review Process: What Will Be Reviewed

The process of program development entails much more than simply writing, testing, and installing a set of code. For that reason, code reviews really should

 ©2003 SAP Professional Journal. Reproduction prohibited. All rights reserved.

Representative Language Levels

The numeric *language level* of a programming language may provide a convenient shortcut for enabling a conversion of program size (in terms of *logical source statements*) from one language to another. For example, if an application were to require 1,000 non-comment ANSI COBOL statements (level 3.5), then it might take only 500 statements in a level-7 language (such as OS/2 REXX). As the language level increases, it takes fewer source statements to code a function point (FP).* The following table provides a language level comparison for a number of popular languages:

Language	Level	Average Number of Source Statements per FP
ABAP	20.0	16
ANSI COBOL	3.5	91
C++	6.0	53
Delphi	11.0	29
Eiffel	15.0	21
HTML 3	22.0	15
Java	6.0	53
Machine language	0.5	640
Pascal	3.5	91
Perl	15.0	21
SAS	10.0	32
SQL	25.0	13
Visual Basic 5	11.0	29

It would be a mistake, however, to correlate language level and total development productivity, since coding may occupy as little as 30% of total activity on a given project. For instance, if a program is written in a language that's twice the level of a similar program written using another language, one could reasonably expect to find the programming effort reduced by 50%. If the coding effort were only 30% of the total development activity, however, then the increase in total development productivity might be only 15%.

* See Jones for a complete description of program function point analysis. A consistent and definitive relationship of logical source statements to function points has not been reported. The values shown in the table above were computed by "backfiring" — identifying the approximate number of logical source statements correlated to a single function point. Bear in mind that local development practices and the ways different languages deliver functionality make the ensuing results problematic. Nevertheless, as a *rough* measure of programming language "height," these results (published by Software Productivity Research) can provide a useful guideline.

focus on all significant elements that affect the outcome of the code preparation process, as well as the code itself. In no particular order, here is a partial list of such elements, extracted from the IEEE "Standard for Software Reviews":

- Program documentation
- Maintenance documentation
- Database table design and index specification
- Source code
- User interfaces
- Business rules and business process models
- Project definition, including scope, schedules, task lists, etc.
- Test plans
- User documentation
- User training materials and plans
- Development standards

While all of these are important aspects of the development process, and should be reviewed throughout that process, in this article I focus on reviews of SAP development code itself (primarily ABAP, and increasingly Java[3]) and the elements of code reviews that pertain most directly to the code. It is my experience that standards, performance, and the proper utilization of the vast set of standard SAP functionality that is available to developers are the primary issues that arise during implementations — these are exactly the types of issues that code reviews address. For instance, while user training is certainly an important aspect of a project implementation, its significance in a code review will be limited only to the ways that training impinges on coding.

[3] With the impending release of SAP Web Application Server 6.30, Java will assume an even more important role in the development of custom code to enhance SAP systems. The points made in this article should apply almost equally to both ABAP and Java development efforts.

The Review Process: How the Reviews Will Be Conducted

An effective review requires, to varying degrees of formality, that a review process be put in place that consists of, at minimum:

- Preparation
- The review meeting
- Documentation
- Rework and follow-up

Preparation

Code reviews are typically triggered when an author announces that he or she has some work product ready for inspection. Reviews can also be triggered by the achievement of a project development milestone ("Ready or not — here we come!"). It's important, however, not to see code reviews as milestone events in and of themselves; the danger here is that they become something to be "gotten through," rather than a development activity that aids in the achievement of a milestone such as phase completion, start of integration testing, etc. Once you have decided at an overview level whether the program is ready to be reviewed, there are four tasks to be accomplished to prepare for an effective review:

1. Assign a moderator.
2. Assemble a review team.
3. Prepare the review team.
4. Schedule relevant events.

Step 1: Assign a Moderator

The moderator will lead the review process and be responsible for all of its activities and elements.

 ©2003 SAP Professional Journal. Reproduction prohibited. All rights reserved.

So, who is likely to make a good moderator for the review process? Those who:

- Are good at planning and follow-through
- Have strong meeting facilitation skills
- Will not tend to dominate the meeting
- Are respected by other participants
- Can be fair and impartial
- Have the requisite technical and application background to do all of the above for the particular project being reviewed

Who should *not* be considered for the moderator role? Those who:

- Lack the skills/characteristics mentioned above
- Are in the direct management chain of the program author

Step 2: Assemble a Review Team

The review team will examine the work product prior to the review meeting and will participate in the review meeting. It will be the team's consensus that a particular product is ready for production, or not. There are two criteria for team selection that are paramount:

- Team members must be technically competent in the area to be reviewed. This implies that review team makeup in your organization may vary from product to product.
- Team members must have the professional respect and trust of the author.

When assembling such a team, the subject of management participation will often arise. It's generally accepted that managers should *not* review deliverables created by those who report to them, since the manager could tend to evaluate the author, rather than the author's work product. It's also possible that other participants may be hesitant to identify defects if it is suspected that the manager is using review results to feed the employee evaluation process. In one memorable situation where I was an observer, participants failed to identify a single defect in the work product due to the manager/employee dynamic at work. The product was subsequently placed into production, where it failed absolutely from its first use.

The subject of team selection is rife with organizational behavior ramifications, and far too complex for a detailed discussion here. The point is that management participation should be addressed as an issue when planning the review, and a decision for such participation should be predicated on the level of trust between author and manager, the technical competence of the manager, and other relevant criteria, such as a proven track record of non-punitive participation in previous inspections. As Wiegers points out, urging a manager to ignore employee capability when observing defect identification is a little "like a judge telling a jury to disregard something a witness just said." To preclude such possibilities, the safest course to follow is to exclude management participation in the review whenever possible.

Step 3: Prepare the Review Team

Decide *what* is to be inspected and assemble a package of all materials that will be required; then disseminate the materials to the review team members. At a minimum, the materials should include:

- *The deliverable to be inspected*, or a pointer to the deliverable to be inspected.
- *Testing outputs*, depending on organizational requirements and the type of problem a particular program is designed to solve. For instance, a testing approach based on induction may be required to show that a program will work under all different input scenarios. That is, if the testing were to show that the program logic was correct for zero inputs, that it continued to be correct for a single input, and that given n inputs it is correct

Figure 1 *A Simple Typographical Error Report Form*

Typographical Errors

Reviewer: ____________________

Work Product: ____________________

Date: ____________________

Page/Line	Description
______	____________________
______	____________________
______	____________________
______	____________________
______	____________________
______	____________________

for *n+1* inputs, then reviewers could reasonably assume that the program will work correctly for *x* inputs (*x = 0,1,2...n*).

- *Pertinent supporting documentation*, such as applicable standards and definitions, specifications, etc.
- *Forms*, which can be useful in facilitating the review process at all stages, including aiding in the preparation and packaging of review documentation of the entire process, and forming the basis for gathering metrics on the effectiveness of reviews in your organization. Such forms can range from the simple to the more complex. For example, **Figure 1** represents a simple form that can be used to record minor program problems, such as typos. **Figure 2**, on the other hand, shows a more complex form that can be used to record a summary of code review results.

  The summary report, by the way, can be used to record more detailed information regarding the level of effort required to support the review process. For instance, it may be useful to gather the labor hours expended by individual team members, the level of effort involved in producing the work product, and the number and severity level of all defects detected during the review.
- *Standardized checklists*, which aid in making sure that reviewers focus on important areas liable to result in defect detection. Given that the checklists are fairly exhaustive (that is, they identify *most* of such areas), they also provide a framework for the gathering of information used in evaluating review success. Here are a few of the sections a code review checklist might include:
  - Comments
  - Documentation
  - Naming (adherence to standards, clarity, relevance, etc.)
  - Coding
  - Design
  - Object-oriented design

 ©2003 SAP Professional Journal. Reproduction prohibited. All rights reserved.

Figure 2 *A More Complex Code Review Summary Report*

Code Review Summary Report

Project: ____________________

Element reviewed: ____________________

Review date: __________ Total meeting time: __________

LOC reviewed: __________

Team members: Prep time:

____________________ __________

____________________ __________

____________________ __________

____________________ __________

____________________ __________

____________________ __________

Appraisal summary

Accepted as is: __________

Conditionally accepted (see below): __________

Not accepted – rework necessary: __________

Not accepted – inspection complete: __________

Terms of conditional acceptance (attach additional pages if required):

____

____

____

Review team moderator (signature):____________________

- Code layout and sequencing
- Minimization of code duplication
- Support for testing
- Use of standard API functionality

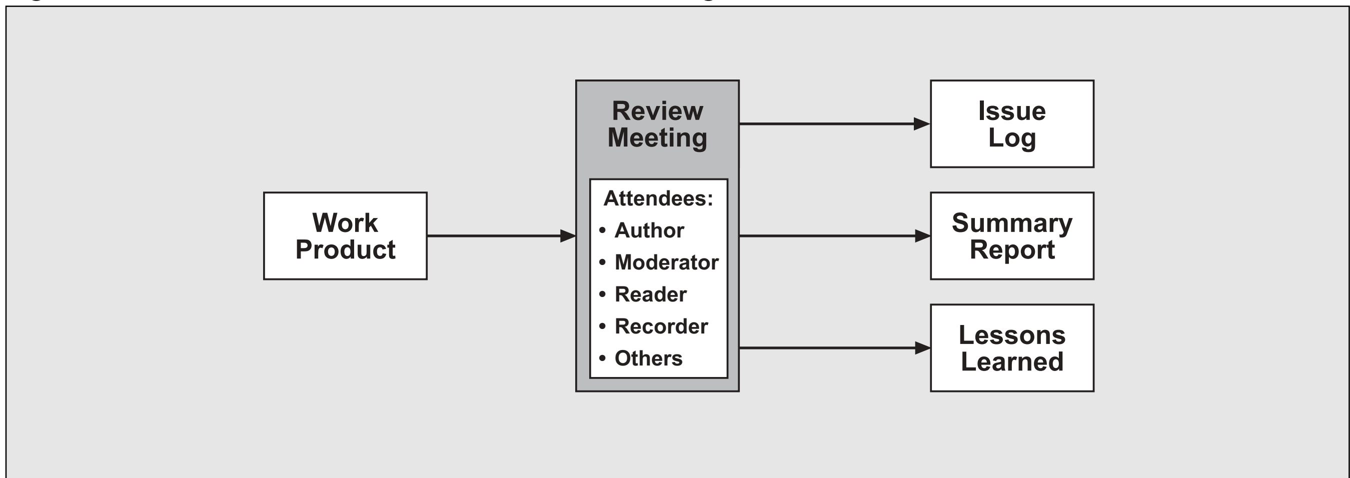
Later in the article, I'll provide more detailed recommendations regarding checklists that are geared to ABAP development.

Step 4: Schedule Relevant Events

Relevant review events include a preliminary overview of review package items by participants, the review meeting itself, follow-up meetings, and so on.

With these four tasks complete, attendees will be well prepared to enter the next phase of the code review process — the review meeting itself.

Figure 3 Review Meeting Elements



The Review Meeting

The review meeting is the activity where team members meet to discuss work product defects discovered independently, prior to the meeting. The primary purpose of the meeting is to make an overall assessment, by consensus, as to the suitability of the product for production. There are additional purposes that the review meeting serves, not the least of which are knowledge transfer and education. Through the use of group discussions, other team members can be made aware of current projects (both requirements and solutions), special techniques, and organizational standards that may pertain to development.

Figure 3 represents an overall view of the review meeting process and its major elements.

It cannot be overly stressed that it is the moderator who *leads the meeting*. She or he will guide the meeting process, make sure that attendees participate effectively, and, above all, lead the meeting to its proper conclusion regarding the quality of the reviewed product.

Most meeting problems arise out of the shortcomings of the moderator's performance (Wiegers). In particular, meetings that moderators allow to become merely unstructured discussions will almost certainly fail to meet the meeting's objectives — all of us have seen meetings that have lost their way and failed to produce meaningful outcomes. For that reason, meeting activities should be well planned in advance — a moderator's checklist is certainly apropos and helpful.

The sidebar on the next page shows just a sample of items that can be included on the moderator's checklist. The point is that the moderator should not enter the meeting "cold." Code review meetings work best when they follow certain choreographed steps — the moderator's checklist will help ensure that those steps are followed. Here is a list of the general steps to be followed:

1. Start the meeting.
2. Present the material.
3. Identify defects.
4. Evaluate the results.

Each of these steps is described briefly in the following sections.

Step 1: Start the Meeting

At the start of the meeting, the moderator must introduce attendees who do not already know each other. More important, the moderator must assess the

 ©2003 SAP Professional Journal. Reproduction prohibited. All rights reserved.

An Example Moderator's Checklist

☑ Things to be brought to the meeting, such as:

- Blank summary report forms
- Issue logs
- Defect checklist

☑ Things to be done at the start of the meeting, such as:

- Introductions of attendees
- Statement of objectives
- Meeting rules (only one person speaks at a time, for example)
- Brief introduction of the work product

☑ Things to be done at the end of the meeting:

- Record the team's consensus in summary form
- If remediation is required, determine and record responsible parties for each follow-up activity
- Schedule follow-up activities

☑ Things to be done following required rework:

- Record final results on review summary form

attendees' level of preparation. If the moderator assesses the group's level of preparation to be insufficient, he or she must be prepared to end the meeting at that time — this will pay off down the line in two ways: it will encourage attendees to be better prepared in the future, and it will remove the possibility that the group will arrive at an erroneous consensus regarding the quality of the work product.

Step 2: Present the Material

The reader (or, more rarely, the author) presents the material under scrutiny. The presentation is structured into manageable "chunks," where each chunk might be a small subroutine, 10-20 lines of code, or some other manageable unit.

Step 3: Identify Defects

As units are presented, others in the group will point out apparent defects. Given the nature of the programming beast, it is likely that there will be lively discussions regarding potential defects — it is up to the moderator to keep the discussions on track and to resist the natural tendency to design/recommend solutions for the defects identified.

Severity Levels

A four-level scale is often used to assign severity levels to programming defects (Jones). This simple scale leads to a natural prioritization that serves to drive follow-on remediation activity:

Severity	Meaning
1	Total application failure
2	Major functional failure
3	Minor defect
4	Cosmetic only

Since the major payoff from code reviews comes from the early detection and rectification of major defects, a simple two-tiered scale often suffices:

Severity	Meaning
High	Major functional or application failure
Low	Minor functional failure; cosmetic issue

For instance, reviewers may note that a programmer has used a *READ TABLE* ... and that no provision has been made for efficient access to the table in question. The correction of that defect may entail using a different table definition, adding a *BINARY SEARCH* to the *READ*, or perhaps using some other mechanism to improve efficiency. It is the moderator's responsibility to make sure that the *identification* of the defect is noted, but that the *solution* is to be crafted in a setting other than the review meeting itself.

As individual defects are identified, they should be assigned a severity level based on some simple scale (see the sidebar above). From my own experience, I would lean toward the simpler of the two rating scales shown in the sidebar. As the severity levels become more granular, there is a greater tendency toward hair-splitting, which is a non-productive activity when the intent of the meeting is to identify problems and make an overall assessment of the production worthiness of a work product.

Step 4: Evaluate the Results

Once defects have been identified and categorized, conclusions must be drawn as to whether the work product as a whole is correct. Some reasonable scale of categorization must be used; typical ratings might be:

- **Accepted as is:** No rework required

 ©2003 SAP Professional Journal. Reproduction prohibited. All rights reserved.

- **Accepted conditionally:** Minor rework required, but no further code review required
- **Rework required:** Defect correction required with a subsequent formal review of the revised work product

A major output of the review process is a package of standardized documentation that incorporates the meeting inputs, the documented defects, and the overall product appraisal. (There are other meeting outputs, such as metrics to be used in evaluating the review process itself, but they are not, strictly speaking, used in the work product evaluation.)

Documentation

In addition to the documents already described (the meeting inputs, the documented defects, and the overall appraisal), the documentation package should include the actual issue logs used by participants to document defects found in their individual reviews of the work product, and the completed checklists used in the review process.

This documentation package serves a number of useful purposes:

1. It serves as a permanent record of defects detected, which aids in the process of knowledge transfer in the case of project personnel movement.
2. It can be used as a source for capturing metrics regarding defect detection.
3. It can be used to assure that standardized criteria are being applied in the appraisal process.

Bear in mind that the collection and maintenance of this documentation should not be an onerous chore — if it is, your review process will suffer. Design your review process to utilize the minimum documentation commensurate with your organization's goals for the process, and provide all participants with useful tools (pre-built templates and forms, email distribution lists, etc.) to ease the documentation load.

Rework and Follow-Up

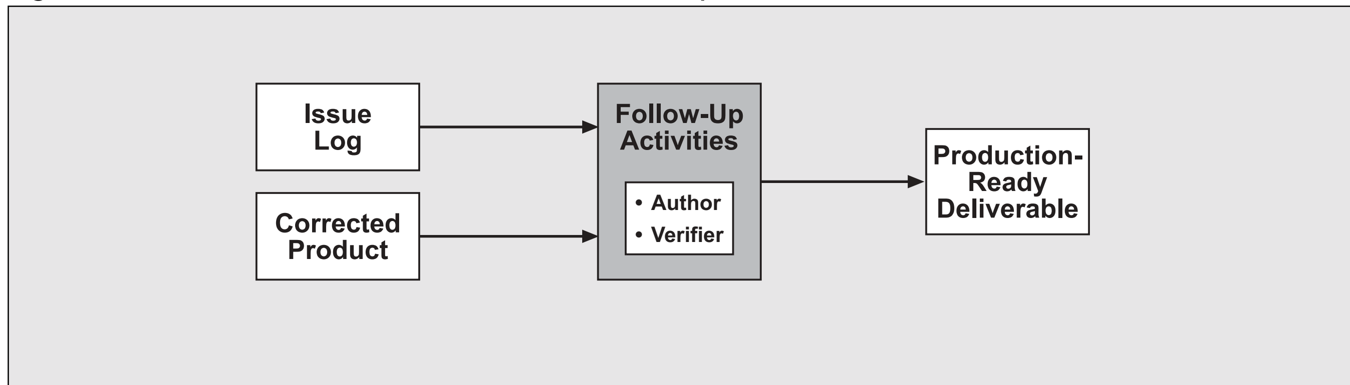
It is inevitable that reviews will reveal defects that require follow-up and rework by the program author. The documentation package is an invaluable aid in providing an organized framework for approaching the rework process.

As part of defect remediation, authors should amend the defect documentation to reflect the completion of the recommended changes, and/or to document the approaches taken to correct the defects. If an author elects *not* to correct a defect, documentation should be updated to reflect the rationale behind that decision. This type of documentation will serve two purposes:

- It will serve to warn of potential problems that might arise during testing or productive operation. For instance, in the review, an identified defect might center on the efficient access to data held in a database table. The programmer may respond that the table in question holds very few rows, and hence efficiency of access need not be addressed in the current version of the code. However, the very fact of including this information in the defect documentation (that no corrective action is being taken) provides notice that a potential problem exists.
- It will help explain to major stakeholders that explicit tradeoff decisions were made during the development process.

It is likely that issues other than those concerned with coding may arise during reviews. In the rework stage, these should also be addressed, resolved, and documented. In the example above, the appropriate remedial activity might be to create an additional index on the table in question. In such a case, the responsibility for remedial activity may more

Figure 4 *The Follow-Up Process*



appropriately lie with a database administration group, rather than with the programmer.

The major purposes of effective follow-up are to assure that authors resolve open issues documented during the review meeting, and to assure that identified defects have been repaired correctly. These two distinct goals satisfy multiple purposes of the follow-up process:

1. It allows authors the freedom to use their own judgment when correcting defects. If the author attempts to rectify defects using *all* of the ideas put forward by various reviewers, he or she will surely lose interest in full participation in future reviews.

2. It assures reviewers that their efforts and recommendations are being taken seriously. Without such assurance, reviewers will lose the motivation to participate fully in future reviews.

3. It has been shown that code corrections can introduce additional errors at the rate of 5% to 60%, depending on the complexity of the subject being modified (Jones). Erroneous fixes are especially insidious when the change is a "simple" one-liner — one that ostensibly needs no further verification. For instance, in order to speed up table access, an internal table might be redefined as a sorted, rather than standard, table. This simple change might not result in syntax errors in the program, and even an extended program check (transaction *SLIN*) may not uncover a problem. However, if the original program placed data in the table using *APPEND* rather than *INSERT*, it is possible that a short-dump runtime error might occur once the program is placed into production.

Figure 4 is a schematic representation of the follow-up process, and it illustrates that the end product of effective follow-up is a production-ready deliverable.

With a little preparation, your review processes can forestall many problems. A major tool in your arsenal will be a review checklist crafted to expose common coding defects — including generic, ABAP-centric, and Java-centric issues (see the sidebar on the next page for a discussion of the considerations involved in an SAP development environment). The checklist in the following sections is meant to provide a framework you can use to build a checklist suited to the specifics of your own organization.

An Example Code Review Checklist

Checklists are an extremely important element of the review process, since they help to ensure that reviews are standardized. The reviewer's checklist is no exception — using a common checklist assures that

 www.SAPpro.com ©2003 SAP Professional Journal. Reproduction prohibited. All rights reserved.

Code Reviews in an SAP Development Environment

What's different about the development of custom programs for use in an SAP environment? Should there be any inherent differences in the way one uses the code review process in an ABAP shop as opposed to some other development environment? I believe so, due to the unique considerations involved in terms of performance, component reuse, language technology, object-oriented programming, and Java:

- ☑ **Performance:** By far the largest numbers of problems that arise early on in the installation of new SAP programs revolve around program performance — ABAP provides ample opportunity for programmers to make coding decisions that can have significant negative impact on performance (e.g., SELECT *, READ TABLE with linear search, LOOP AT … WHERE). Luckily, these are the problems that are best-suited to a thorough pre-installation review — experienced programmers will be able to spot these quicker than a duck on a pill-bug (as we say in Texas).
- ☑ **Component reuse:** The ABAP Workbench provides extensive capabilities for the reuse of programming through the use of function modules, BAPIs, type groups, remote invocations of forms (subroutines), and include files. From a single IDE, the programmer has access to all of these with but a click or two of the mouse. The problem, of course, is one of *awareness* — of knowing that those components are available, and of making proper use of them when they've been identified. Peer review offers a vehicle for pointing out to the program author opportunities for the effective reuse of previously written (and presumably well-reviewed!) components. For instance, I can't count the number of times I have seen newer programmers write their own forms and functions to access nodes and leafs in a SET table, when there are literally hundreds of functions already available to handle this type of processing.
- ☑ **Language technology:** To the extent that ABAP enjoys a relatively high language level, simple changes in ABAP coding can have a significant impact on both performance *and* functionality. For instance, the failure to check for an empty reference table when using a simple SELECT … FOR ALL ENTRIES can result in disastrous functional results accompanied by poor performance; the failure to check the SY-SUBRC return code can result in inconsistent outcomes (and often provides opportunities for some interesting debugging experiences). These and other common coding missteps are often overlooked by program authors, but can frequently be identified via the code review process.
- ☑ **Object-oriented programming:** SAP has embraced the object-oriented programming model, while still supporting an extensive set of Basis and functional programming, which retains the procedural programming paradigm. ABAP has not become totally object-oriented, and hence it is not uncommon to find programs that employ elements of both programming models. It is important, therefore, that programs be scrutinized to see if they would be better written using one technology versus another — reviewers with broad SAP/ABAP experience can often make that judgment more effectively than newer programmers.
- ☑ **Java:** With the advent of Java as an alternative SAP development language, it will become increasingly more important for programs to be reviewed to assure that they have been written a) with the best available language; and b) using the best techniques for SAP development in that language. The addition of Java as a development alternative may serve to complicate the development process, since it adds an extra dimension (language selection) to analysis and design (while at the same time it provides for the development of programs better matched to functional requirements). For programs that address business-related problems, ABAP is often the better choice, where Java may be better for complex GUI considerations.

all reviewers are scrutinizing programs for those defects that the organization deems important, and provides a standard set of criteria for measuring the program (*not* the programmer).

The checks discussed in the following sections are examples of what you might look for in ABAP and Java programs. They are not offered as a preferred solution, but more as a framework upon which you can craft a checklist that meets the particular needs of your own organization.

Coding Documentation

All program "modules" should have their own set of documentation. The documentation may be external, it may be included in the program itself as a set of comments, it may be included using the standard ABAP module documentation facilities, such as those provided for function modules, or it may be incorporated into JavaDoc documentation. At a minimum, the following elements should be documented:

- ☑ Program; class
- ☑ Function module; method
- ☑ Include file; import
- ☑ Code block (*try/catch*; *if/then/else/endif*; *while/ endwhile*; *select/endselect*; etc.)

Review Documentation

Programmers should provide additional documentation to support the review, including documentation of:

- ☑ Extended program check (transaction *SLIN*) output
- ☑ SQL Explains for all *SELECT* statements (using a realistic representative set of selection constants)
- ☑ ABAP Runtime Analysis (transaction *SE30*) output

Even for ABAP programs without apparent performance problems, ABAP Runtime Analysis output is useful. It will point out the percentage of time spent in database accesses versus time spent in other ABAP activity. Other reporting options will allow reviewers to see which database accesses are the most "expensive" and in which forms or functions the program is spending most if its time.[4]

For analysis of Java programs, a number of third-party solutions are available that will provide much the same information for Java that ABAP Runtime Analysis provides for ABAP.[5]

Naming Standards

The subject of naming standards is rife with emotion — for every ABAP developer who believes that internal tables should have names beginning with *itab_*, there are just as many who believe that tables should begin with *t_*. And for each ABAP developer who believes that global variables should be identified by a *g_* prefix, there are as many who feel strongly that identification of global scope is an unnecessary burden upon the programmer or that global variables should be noted using some other identifier.

In the Java world, there is as much loud discussion regarding naming conventions as in the ABAP world. Most Java programming texts, for instance, devote a good part of an early chapter to this subject. Since Java is case-sensitive, naming rules usually emphasize the use of case to impart information about the object being named — this provides the opportunity for even more lively discussions regarding naming standards.

The point is that *some* standard for naming should be adopted, and one element of the review process

[4] For more on using the ABAP Runtime Analysis tool, see the article "Performance Problems in ABAP Programs: How to Find Them" in the May/June 2003 issue of this publication.

[5] Google for "JAVA PROGRAM PERFORMANCE" to find pointers to toolboxes, discussions, papers, etc., dealing with Java performance.

 ©2003 SAP Professional Journal. Reproduction prohibited. All rights reserved.

should focus on adherence to whatever naming standard has been embraced. The following list identifies some of the more common items that should be considered when promulgating your own ABAP development standards, and provides examples of some of the more commonly used naming conventions:

- ☑ Variables
  - Local: *l_...*
  - Global: *g_...*
- ☑ Internal tables
  - Local: *l_t_...*
  - Global: *g_t_...*
- ☑ Constants: *c_...*
- ☑ Selection screen
  - Select options: *s_...*
  - Parameters: *p_...*
- ☑ Ranges: *r_...*
- ☑ Forms
  - Parameters: *p_...*
  - Names: *f_...*
- ☑ Function modules
  - Imported items: *i_...*
  - Exported items: *e_...*

Opportunities for more complex naming standards exist — programs, function modules, BAPIs, development classes, variants, and so on. You may choose to embed intelligence in the names of these objects, such as company, type of program, primary functional area, periodicity, etc. Each of these constraints provides further opportunity for programmers to stray from the prescribed path, and hence should be subject to an organized review.

> ✓ ***Remember!***
>
> *Naming conventions are a form of restriction on the code, which is the personal product of the author. As such, changing it is like changing the colors on an artist's palette. Make sure, then, that your naming standards have some basis of rationality, that they are easy to remember, and that they are not overly cumbersome.*

Coding Techniques

Your organization has probably already defined a set of coding practices that are to be treated as mandatory techniques. These might include:

- ☑ Return code testing and *try/catch* coding: Are all available return codes tested and handled?
- ☑ Program performance considerations:
  - Dead code removal: Has all unreachable code been removed?
  - Binary searching of internal tables: Is data in internal tables handled efficiently?
  - "Copyless" access to internal table rows using field symbols: Is table data accessed using the *READ TABLE ... ASSIGNING* construct?
- ☑ Database performance considerations:
  - *SELECT* statements
    - Optimal use of *ORDER BY*: Is database data more efficiently ordered by ABAP sorting, rather than through the use of *SELECT ... ORDER BY*?
    - *SELECT SINGLE*
    - *SELECT ... INTO TABLE*
    - *FOR ALL ENTRIES IN*

- Index usage: Have *SELECT* statements been crafted to take advantage of available table indexes?
- SQL aggregate functions (*MAX*, *MIN*, *SUM*, etc.)
- Nested *SELECT* usage: Are *JOIN* commands used where applicable?
- *OR* usage: *SELECT ... WHERE ... OR* can affect DBMS index selection — can this be eliminated?

☑ Texts:

- Language support: Are text elements provided for all expected languages?
- Text element usage: Are text elements used in lieu of text constants/literals?

☑ Code appearance: This is another subject sure to cause lively discussion, but one that should be standardized to one level or another. Some of the more common areas of scrutiny are:

- White space usage: To what extent should white space be used to make code more readable?
- Comments: To what extent should comments be used?
- Code section (*END-OF_SELECTION, FORM*, *AT*, etc.) order: This may be an issue, especially in organizations where code may be processed/reviewed/worked on in printed form, rather than online form.

As you can see, this example checklist just barely scratches the surface of what might be important to your organization. There are many other areas that you may wish to consider depending on your particular environment:

☑ Strong(er) typing

☑ *CASE* statement usage

☑ Consistent pretty printing

☑ Inclusion of "unit test" methods in every class

☑ Java multithread considerations: deadlock; serialization; events

☑ Scalability; appropriate field sizing

☑ Application-specific type definition and usage

☑ *Meaningful* variable naming

☑ Program maintainability (this is an area of subjective opinion — another reason for having experienced technical heavyweights involved in the reviews)

☑ Minimization of global variables

☑ Removal of debug code

☑ Testing for all exceptions following function calls

☑ Violation of encapsulation principles

☑ Placement of behavior in the correct classes

☑ Placement of modification and query behavior in the same method

☑ Violations of the Law of Demeter[6]

☑ Storage of instance data in class variables

☑ Multiple statement lines

☑ "Long" forms[7]

☑ Excessive form/method argument list item counts

☑ Syntax errors in JavaDoc comments

☑ Full definition of external interfaces using JavaDoc specification

---

[6] See **www.ccs.neu.edu/home/lieber/LoD.html** for further information on this subject.

[7] Where "long" is commonly defined in terms of statement line counts.

 ©2003 SAP Professional Journal. Reproduction prohibited. All rights reserved.

The standards you'll be espousing and the review criteria you'll be applying are often particular to your organization's needs and are further dependent on subjective opinion. The formulation of your particular review checklist, therefore, can be a contentious and time-consuming exercise. Given the mobility of today's IT workforce, it's highly probable that you will have programmers who have lived with standards that differ markedly from those you've adopted, and hence may have strong feelings (one way or the other). It's likely that you'll form a working group or committee to document your standards. The individuals you choose, therefore, should be those with not only the technical maturity to be able to come up with a workable set of standards, but also with the flexibility to recognize that there may be more than one satisfactory way to do things.

The process itself, however, will provide benefits, since it yields a consensus-derived set of standards against which programs will be measured, and it serves to document those aspects of development that your organization has deemed to be critical success factors, which I will discuss next.

Critical Success Factors

A number of factors will directly affect the overall success of your review (see also the sidebar on the next page for some common pitfalls to avoid):

- ☑ Having peers, rather than customers, find defects
- ☑ Proper training of reviewers and team leaders
- ☑ Including reviews in the overall project plan
- ☑ Setting goals for the review, such as numbers of items to be reviewed or quantitative goals for defect detection
- ☑ Using analysis of prior reviews to shape the review process

While the exact nature of these critical success factors will depend upon your particular environment, there are two aspects that must not be overlooked: securing management support and preventing the "program" review from becoming a "programmer" review. Without appropriate attention to these issues, your code review cannot be successful.

Management Support

Do not believe for even an instant that the success or failure of a review does not reflect the attitudes and behavior of managers toward code reviews. While managers always want to produce quality products, they also feel competing pressures to produce products according to aggressive schedules that may not appear to allow for the time or resource commitment of performing reviews or inspections.

If managers are not cognizant of peer reviews and their potential benefits for the organization, they will not build those reviews into their project plans, they will not allocate the required resources, and, above all, they will not communicate their commitment to development team members. Lacking such management commitment, the equation is simple: if code reviews aren't supported and planned, they won't happen.

Wiegers has identified 11 indicators that help to gauge the level of management commitment to code reviews. These are the levels to which managers:

1. Provide resources and time to develop, implement, and sustain an effective review process.
2. Set policies, expectations, and goals regarding review practices.
3. Pursue the practice of reviews, even under schedule pressure.
4. Include review time in project schedules.
5. Provide review training for participants in the review process and attend the training.
6. Refrain from using review results as personnel performance evaluation criteria.

Avoiding Code Review Pitfalls

Code reviewing is not without its pitfalls. There are a number of common factors that cause a review to fail:

- ☑ Including participants who do not understand the review process
- ☑ Including participants who understand the process but do not follow it
- ☑ Including unqualified participants
- ☑ Allowing the review meetings to become forums for problem-*solving*, rather than problem-*detection*
- ☑ Lack of a champion — Someone who is familiar with the benefits of reviews can help assure the success of your review. Such a champion can overcome initial resistance and indicate to all levels within the organization that projects can benefit from the review process. I have seen instances where the *lack* of such a champion resulted in the early demise of a review begun with enthusiasm and promise.
- ☑ Chasing rats — Do not over-schedule the time allowed for review meetings; the goal of these events is *not* to scrutinize, analyze, and comment on each and every line of code. Rather, adequate time should be allocated to review the code at a high enough level to assure that apparent defects are discussed and obvious points of conflict with your organization's standards are identified. If too much time is allowed for the actual review, reviewers may get bored, they may resent the time away from their normal duties, and they may end up using their time to discuss inconsequential matters.

And finally, once you have implemented a vigorous review process, don't let yourself be lulled into a false sense of security. For instance, code reviews may not, and often don't, detect the kinds of errors that only good quality assurance and actual use will detect. For instance, where a code review might uncover a memory *leak*, only usage monitoring may discover exceptional situations the program fails to handle, such as memory *exhaustion*.

7. Refrain from using the level of review participation and constructive contribution as personnel performance evaluation criteria.
8. Publicly reward early adopters of the code review process.
9. Defend challenges to the review process from other managers and customers.
10. Respect the review team's appraisal of work products.
11. Take an active, continuing interest in the review:
    - Solicit status reports.
    - Monitor the cost of the review.
    - Assess the benefits of the review.

 www.SAPpro.com ©2003 SAP Professional Journal. Reproduction prohibited. All rights reserved.

Program (vs. Programmer) Review

The literature listed in the "References" sidebar to the right is full of discussions regarding the implications of the review process vis-à-vis interpersonal relationships. It is important, for example, that both authors and reviewers understand that the selection of work to be reviewed is not a form of punishment.

It is also important that the data collected during reviews not be used to assess author performance — review metrics should not be used to either reward or penalize individuals. The purpose of such metrics is to assist in understanding and improving your development processes, not to evaluate individuals.

Austin has devised a term, *measurement dysfunction*, to describe the outcome of such evaluation: people are motivated to behave in ways that produce results inconsistent with desired goals. There are a number of behaviors that can be caused by such dysfunction:

☑ Developers may avoid submitting work products for inspection, or may refuse to inspect others' work for fear of participating in an activity construed to result in possible punishment for the reviewed author.

☑ Reviewers may withhold identification of defects during formal reviews.

☑ Reviewers may spend unproductive time discussing finer points of whether the problem with an item is a true defect or a cosmetic issue.

☑ Review goals may make a subtle shift toward finding fewer, rather than more, defects.

In summary, if program authors or reviewers perceive the review process to be a punitive one, rather than one that provides benefits for all participants, they will be reluctant to play a productive part and the review will surely fail.

References

☑ Austin, Robert D. 1996. *Measuring and Managing Performance in Organizations*. New York: Dorset House Publishing.

☑ Beck, Kent. 2000. *Extreme Programming Explained: Embrace Change.* Boston, Massachusetts: Addison-Wesley.

☑ Boehm, Barry, and Victor R. Basili. 2001. "Software Defect Reduction Top 10 List." *IEEE Computer* 34, no. 1: 135-137.

☑ Cooper, Kenneth G., and Thomas W. Mullen. 1993. "Swords and Plowshares: The Rework Cycles of Defense and Commercial Software Development Projects." *American Programmer* 6, no. 5: 41-51.

☑ Gilb, Tom, and Dorothy Graham. 1993. *Software Inspection.* Workingham, England: Addison-Wesley.

☑ Grady, Robert B., and Tom Van Slack. 1994. "Key Lessons in Achieving Widespread Inspection Use." *IEEE Software* 11, no. 4: 46-57.

☑ Haley, Thomas J. 1996. "Software Process Improvements at Raytheon." *IEEE Software* 13, no. 6: 33-41.

☑ Holland, Dick. 1999. "Document Inspection as an Agent of Change." *Software Quality Professional* 2, no. 1: 22-33.

☑ Humphrey, Watts S. 1989. *Managing the Software Process*. Reading, Massachusetts: Addison-Wesley.

☑ Jones, Capers. 1986. *Programming Productivity*. New York: McGraw-Hill.

☑ Paulk, Mark et al. 1995. *The Capability Maturity Model: Guidelines for Improving the Software Process*. Reading, Massachusetts: Addison-Wesley.

☑ Russell, Glen W. 1991. "Experience with Inspection in Ultralarge-Scale Developments." *IEEE Software* 8, no. 1: 25-31.

☑ Van Veenendaal, Erik P. W. M. 1999. "Practical Quality Assurance for Embedded Software." *Software Quality Professional* 1, no. 3: 7-18.

☑ Wiegers, Carl E. 2002. *Peer Reviews in Software: A Practical Guide*. Reading, Massachusetts: Addison-Wesley.

Summary

This has been a very brief introduction to the subject of code reviews and their applicability in the SAP development environment. A number of authors have written extensively on this subject — I have included a list in the sidebar on the previous page that contains references to publications cited in this article. These books, papers, and articles will provide a broader treatment of the subject than these pages allow.

In brief, ABAP (and Java, as well) code reviews can significantly reduce development costs and can assist in assuring that your SAP development projects are completed on time. The very existence of a set of review considerations means that your organization has made a thoughtful evaluation of what is important. They have the additional benefits of providing a framework for the generation of documentation that records development tradeoffs, aids in training, and provides a penalty-free forum for knowledge transfer.

David Jenkins has been involved with Data Processing since 1957, when he entered the business as a punchcard machine operator. Dave worked for a Houston bank for 10 years before receiving his B.S. in Math from the University of Houston. He then joined IBM, where he worked in various marketing support positions, supporting contractors at the NASA Johnson Space Center. While at IBM, Dave spent a year teaching at New Mexico Highlands University as part of IBM's Faculty Loan Program. Since leaving IBM, Dave has received a master's in Management, Computing and Systems from Houston Baptist University, and has finished coursework for a Ph.D. in Management Information Systems at the University of Houston. Since 1996, Dave has been a consultant specializing in ABAP development — his latest assignment has been at ChevronTexaco, supporting their installation of SAP IS-Oil Production and Revenue Accounting.

Dave is married with four children and one beautiful granddaughter. He and his wife Joy live in the country 80 miles west of Houston, where they enjoy the wide-open spaces and fresh air, and their dog, cats, rabbit, fish, and purple martins. Dave can be reached at david.f.jenkins@usa.net.

 ©2003 SAP Professional Journal. Reproduction prohibited. All rights reserved.