9

Design and Code Reviews

Doing thorough design and code reviews will do more to improve the quality and productivity of your work than anything else you can do. In addition to reviewing your personal work, you should also review everything you use as a basis for this work. This includes the program's requirements, the high-level design, and the detailed design. Then, after you write the program, review it before compiling or testing it. If you find problems, fix the defects, correct faulty logic, simplify the program's structure, and bring the code up to your personal and team coding standards. If a program is unclear or confusing, add comments or rewrite it to make it simpler and cleaner. Make your programs easy to read and to understand.

Produce programs that you would be proud to publish and to show to your friends. If you do this on a project, you will produce better programs. You will also cut team testing time from months to weeks or even days, make your individual and your team's work more predictable, and have a more enjoyable and rewarding job. This chapter describes design and code reviews, why they are important, how to do them, and how to use your personal data to improve your reviewing skills.

9.1 What Are Reviews?

There are many ways to review software products. The principal methods are inspections, walk-throughs, and personal reviews. An **inspection** is a structured team review of a software product. Inspections were introduced by Mike Fagan in 1976 and are being used by a growing number of leading software organizations (Fagan 1976; Fagan 1986; Humphrey 2000). **Walk-throughs** are less formal and follow a presentation format, with an audience raising issues and asking questions. Although both walk-throughs and inspections can be effective and are used by TSP teams, they are not part of the PSP, which is a personal process. In addition, they should be done only after each developer has personally reviewed his or her work product.

In a **personal review**, you examine a product that you have developed before you give it to anyone else. Your objective is to find and fix as many of its defects as you can before anyone else implements, compiles, inspects, tests, or even reads the program. You should review the requirements before you start on the design and then review the design before implementing it. On a TSP team, you should also have team members inspect all your products after you have reviewed them and fixed all of the defects you found.

9.2 Why Review Programs?

When developers regularly produce programs that run correctly the first time, ask them how they do it. You will find that they are proud of their work. They carefully review each program before they compile or test it. They don't want anyone to see their product until they are sure it is something they are proud of. If you want quality products, you must personally review and rework them until you are satisfied with their quality.

In my early PSP research, I wrote 72 Pascal and C++ programs. The relative times it took me to fix defects are shown in Table 9.1. On average, it took eight times longer to find and fix a Pascal defect during testing than it did in the code review. The average code review time to find and fix a defect in both Pascal and C++ was between one and two minutes. The average time to find and fix a defect in unit testing was longer for both languages and was even longer after unit testing. This fix time, however, was made up of a lot of fixes that took only a few minutes and a few that took much longer.

My data for these programs also show that the fix time is essentially the same for most defect types. The principal exception was typographical defects, which took about five times as long to find and fix in unit testing as in the code

TABLE 9.1 RELATIVE DEFECT FIX TIMES

Products	Relative Code Review Fix Time	Relative Unit Test Fix Time	Relative Post-Unit Test Fix Time
47 Pascal programs	1	8	16
25 C++ programs	1	12	60

review. This was true for both the Pascal and C++ programs. Figure 8.1 (see p. 138) shows similar data for a Xerox TSP team. This figure has an exponential scale, so although finding and fixing defects took only a few minutes in code reviews, they averaged over 20 hours each in system testing.

Until they learn how to do effective code reviews, many developers write programs as fast as they can and immediately compile and test them. The compiler then leads them from one problem to the next. When the program finally compiles, they are so relieved that they try to run it. However, when programs have compile defects they almost certainly have test defects. If the first test doesn't find defects, the developers will likely declare victory and ship. Then all of the program's remaining defects must still be found, either in system testing or by the user. That is when finding defects becomes truly expensive. Reviewing a program is like reviewing a draft paper. Professional writers will tell you that the secret of good writing is rewriting. Writing good programs is much the same. Few programmers can

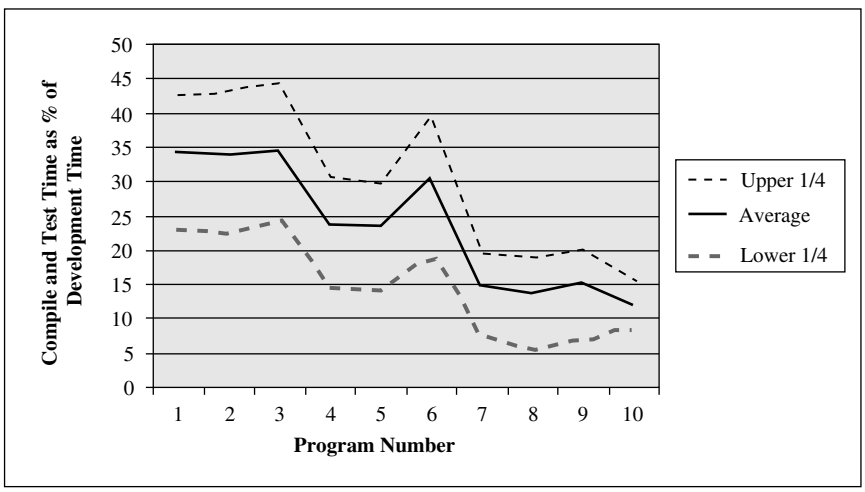

FIGURE 9.1 COMPILE AND TEST TIME RANGE (810 DEVELOPERS)

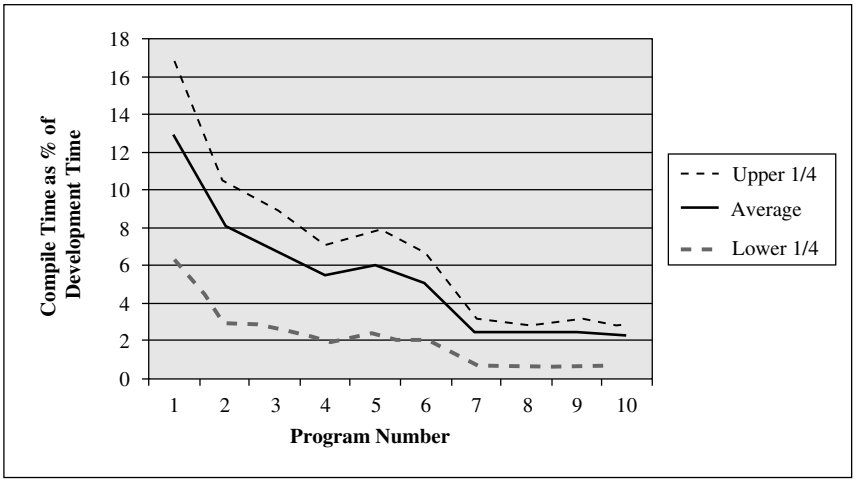

FIGURE 9.2 COMPILE TIME RANGE (810 DEVELOPERS)

write a perfect program on the first try. Many waste a lot of time compiling and testing a defect-prone first draft. Even after all this work, they still produce programs with numerous patches and many remaining defects.

Many developers, when they start PSP training, spend more than one-third of their time compiling and testing. Figure 9.1 shows the average and the upper and lower quartiles for the percentage of development time a group of 810 experienced software developers spent writing ten PSP exercise programs. On Program 1, one-quarter of the developers spent over 43% of their time compiling and testing, and three-quarters of them spent at least 23% of their time compiling and testing. The average compile and test time for these developers was 34%. By the end of PSP training, they were doing design and code reviews, and their average compile and test percentage dropped from 34% to 12%, or only about one-third of the time they had spent at the beginning of the course.

As shown in Figure 9.2, the improvement in compile time was even more dramatic. At the beginning of the course, average compile time was about 13% of development, and it dropped to about 2% by the end of the course. This improvement not only saved the developers time, but also made their development process more predictable and produced higher-quality products. Developers often argue that the compiler finds defects faster than they could and that they spend the saved compile and test time doing the reviews. As shown in Figure 9.3, however, these same 810 developers spent about 8% of their time in reviews on Program 10, while their compiling time was cut by 11%. The reduced test time and improved product quality are a free bonus.

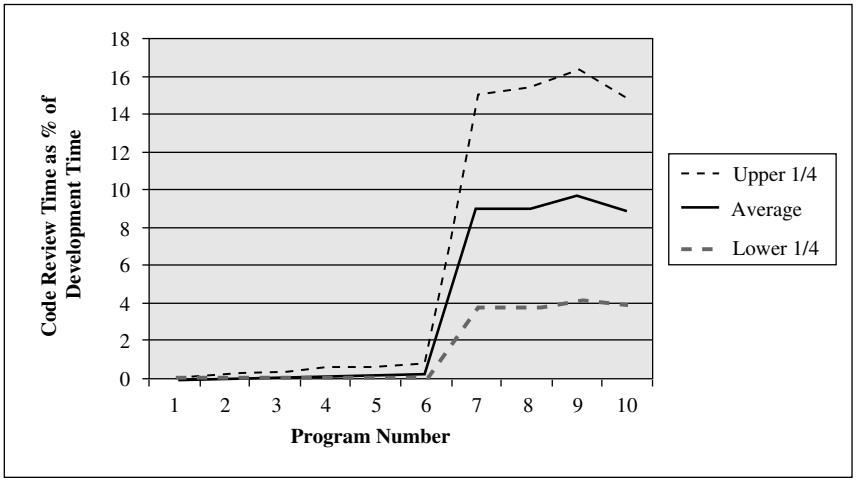

FIGURE 9.3 CODE REVIEW TIME RANGE (810 DEVELOPERS)

Although the correlation between compile and test defects is not very good for groups, Figure 9.4 shows that it is often quite good for individuals. Furthermore, Figure 9.5 shows that when developers find a lot of defects during compiling, they usually will have a lot of defects during testing. The dark bars in Figure 9.5 show

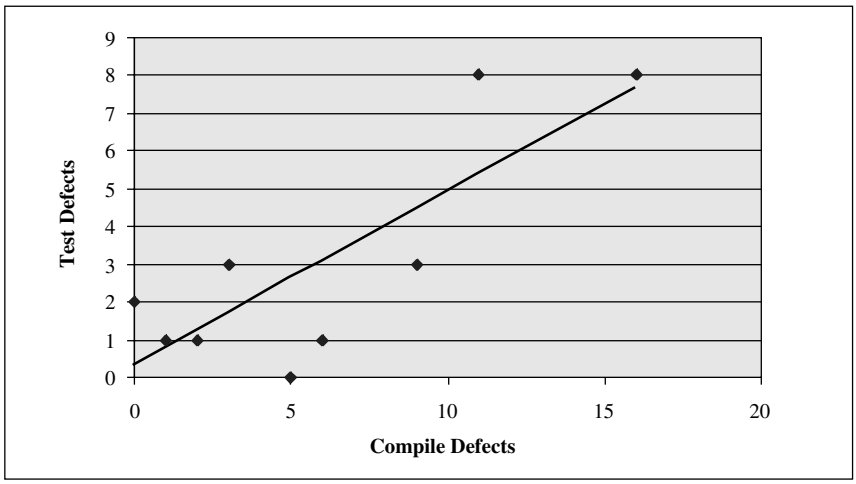

FIGURE 9.4 COMPILE VERSUS TEST DEFECTS, STUDENT 1 ($r = 0.808$)

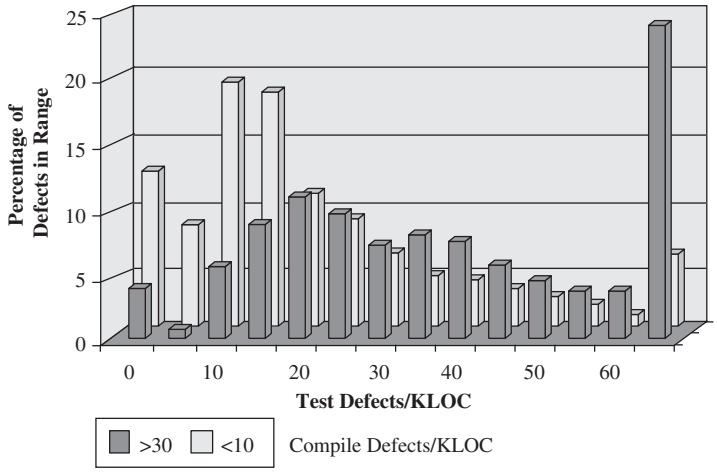

FIGURE 9.5 COMPILE VERSUS TEST DEFECT HISTOGRAM (810 DEVELOPERS)

the test defects for programs that had 30 or more compile defects per KLOC, and the light bars are for those with 10 defects or less per KLOC during compiling.

By examining your own PSP data, you will see that it is fastest and cheapest to find and fix problems before you design the wrong function or implement an erroneous design. A little time spent reviewing a program can save much more than the review time during compiling and testing. More importantly, when everyone on a development team does thorough design and code reviews, integration and system testing time is cut by a factor of five to ten or more. That is why TSP teams typically deliver on or ahead of schedule.

9.3 Review Principles

Each person is different. Until you have data on your own performance, you cannot know the most effective way for you to review programs. By using the PSP, however, you can gather your own data and see what works best for you. Try the methods that work for others and experiment with techniques you think may be better. Learn from the facts, let the data talk, and use your judgment. Give each method a fair trial but be guided by what the data tell you. Regardless of what I or anyone else may say, if your data support a particular method, then that is probably the right method for you.

The PSP process takes advantage of the fact that people are error-prone but that their errors are predictable. It is not that we are incompetent, just that we are human; and as humans, we tend to repeat our mistakes. By using our personal defect data, we can most efficiently look for the defects that we are most likely to make. The principles of the personal review process are as follows:

1. Personally review all of your own work before you move on to the next development phase.
2. Strive to fix all of the defects before you give a product to anyone else.
3. Use a personal checklist and follow a structured review process.
4. Follow sound review practices: review in small increments, do reviews on paper, and do them when you are rested and fresh.
5. Measure the review time, the sizes of the products reviewed, and the number and types of defects you found and missed.
6. Use these data to improve your personal review process.
7. Design and implement your products so that they are easy to review.
8. Review your data to identify ways to prevent defects.

By following these principles, you will improve your personal productivity and you will produce substantially better products. The reasons why these eight principles are so important are explained in the following paragraphs.

Review Your Products to Find and Fix All of Their Defects

Before doing a review, define your objective. Are you striving to improve the product or are you just glancing over the product so you can say you did a review? To avoid wasting your time, strive to find and fix the product's defects. Once you have made this your objective, the only rational goal is to find and fix all of the defects. Again, if that is not your goal, you are likely to waste your time by doing a superficial review. When you use the PSP on a TSP team, even though you will want your teammates to inspect your products as well, you should first thoroughly review them.

Although it is hard to be completely objective about your own work, it is even harder for other people to know what you intended to do. Therefore, once you finish drafting a requirement, producing a design, or writing some code, review it yourself to make sure it is what you intended to produce. By making this first check yourself, the reviews other people do will be more efficient. They won't be distracted by your obvious errors and can concentrate on the more subtle issues that you might have missed.

Your unique knowledge of your own work enables you to see things that others will miss, but the reverse is equally true. You will occasionally miss defects

because you can't see them. We often see what we intended to do and not what we actually did. Although the review process is most effective for finding almost all kinds of defects, it is a human process and error-prone. Therefore, when working on a TSP team, complement your reviews with inspections and do thorough unit tests, integration tests, and system tests. By tracking and analyzing all of the defects that you find and miss, you will do better reviews, be more productive, and produce essentially defect-free products.

Use a Checklist and Follow a Structured Review Process

Checklists help you to completely and precisely follow a procedure. To do the job properly, you must follow the checklist exactly. How would you feel if you boarded a flight and heard the pilot tell the copilot they didn't have time to do the preflight checklist? Experienced pilots rigorously complete the checklist before every flight. Their lives and ours depend on the aircraft being fully prepared for flight. Even though they may have checked the identical items one hour earlier, they will completely re-review the entire checklist. Like the pilot, you need to be convinced that missing even one item could have serious consequences.

Reviewing a program is such an apparently obvious activity that it seems almost pointless to define a checklist and a process for doing it. However, a defined process is essential. When people just scan through their programs looking for errors, they rarely find very many. To do a truly effective review, you must follow an orderly process, use sound review practices, and measure your work. Without a defined process, you cannot define the measures, and without measures, you will not have the data to evaluate and improve your reviews. The key is to take the time, preferably during the postmortem for each program, to analyze the defects you found and missed, and to determine how to do a better review the next time.

Follow Sound Review Practices

Based on the experiences in developing over 30,000 programs in PSP courses, three review practices stand out as consistently effective. First, review relatively small amounts of material at any one time. Second, do the review on paper. Third, if possible, do the review when you are rested and refreshed. If that is not possible, at least take a short break first. These practices are discussed further in later sections of this chapter.

Measure Your Reviews

The biggest problem with reviews is convincing developers to take the time to do them properly. Somehow, manually searching through a program to find and fix its defects seems slow and laborious when a powerful computer, a compiler, and a debugging facility are waiting to be used. You can read all that I or anyone else says on the subject of reviews, but the only way to convince yourself is to measure your own reviews.

Use Data to Improve Your Reviews

Human performance is extraordinary. Each year, countless world records are broken, and performances that were once considered impossible become routine. Although this is a marvelous testament to the ability of people to surpass seemingly insurmountable barriers, it doesn't happen by accident. On close examination, you will find that these record-breaking performances are only for regularly measured events. When human performance is measured only occasionally, it doesn't seem to change much from year to year.

Although we know that measured human performance improves every year, why does it? Some have argued that better equipment, training, and diet contribute, but their effects are limited. A comparison between the performance of horses and people is instructive. Figure 9.6 shows 130 years of data for the time it takes men

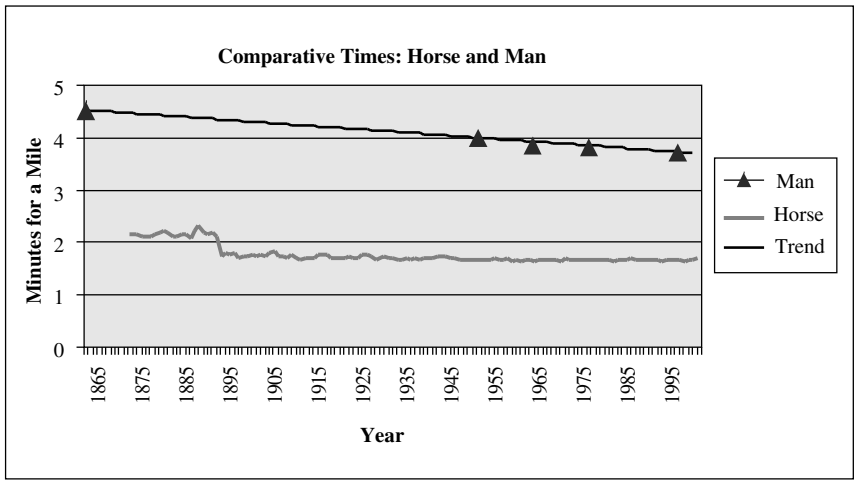

FIGURE 9.6 RECORD TIMES FOR THE MILE: MEN AND HORSES

and horses to run a mile. Whereas the world record for men has steadily improved, the winning times for horses in the Kentucky Derby have not changed significantly in over 100 years. Both men and horses benefited from better training, equipment, and diet, but only men improved consistently. The difference is motivation, and the key to motivation is feedback. The key to feedback, of course, is data.

Without measures, we have no objective way to know how well we are doing. Being essentially an optimistic species, we tend to remember what worked and forget what didn't. Although this keeps us happier, it also makes us complacent and less motivated to improve. Even worse, without measures, we cannot know how to improve, or even if we are improving. This is why a fundamental principle of PSP reviews is measurement.

Produce Reviewable Products

The seventh principle of PSP reviews is to produce products that are easy to review. This requires a lot of comments in the code and a precise and clearly documented design. To consistently produce such products, you must have both design and coding standards, and you must use them. Furthermore, as you find defects that could have been prevented with a better standard, update the standard.

Use Data to Prevent Defects

By just gathering defect data, you will be more conscious of your mistakes and make fewer of them. Figure 9.7 shows data on the defect-injection rates of 810 developers for 10 PSP programming exercises. It indicates that their defect-injection rates declined even before they used the quality methods introduced with Program 7. Defect measurement appears to be largely responsible for the 33% improvement from 120 defects per KLOC at the beginning to 80 defects per KLOC for Program 4.

You can use PSP defect data to reduce your personal defect-injection rate. Although there is not generally time to do this during the PSP course, when you work on a TSP team, consider forming a group of teammates to periodically review defect data and devise ways to prevent the highest-priority problems. There are many ways to prioritize defects—for example, by frequency, by total time to fix, by severity, or even by level of annoyance. Because these methods all have advantages and disadvantages, pick the approach that most appeals to you and then be guided by your data.

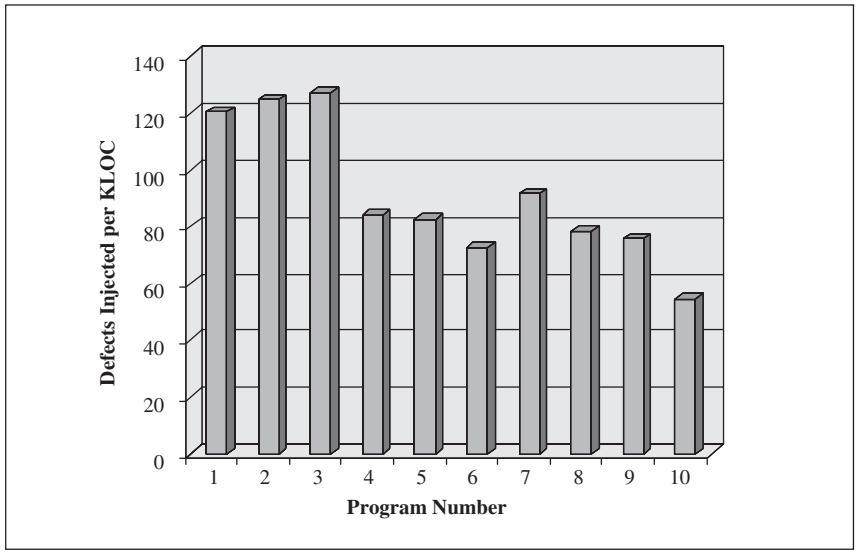

FIGURE 9.7 DEFECTS INJECTED PER KLOC (810 DEVELOPERS)

9.4 The PSP Code Review Process

The PSP Code Review Script is shown in Table 9.2. The code review entry criteria call for a complete design and a source-code listing of the program to be reviewed. You must also have copies of the Code Review Checklist, the coding standard, and the defect standard. The checklist that I used for my C++ programs is shown in Table 9.3.

In conducting the review, I first printed a listing and then went through the entire program line by line for each major checklist entry. That is, after checking that the program covered all of the design and that the includes were complete, I again reviewed every line for initialization errors. Then, I did the same for calls, names, and all subsequent entries in the checklist until I was done. At the end, I checked the exit criteria to make sure I had checked every box and fixed all of the defects. The reasons for printing a listing rather than reviewing on the screen are discussed on p. 194.

TABLE 9.2 PSP CODE REVIEW SCRIPT

Purpose		To guide you in reviewing programs
Entry Criteria		• A completed and reviewed program design • Source program listing • Code Review checklist • Coding standard • Defect Type standard • Time and Defect Recording logs
General		While following this process • do the review from a source-code listing; do not review on the screen! • record all time spent in the Time Recording log (LOGT) • record all defects found in the Defect Recording log (LOGD)
Step	**Activities**	**Description**
1	Review	• Follow the Code Review checklist. • Review the entire program for each checklist category; do not try to review for more than one category at a time! • Check off each item as it is completed. • For multiple procedures or programs, complete a separate checklist for each.
2	Correct	• Correct all defects. • If the correction cannot be completed, abort the review and return to the prior process phase. • To facilitate defect analysis, record all of the data specified in the Defect Recording log instructions for every defect.
3	Check	• Check each defect fix for correctness. • Re-review all design changes. • Record any fix defects as new defects and, where you know the number of the defect with the incorrect fix, enter it in the fix defect space.
Exit Criteria		• A fully reviewed source program • One or more Code Review checklists for every program reviewed • All identified defects fixed • Completed Time and Defect Recording logs

TABLE 9.3 CODE REVIEW CHECKLIST

Student _____ Date _____

Program _____ Program # _____

Instructor _____ Language C++

Purpose	To guide you in conducting an effective code review
General	• Review the entire program for each checklist category; do not attempt to review for more than one category at a time! • As you complete each review step, check off that item in the box at the right. • Complete the checklist for one program or program unit before reviewing the next.

Complete	Verify that the code covers all of the design.				
Includes	Verify that the includes are complete.				
Initialization	Check variable and parameter initialization. • at program initiation • at start of every loop • at class/function/procedure entry				
Calls	Check function call formats. • pointers • parameters • use of '&'				
Names	Check name spelling and use. • Is it consistent? • Is it within the declared scope? • Do all structures and classes use '.' reference?				
Strings	Check that all strings are • identified by pointers • terminated by NULL				
Pointers	Check that all • pointers are initialized NULL • pointers are deleted only after new • new pointers are always deleted after use				
Output Format	Check the output format. • Line stepping is proper. • Spacing is proper.				
() Pairs	Ensure that () are proper and matched.				
Logic Operators	• Verify the proper use of ==, =, ||, and so on. • Check every logic function for ().				
Line-by-line check	Check every line of code for • instruction syntax • proper punctuation				
Standards	Ensure that the code conforms to the coding standards.				
File Open and Close	Verify that all files are • properly declared • opened • closed				

9.5 The Code Review Checklist

A checklist is a specialized form for your personal use in doing reviews. When you use a checklist to do a review, the completed checklist copy is your record of the review. The checklist also disciplines your work and guides you through the review steps. Because you will likely change the checklist as you improve your reviews, it was designed as a separate script that you can change without changing the review process. It also simplifies the review script. When reviewing large programs, review them in parts and complete a checklist for each part. The several columns on the right of Table 9.3 enable you to use one checklist to review four separate parts of a program.

Building the Checklist

The PSP process requires that you develop your own review checklist and that you check off each item as you complete it. To build the checklist, start with the PSP Defect Type standard that was introduced with PSP0. It is shown again in Table 9.4. By gathering defect data, you will soon discover that some of these categories cause a lot of trouble, while others are not worth much attention. I found that four areas—syntax, function, interface, and assignment—accounted for 97% of all my compile and test defects.

Next, examine your defect logs to determine the defect types you encounter most frequently. Then devise checks to find them. Merely specifying "verify the logic" or "check the punctuation" is not adequate. Add concrete guidelines like "Check that all pointers are initialized to NULL" or "Verify the proper use of = and ==." When I examined my C++ defect data, I expanded the four most troublesome defect types into three or four subcategories, as shown in Table 9.5. These were not additional defect types, but more refined categories of the existing types. To produce this table, I used the descriptions recorded with each defect. This refinement enabled me to focus my reviews on the specific defect types that occurred most frequently. The percentages in the right-hand column show the total number of defects found in reviews, compiling, and testing.

A defect list sorted in frequency order is called a **Pareto distribution**. The defect type frequency shown in Table 9.5 is shown as a Pareto distribution in Figure 9.8. Using a Pareto distribution of your defects, construct a review checklist that focuses on the most prevalent types first. Although such a refined defect standard can be very helpful, its content will change as your process improves. I have found that the best practice is to use the ten basic PSP defect types and make sufficiently detailed comments to enable me to produce more precise defect categories whenever I update my checklists. In developing a checklist, also start with a well-designed coding standard.

TABLE 9.4 PSP DEFECT TYPE STANDARD

Type Number	Type Name	Description
10	Documentation	Comments, messages
20	Syntax	Spelling, punctuation, typos, instruction formats
30	Build, Package	Change management, library, version control
40	Assignment	Declaration, duplicate names, scope, limits
50	Interface	Procedure calls and references, I/O, user formats
60	Checking	Error messages, inadequate checks
70	Data	Structure, content
80	Function	Logic, pointers, loops, recursion, computation, function defects
90	System	Configuration, timing, memory
100	Environment	Design, compile, test, other support system problems

Because I based this PSP Code Review Checklist on my experiences writing C++ programs, it may not precisely fit your needs. To build a checklist that suits you, first review your defect data to see where you should focus the most attention. In addition, be careful about how you group the items on the checklist. For example, you could group all punctuation items together, as they can generally be searched for as a group. If you believe some action should be taken on every review, state it that way. However, if you believe that a checklist item is needed only under certain conditions, word it to indicate that. For example, if you want to recheck the design of any loop constructs that had defects in the design review, then your checklist should explicitly say something like "Recheck the design of every loop construct that was changed either during or since the design review."

Updating the Checklist

Examine the Pareto distribution for your defects every four or five programs to ensure that you are still focusing on the defect types that you most frequently miss in the reviews. If your data show a lot of change, do these analyses more often. By doing this, you will become more conscious of some defect types and stop injecting them. When you do, drop that defect type from the checklist. In the PSP course, do this analysis during the postmortem for every exercise program.

TABLE 9.5 EXPANDED DEFECT TYPE STANDARD

Purpose		To facilitate causal analysis and defect prevention	
Note		The types are grouped in ten general categories. • If the detailed category does not apply, use the general category. • The % column lists an example type distribution.	
No.	**Name**	**Description**	**%**
10	Documentation	Comments, messages, manuals	1.1
20	Syntax	General syntax problems	0.8
21	Typos	Spelling, punctuation	32.1
22	Instruction formats	General format problems	5.0
23	Begin-end	Did not properly delimit operation	0
30	Packaging	Change management, version control, system build	1.6
40	Assignment	General assignment problems	0
41	Naming	Declaration, duplicates	12.6
42	Scope		1.3
43	Initialize and close	Variables, objects, classes, and so on	4.0
44	Range	Variable limits, array range	0.3
50	Interface	General interface problems	1.3
51	Internal	Procedure calls and references	9.5
52	I/O	File, display, printer, communication	2.6
53	User	Formats, content	8.9
60	Checking	Error messages, inadequate checks	0
70	Data	Structure, content	0.5
80	Function	General logic	1.8
81	Pointers	Pointers, strings	8.7
82	Loops	Off-by-one, incrementing, recursion	5.5
83	Application	Computation, algorithmic	2.1
90	System	Timing, memory, and so on	0.3
100	Environment	Design, compile, test, other support system problems	0

Be careful not to drop defect types unless you no longer encounter them. If you still make a lot of off-by-one errors, for example, but find them all in the code review, you cannot stop reviewing for that defect type. If you did, you would then have to find more of them during testing. This is why you should look at the Pareto distribution sorted by both the defects you find and those that you miss.

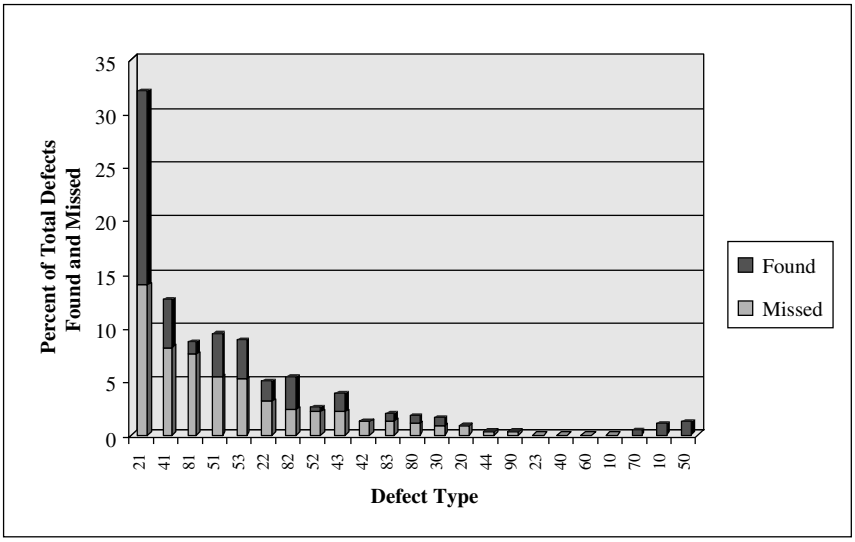

FIGURE 9.8 PARETO DISTRIBUTION OF C++ REVIEW DEFECTS

To add items to your checklist, sort the Pareto distribution in the order of the defects you miss, as in Figure 9.8. However, keep items on the list based on a Pareto distribution that is sorted by the number of defects you find by type. You should also do a sort based on the total fix times to make sure you are addressing all of the most important defects.

Reviewing against Coding Standards

Your principal interest in a code review is to ensure that all of the details are correct. Here is where a coding standard can be a big help. It should require that you initialize every parameter before use and that you first null all new pointers and then later delete them. It should also call for robustness checks to ensure that all calls have the proper parameter values, overflows are detected, and I/O errors do not disrupt program operation. Each procedure should check for erroneous input and provide clear error messages in the event of problems.

By reviewing against defined standards, you can ensure that your programs comply with the established system interfaces, the reuse criteria, the commenting guidelines, the header formats, the naming conventions, and so on. Although your programming practices will be influenced by the particular application and system environment, establish a consistent set of personal practices. Once you have

decided on your practices, incorporate them in your standards and check them in your reviews. They will help you to prevent problems, to identify problems during testing, and to protect your users from the often unpleasant consequences of using defective software.

Using a Checklist

When doing a precise task like a code review, most people find it difficult to review more than one topic at a time. For example, if you attempted to simultaneously check for name/type defects and for Boolean logic problems, you would be likely to concentrate on the name/type defects and overlook Boolean logic problems, or the reverse. Partway through the review, you might realize this and then start to concentrate on the other defect type and slight the first. As a result, you would probably not do a proper review for either defect type. You can address this problem by dividing the checklist into sections with similar characteristics. By looking for all of the name/type problems at once, for example, you are likely to be both quicker and more effective.

After an item has been on your checklist for some time, you may tend to follow it superficially. If your review experience is like mine, periodically review your data to see how you are doing. For example, on the name/type check, see how many of these defects you catch in the reviews. If you are missing a lot of them, be more careful during your next review and see if you can devise a way to be more effective.

The Review Strategy

You may also have problems reviewing multiple small routines or procedures. For example, when reviewing a class with several methods, establish a review strategy. Suppose, for example, that the class has a control method and three application methods. If the methods each have about 50 to 75 lines of source code, you might consider reviewing the entire class in one pass through the checklist. To decide if this would be a sound strategy, look again at Table 9.3 (see p. 175) and picture how you would likely do the review.

1. To ensure that the code covered all of the design, review each method to ensure that all of the required functions are included.
2. To check the includes, examine each method to ensure that the proper includes are entered for each library function.
3. To check for initialization problems, walk through the logic of all methods.

When you review in this way, you will jump back and forth among the methods. As you examine one method, you will likely build a mental context that

you may lose when you switch to another. These context switches take time, cause errors, and often result in low-yield reviews. When programs are even moderately complex, it pays to review each separable part as a unit. You would then complete the entire checklist for each part before going on to the next part. To see if this is the best approach for you, try one strategy with several programs or parts of programs, and try another strategy with some others. The data on defects per hour, LOC per hour, and yield (or percentage of defects found) should help you to decide which approach is most effective for you.

With a large hierarchical program, your design review strategy will likely follow a top-down pattern. For the code review, however, it is often best to start reviewing at the bottom. For example, consider a main routine that calls various procedures. One of these procedures, in turn, calls a file-handling routine. Start the code review with the lowest-level procedures that depend on no others—in this case, the file-handling routine. After reviewing the code in the lowest-level procedures, move up to the procedures that call them. As part of this higher-level review, check to ensure that every call is consistent with the procedure specifications. However, because you have just reviewed the lower-level procedures, you need not examine them further. By following this order, you can trust the procedural abstractions when you encounter them in the higher-level code reviews. If you don't follow such a strategy, you will keep tracing through the called procedures just to be sure that they do what they are supposed to do.

This strategy applies only when you are reviewing your own code. Here, you have a context for the reviews and probably remember the overall design, or have some design records or notes that explain the program's purpose and structure. When you start at the lowest level and work up, you are looking for local issues and ensuring that all of the coding details are correct. However, when reviewing unfamiliar code, you will likely want to follow more of a top-down strategy, at least until you understand the program's overall structure and behavior.

9.6 Design Reviews

Some design problems are extremely hard to find in source code. Examples are security problems, safety problems, state-machine design mistakes, and interface issues. Over 90% of Internet software vulnerabilities are due to software defects, and these are usually design problems. Similarly, system safety is also increasingly a software problem. All parts of safe and secure systems must be properly designed, and the design must be of high quality. This requires thorough design reviews. This chapter covers the more general design review considerations; Chapters 10, 11, and 12 cover design and design verification.

One problem with attempting to decipher a program's logic from the source code is the volume of material to be reviewed. Even simple source programs can involve several pages of documentation. A condensed design format such as pseudocode, precise mathematical notations, or standardized functional abstractions is much easier to understand. For example, name authentication is critical for security management and it is easy to lose track of name status when mired in coding details.

When reviewing source code for design defects, it is also easy to be distracted by coding problems. Obvious coding errors get in the way and make it much more difficult to visualize the larger design issues. A focus on design while reviewing code can also cause you to overlook many coding problems. To produce high-quality products, review your designs before writing the code. Then review the code too.

Another reason to review the design before writing the code is that you are much more likely to produce smaller, neater, and more understandable programs. You will also save the time that you would have spent coding incorrect logic. When you implement an incorrect design and then later review it, you will unconsciously try to minimize changes. Then you will try to correct the design to preserve as much of the existing code as possible. Because you no longer have the single priority of producing a quality design, however, design quality will almost certainly suffer.

An additional reason to review the design before implementation is that you are more likely to see and incorporate potential design improvements. This can be particularly important if you design the way I do. When faced with a logic problem, for example, I usually see some "obvious" way to solve it. Sometimes this way is sound, but occasionally it is not. With a little reflection, I will often see a neater solution that takes less code, has a cleaner structure, and is faster and easier to understand and to modify. If you design this way, and if you implement your designs before reviewing them, you will have made a large investment in this initial design.

When you review the design after writing the code, you are more likely to stick with a poor but workable design, even if it does not cleanly fit the functional need. With each new problem, you will patch this design, and soon it will be so complex that you will have trouble understanding it yourself. The practice of designing while coding is the cause of many code thickets in big systems. Such programs are often impossible to modify or even to fix. Clean designs are more likely to stay clean, even when modified. Patched-up designs generally get more complex with every change.

9.7 Design Review Principles

The PSP2 Design Review Script is shown in Table 9.6 and a basic Design Review Checklist is shown in Table 9.7. Some helpful design review guidelines are as follows:

- Produce designs that can be reviewed.
- Follow an explicit review strategy.
- Review the design in stages.
- If the design is incomplete, review the code.
- Verify that the logic correctly implements the requirements.
- Check for safety and security issues.

TABLE 9.6 PSP2 DESIGN REVIEW SCRIPT

Purpose		To guide you in reviewing detailed designs
Entry Criteria		• Completed program design • Design Review checklist • Design standard • Defect Type standard • Time and Defect Recording logs
General		Where the design was previously verified, check that the analyses • covered all of the design and were updated for all design changes • are correct, clear, and complete Where the design is not available or was not reviewed, do a complete design review on the code and fix all defects before doing the code review.
Step	**Activities**	**Description**
1	Preparation	Examine the program and checklist and decide on a review strategy.
2	Review	• Follow the Design Review checklist. • Review the entire program for each checklist category; do not try to review for more than one category at a time! • Check off each item as you complete it. • Complete a separate checklist for each product or product segment reviewed.
3	Fix Check	• Check each defect fix for correctness. • Re-review all changes. • Record any fix defects as new defects and, where you know the defective defect number, enter it in the fix defect space.
Exit Criteria		• A fully reviewed detailed design • One or more Design Review checklists for every design reviewed • All identified defects fixed and all fixes checked • Completed Time and Defect Recording logs

TABLE 9.7 PSP2 DESIGN REVIEW CHECKLIST

Student _____ Date _____

Program _____ Program # _____

Instructor _____ Language _____

Purpose	To guide you in conducting an effective design review
General	• Review the entire program for each checklist category; do not attempt to review for more than one category at a time! • As you complete each review step, check off that item in the box at the right. • Complete the checklist for one program or program unit before reviewing the next.

Complete	Verify that the design covers all of the applicable requirements. • All specified outputs are produced. • All needed inputs are furnished. • All required includes are stated.				
External Limits	Where the design assumes or relies upon external limits, determine if behavior is correct at nominal values, at limits, and beyond limits.				

Logic	• Verify that program sequencing is proper.
	  • Stacks, lists, and so on are in the proper order.
	  • Recursion unwinds properly.
	• Verify that all loops are properly initiated, incremented, and terminated.
	• Examine each conditional statement and verify all cases.				

Internal Limits	Where the design assumes or relies upon internal limits, determine if behavior is correct at nominal values, at limits, and beyond limits.				
Special Cases	• Check all special cases. • Ensure proper operation with empty, full, minimum, maximum, negative, zero values for all variables. • Protect against out-of-limits, overflow, underflow conditions. • Ensure “impossible” conditions are absolutely impossible. • Handle all possible incorrect or error conditions.				
Functional Use	• Verify that all functions, procedures, or methods are fully understood and properly used. • Verify that all externally referenced abstractions are precisely defined.				
System Considerations	• Verify that the program does not cause system limits to be exceeded. • Verify that all security-sensitive data are from trusted sources. • Verify that all safety conditions conform to the safety specifications.				
Names	Verify that • all special names are clear, defined, and authenticated • the scopes of all variables and parameters are self-evident or defined • all named items are used within their declared scopes				
Standards	Ensure that the design conforms to all applicable design standards.				

Produce Designs That Can Be Reviewed

If you are the only person who will review or use your design, then reviewability is not as great a concern. However, if other people will review your design or use it to do the implementation, make repairs, or develop enhancements, the design must be clear and understandable. I have even found that a clear and understandable design is important when reusing or enhancing my own programs. It is almost impossible to pick up a poorly documented design and review it. Not only are there issues about the design's objectives, but also there are questions about its relationship with other programs, the assumptions the designer made, and the various system standards and conventions. To minimize these problems, consider review issues when producing a design. The goal should be to produce a design that is complete, correct, and reviewable.

For a design to be reviewable, its purpose and function must be clearly stated, including an explicit list of the constraints and conditions it must satisfy. When working in a development organization, you will probably have design standards or system conventions. You should also have personal design standards. Without these standards, you will have a poor basis for doing a design review. The design description must also be complete and precise. Without a well-documented and complete design, it is impossible to do a competent design review. In Chapter 11, the design templates introduced with PSP2.1 satisfy the PSP design-completeness criteria.

Follow an Explicit Design Review Strategy

The review strategy specifies the order to follow in reviewing the various parts of the design. Because this order depends on the structure of the product, the review strategy should be part of your development strategy. The strategy of the Design Review Checklist in Table 9.7 is to start with the big picture and gradually get more detailed. You will want to review each design in conjunction with other related designs. Doing this will help you to build a review context and improve your ability to see coupling and interdependency issues. Plan the order of your design work to provide a logical and coherent order for the reviews. Although you need not review in the order you design, you do need to provide a review context for every design you review.

Review the Design in Stages

A reviewable design should be built in segments that encapsulate closely related logical elements. That is, it must suit a review strategy that covers one limited design segment at a time. During the review, you must build a mental picture of the

entire design. My rule of thumb is to limit the design review material to about one to three pages of text if possible. Although this is not always practical, good designs are usually structured into sections that are compact and easy to understand. Complex designs tend to be redundant, poorly structured, and hard to decipher.

By reviewing only relatively small and self-contained portions of the design at one time, you are more likely to do a thorough review. Then, unless you make changes, the code review should ensure only that the design was implemented properly. However, if the design has been changed after the design review, you must review these changes during the code review. Rather than debate the amount of design material to review at one time, gather data and see what review approach works best for you. We all have limitations on what we can visualize, and individual abilities vary enormously. If your design reviews have high yields, even up to several pages of logic, then feel free to review such designs. I suspect, however, that there is some size beyond which you will start missing design problems. Until you have a sense for that size limit, I suggest that you start with one to three pages.

If the Design Is Incomplete, Review the Code

If you did not produce a precise design or the design has changed, do a design review of the source code before doing the code review. Although such reviews have all of the problems described in the preceding section, it is still important to find and fix all design defects before testing. Furthermore, if you wait and try to do the design review as part of the code review, you run the risk of not doing either review properly. In addition, you will not have the benefit of a full code review for all of the design fixes you made as a result of the design review.

Verify That the Logic Correctly Implements the Requirements

Verifying that the logic correctly implements the requirements is simple in principle but it can involve considerable work. You must review every requirement and verify that it is completely covered by the design. Oversights and omissions are important defect categories that can be handled almost completely by a careful review of the design against the requirements. However, such oversights are extremely hard to find either in code reviews or in testing. Don't forget to check the use of names and parameters. Some programmers feel that the compiler handles name and type checks so efficiently that they should not bother to do them. This would be reasonable if their data showed that they never had such problems. My preference is to check all of the globally declared and state-controlling parameters and all of the specially declared types during the design review. I then defer the more local name and type checks until the code review.

The design review is also the best time to check overall product structure. A possible symptom of design problems is the passing of parameters through multiple classes or methods. Although well-designed programs often do this, it can also indicate that the original design concept did not neatly fit the problem being solved or that the design was poorly structured. This condition is easy to check during the design review and provides an early indication of possible design problems.

Check for Safety and Security Issues

The principal security checks concern overflow and exception handling and name verification and authorization. You can prevent most security problems by ensuring that *all* attempts to access or modify the system or its contents are properly authorized beforehand and that *all* possible overflow and exception conditions are properly handled. You should also follow your organization's security guidelines.

For safety, the guidelines are similar: verify that the design conforms to all of the design specifications and that all safety standards and requirements are met. The design must ensure that a safe state is always maintained, both internally and in the external environment, and potentially hazardous states must be either prevented or mitigated by the design. To the extent possible, the system should also tolerate failures, either with fail-safe modes or with degraded but safe performance in the event of failure. Safe design requires a continuous focus on quality, as defective systems cannot be safe.

9.8 Review Measures

By measuring your reviews, you get the data needed to improve their quality. A high-quality review is one that efficiently finds the greatest number of defects. To assess review quality, measure the time spent on each review and track all of the defects that you find and miss. With these data, you will see where and how to improve your reviews. Like other skills, however, reviewing takes time to learn and to master, so give yourself time to improve. The standard PSP process measures provide the data needed to evaluate and improve your reviews:

- Size of the program being reviewed
- Review time in minutes
- Number of defects found
- Number of defects in the program that were later found (the **escapes**)

The following discussion uses LOC to describe some example code review measures. However, for design reviews, no size measure is available at review time, so use text lines, pages of design, or estimated program size. For other products, use the size measures used to plan and track the work. From the basic size, time, and defect measures, you can derive several other useful measures. The most important ones are as follows:

- **Review yield:** The percentage of defects in the program that were found during the review
- **Defect density:** The defects found per volume of design or code reviewed
- **Defect rate:** The defects found per hour of review time
- **Review rate:** The size of the product reviewed per hour

These measures are discussed in the following sections.

Review Yield

Review yield was discussed in Chapter 8. It refers to the percentage of the defects in the product at the time of the review that were found by that review. The yield cannot be precisely calculated until after the product has been thoroughly tested and extensively used, but you can make fairly accurate early yield approximations. For example, if you find 8 defects in the code review and 6 while compiling, you know that you found only 8 of the 14 defects known after compilation. As shown in Table 9.8, this would result in an initial review yield estimate of 57.1%.

The essential data for the yield calculations are shown in Table 9.9. These data are available from the defect log. The yield calculations are shown in Tables 9.8 and 9.10. To produce the yield table, find in the defect log the number of defects injected and removed in each phase and enter the totals in the proper spaces. The *Cum. Injected* and *Cum. Removed* columns are the cumulative sums of the *Injected* and *Removed* columns for each phase. Note that at compile time, you would know

TABLE 9.8 YIELD CALCULATIONS (AFTER COMPILE)

Phase	Injected	Removed	Cum. Injected	Cum. Removed	Escapes	Defect Objective	Yield %
Detailed Design	5	0	5	0	5	5	0
Design Review	0	3	5	3	2	5	60.0
Code	13	1	18	4	14	15	5.5
Code Review	0	8	18	12	6	14	57.1
Compile	0	6	18	18	0	6	100.0

TABLE 9.9 DEFECT DATA

Defect #	Phase Injected	Phase Removed
1	Design	Design Review
2	Design	Design Review
3	Design	Design Review
4	Design	Code
5	Code	Code Review
6	Code	Code Review
7	Design	Code Review
8	Code	Code Review
9	Code	Code Review
10	Code	Code Review
11	Code	Code Review
12	Code	Code Review
13	Code	Compile
14	Code	Compile
15	Code	Compile
16	Code	Compile
17	Code	Compile
18	Code	Compile
19	Code	Test
20	Code	Test
21	Design	Test

about the defects found through compiling, but not the ones found in unit testing or later.

The escapes are those defects injected before or during a phase that were not found until after that phase. These escapes can be calculated for each phase by subtracting each entry in the *Cum. Removed* column from the corresponding entry in the *Cum. Injected* column. The column called *Defect Objective* holds the sum of the escapes from the previous phase and the defects injected in this phase. This is the total number of defects that could be removed in the phase. For example, in Table 9.10, there are 21 – 12 = 9 escapes from the code review.

TABLE 9.10 YIELD CALCULATIONS (AFTER UNIT TEST)

Phase	Injected	Removed	Cum. Injected	Cum. Removed	Escapes	Defect Objective	Yield %
Detailed Design	6	0	6	0	6	6	0
Design Review	0	3	6	3	3	6	50.0
Code	15	1	21	4	17	18	5.5
Code Review	0	8	21	12	9	17	47.1
Compile	0	6	21	18	3	9	66.7
Unit Test	0	3	21	21	0	3	100.0

The yield for any phase is the number of defects removed as a percentage of the defect objective. For the code review, the defect objective was the escapes from coding (17) plus those injected in the code review (0), so the yield is 100*8/17 = 47.1%. As you get the defect data from subsequent phases, you will have to recalculate the yield. Many TSP and PSP support tools calculate these figures for you. After unit testing, the yield table would look like the one shown in Table 9.10.

A high yield is good; a low yield is poor. The yield goal should be 100%. The PSP goal is to find and fix all defects before the first compile or test. By doing design and code reviews before first compiling or testing the program, you will find many defects that compiling and testing would also have caught. You are also likely to miss several. When developers start doing reviews, their PSP data show that they find only about one-third to one-half of the defects present.

By analyzing your review data and updating the Code Review Checklist, you can significantly improve review yields. With care and practice, many developers can catch 80% or more of their defects before compiling or testing. A significant number achieve one or more 100% process yield reviews during the PSP course. That is, they find all of the defects in a program before they first compile or test it. Having a program compile correctly the first time and then run all of the test cases without a single error is a marvelous experience. Data on the yields achieved by 810 developers in PSP courses are shown in Figure 9.9.

Yield Estimates

To make a yield estimate for any phase, you must assume a yield for the last completed defect-removal phase. For the data in Table 9.10, you might estimate that the unit test yield was 50%. That would mean that, after unit testing, three more defects would remain to be found. Next, assume that these defects were injected in the same proportion as those found to date. In this case, 3*6/21 = 0.857 of these undis-

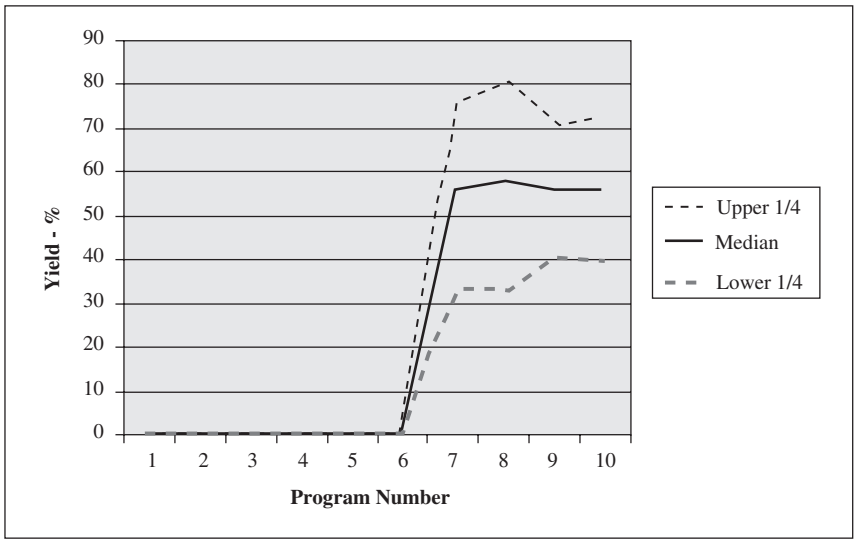

FIGURE 9.9 YIELDS FOR 810 DEVELOPERS (UPPER, MEDIAN, AND LOWER QUARTILES)

covered defects would then have been injected in detailed design, and 3*15/21 = 2.143 during coding. After system testing, you would then assume a yield of 50% for system testing and calculate the yields of all the previous phases, including unit testing. Of course, after you have historical data on the yields of these phases, you can use those numbers instead of making the 50% assumption.

Defect Density

Although the yield measure provides the best guide to review quality, it cannot be accurately calculated until well after the review is completed. Thus, to manage the quality of the reviews while you do them, you need current or instant measures that correlate with yield. That is, if you generally had high yields when you reviewed 100 or fewer LOC in an hour, and generally had low yields when you reviewed more than 300 LOC in an hour, LOC/hour would be a useful instant measure. Although defects per KLOC is also an instant measure, a low defect rate could mean either that the review was superficial or that the program had few defects. This would not be a useful measure for managing review quality. Defects per hour is also an instant measure, but it indicates only the speed with which you find defects, not how many you find or the review's effectiveness.

Review Rate

As discussed in Chapter 8 (see p. 146), the review rate measure provides useful guidance on review quality. In Figure 8.9 for example, the lowest review rates are at the front of the figure and the highest ones are at the back. By looking at the trends for 0% yield at the left and 100% yield at the right, it is clear that the lower review rates are much more effective than the higher ones.

The yield-rate relationship is often clearer for individual developers. Figure 9.10 shows review data for one developer who wrote 10 PSP exercises, and Figure 9.11 shows the results for 25 of my C++ programs. In both of these cases, reviews above about 200 LOC per hour did not generally have high yields. Similarly, although low review rates were more likely to have higher yields, that was not guaranteed. This result is consistent with the findings on review rates for code inspections, where 300 LOC per hour is considered an upper limit (Fagan 1986; Gilb and Graham 1993; Humphrey 1989). Achieving high review yield is obviously a personal skill. Lower review rates will generally improve yield, whereas an excessively fast review is likely to be a waste of time.

Defect-Removal Leverage

Defect-removal leverage (DRL) measures the relative effectiveness of defect-removal methods. It is the ratio of the defects removed per hour in any two phases, and it is useful for comparing review phases with one of the test phases, such as

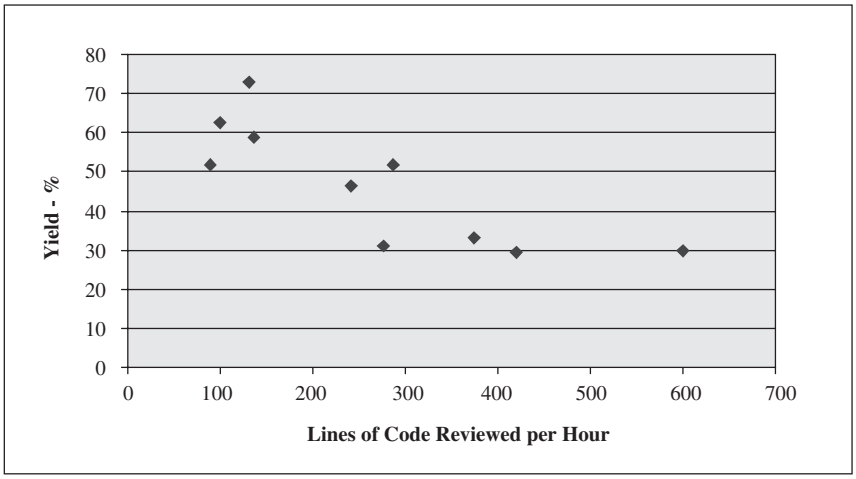

FIGURE 9.10 REVIEW YIELD VERSUS REVIEW RATE IN LOC/HOUR (STUDENT 12)

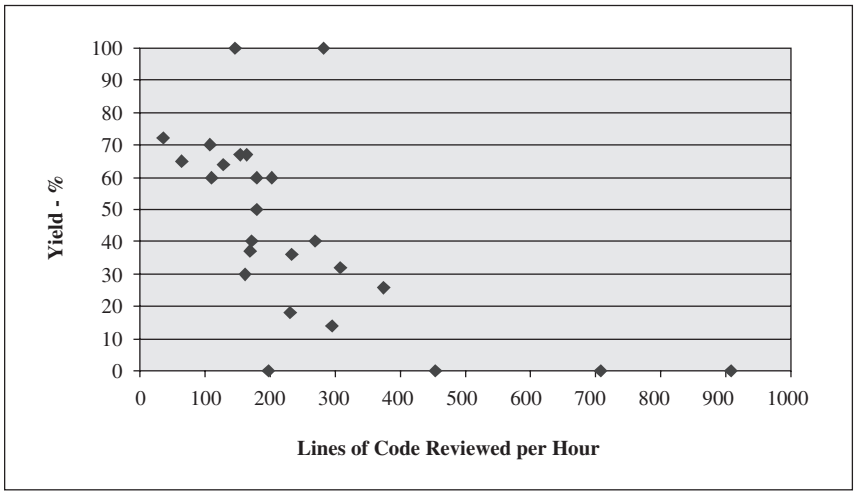

FIGURE 9.11 REVIEW YIELD VERSUS REVIEW RATE IN LOC/HOUR (25 C++ PROGRAMS)

unit testing. For example, Table 9.11 shows the average of the design and code review DRLs versus unit testing for 810 developers on PSP Programs 7 through 10. Although there is wide variation among developers, design reviews are about as fast as testing for finding and fixing defects, and code reviews are much more efficient than testing. For every 10 hours you now spend finding and fixing defects in testing, a code review would take 10/2.08 = 4.8 hours. By spending a few hours on a review, you would save 5.2 hours (less review time) and have higher-quality code. The improved program quality would then save you even more time in the integration, system, and acceptance testing to come.

TABLE 9.11 DEFECTS PER HOUR AND DRL (PROGRAMS 7 THROUGH 10)

Phase	Defects/Hour	DRL vs. UT
Design Review	3.22	0.97
Code Review	6.90	2.08
Unit Test	3.31	1.00

9.9 Review Issues

Thousands of PSP exercise programs have now been written by experienced developers, and the data show that personal reviews are the most cost-effective and efficient way to remove defects. However, to get the maximum benefit from personal reviews, you must do them properly. The following sections discuss some of the most common review issues.

Review Practices

By consistently following five review practices, your reviews will be most effective and efficient:

1. Take a break after coding. By taking even a brief break before you start the review, you will be more rested and more likely to see problems.
2. Print and use a program listing for the review. Trying to review a program on the screen is often a complete waste of time. If you don't believe me, do several reviews each way and see what your data tell you.
3. As you do the review, check off each checklist item as you complete it. When you do, treat the check as your certification that the program has no remaining defects of that type.
4. Periodically update the Design and Code Review checklists to reflect your latest data.
5. Develop and use a different checklist for each programming language.

Reviewing on the Screen

Many developers, when they first do code reviews, want to do the review on the screen. However, if you do this, you will find that the screen focuses your attention on a very small portion of the program. This narrow view makes it hard to see the relationships between your program and any other programs or to examine overall structure, security, performance, or any other larger-scale program properties. With a little experience, you will see that reviewing a listing is actually faster than reviewing on the screen, and you will find many more of the program's defects.

Review Economics

If there were only one or two defects in a one-KLOC program, you might question the wisdom of studying every line to ensure its correctness. Furthermore, your willingness to study every line will depend on how important it is to you to produce a

defect-free program. PSP data show, however, that even experienced developers typically inject about 100 defects per KLOC. The defect-injection rates for 810 experienced developers who wrote ten PSP programs are shown in Figure 9.7 (see p. 173). At the beginning of PSP training, the average defect-injection rate was 120 defects per KLOC, and 25% of the developers injected 150 or more defects per KLOC.

Before PSP training, developers typically find about half of their defects in compiling and about half of the rest in unit testing. The ones they don't find, of course, are left for integration testing, system testing, or the users. By the end of the PSP course, the average defect-injection rate is reduced to 54 defects per KLOC, a 55% reduction. Although some of this improvement is due to the PSP design methods discussed in the next three chapters, these methods are not introduced until the defect-reduction rate has fallen by about 30% to 40%. The principal reason for this initial improvement is that the developers have been recording data on their defects, so they are more aware of their personal mistakes and are both more motivated and better able to prevent them. This improvement is largely due to defect recording and defect analysis.

Because of the large numbers of defects, the major quality-management issue is not how long it takes to find a rare defect, but how to best find, fix, and prevent a lot of defects. The principal lesson from the PSP is that finding defects in personal reviews is much more efficient than finding them during testing. Of course, preventing them in the first place is even more efficient. As shown in Chapter 8, the enormous schedule impact of finding and fixing numerous defects in system testing justifies almost any time developers spend doing thorough design and code reviews.

Review Efficiency

Code reviews are inherently more efficient than testing. In a review, you find the defects directly; in testing, however, you only get symptoms. The time required to get from symptoms to defects is called debugging. **Debugging** is the process of finding the defective code that caused the program to behave improperly. Sometimes trivial defects can produce unbelievably complex system behavior and require a great deal of time to figure out what caused the weird symptoms. The amount of debugging time generally bears little relationship to the sophistication of the defect.

It might seem surprising that the difference in defect fix time between reviews and testing could be so large. When reviewing a program, you know where you are and can see the results that the logic is supposed to produce. As you work through the design or code, you establish logical relationships and gradually construct a mental context of the program's behavior. When you see something that you don't understand, it is often because the program doesn't do what you thought

it would. With deeper study, you sometimes find that the program is correct, but you sometimes conclude that it is not. You thus start with a potential problem and try to find its consequences. With experience, you learn to make such searches quite directly and logically. You don't need to stumble into a lot of blind alleys to figure out what could have caused the unexpected behavior. You are also in a better position to correctly fix any problems you find because the review helps you to build a mental picture of what the program does and why.

In debugging, the situation is quite different. You start with some unexpected system behavior. For example, the screen could go blank, or the system may just hang, produce an incorrect result, or print out gibberish on the eighty-seventh iteration of a loop. Some confirmed debuggers will argue that they can find defects more efficiently with a debugging tool than they can by inspecting the design or the code. Although it is true that some defects can be traced quite quickly, others can be extraordinarily hard to find. In one operating system example, three experienced developers worked for three months to find a subtle system defect that was causing persistent customer complaints. At the time they found the defect, the same code was being inspected by a different team of five developers. As an experiment, the inspection team was not told about the defect. Within two hours, this team had found not only this defect, but 71 others! Once found, the original defect was trivial to fix. Interestingly, no one on the inspection team thought that it would take more than an hour or two to find this defect during testing.

The debugger's principal advantage is that it helps you to step through the program logic and to check the important parameter values. However, this process is effective only if you know what the parameter values are supposed to be. Therefore, you must have already worked through the program's logic. If you must go to all that trouble anyway, why not be a little more thorough and check the accuracy of the logic? In complex cases, you should also use the debugger to check your review results.

Reviewing before or after Compiling

Although a growing number of development environments no longer require compiling, some still do. With a newer environment, you could turn off the automated checking until after doing the code review. However, I have not found that useful. The best practice is to use all of the available tools to produce the code but then to do a high-yield review before using any automated checking, formatting, compiling, or testing tools. Again, the best guideline is to try various techniques and see what your data tell you.

For those who still use compilers, one of the more contentious issues is whether to review the code before or after compiling it. Some programmers refuse to consider reviewing their code before they do a first compile. They feel that the

compiler is designed to find simple syntax errors and that trying to find them by hand is a waste of time. However, the data show that this is not true. The compiler is not designed to find errors; its job is to produce code. If it can produce code from the erroneous source you give it, it will. The compiler flags defects only when it can't figure out how to generate code from whatever you wrote.

The issue of reviewing before or after compiling is related to tool effectiveness. For example, if you had a tool that would reliably find 100% of a class of your programming mistakes, then you would not have to worry about any of them. However, if the tool identified 99.9% of these mistakes, then you would have to find the other 1-in-1,000 error yourself. Now the question concerns the importance of finding that 1-in-1,000 defect. If it might cause a little user inconvenience, you could ignore it; but if it could result in a user's death, you would want to find it.

Assuming that you planned to use code reviews to find and fix these rare defects, you must decide whether to run the tool before the review or afterwards. By running the tool first, you would find practically none of these defect types. Then you would have no way to judge the quality of your review. However, if you reviewed the code first and then ran the tool, a comparison of the number of defects you found with those found by the tool would indicate the quality of your review.

If the compiler caught 100% of all your syntaxlike mistakes, you would not have to search for them in a code review. My data on 2,041 defects show that 8.7% of my Pascal syntaxlike mistakes and 9.3% of my C++ syntaxlike mistakes were not caught by the compiler. This does not mean that the compiler was faulty, but that random coding and typing mistakes occasionally produce valid syntax. The code, however, does not do what was intended. The defect types that were caught and missed by the C++ compiler are shown in Figure 9.12. Here, the 9.3% syntaxlike defects that were missed by the compiler were of defect types 20, 21, 22, 41, and 51. These defect types are shown in Table 9.5 (see p. 178). Compilers can be very effective at finding syntax, naming, and referencing defects, but you cannot rely on them to find all of even these defect types.

The arguments for compiling before doing a code review are the following:

- For some defect types, compiling will find defects about twice as fast as code reviews.
- The compiler will find about 90% of the coding and typing mistakes.
- In spite of your best efforts, your reviews will likely miss between 20% and 50% of the syntax defects. This result is highly individual, however, as some reviewers can catch most syntax problems while others catch very few.
- The syntaxlike defects missed by the compiler are generally not difficult to find during testing.
- Some development environments, such as Java and VB.NET, highlight the defects so that they can be fixed directly.

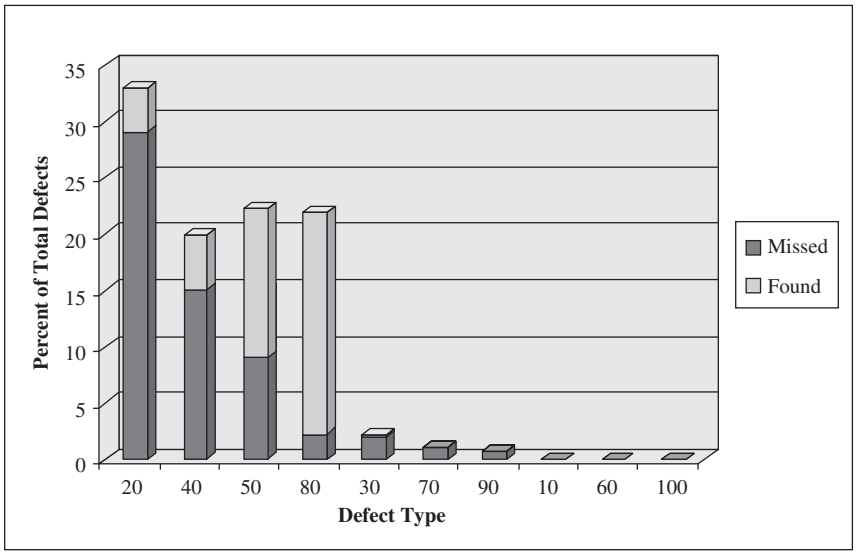

FIGURE 9.12 C++ DEFECTS FOUND AND MISSED BY THE COMPILER

The points that favor reviewing before compiling are as follows:

- Because the compiler will miss about 9% of your syntaxlike mistakes, you will have to review for them anyway.
- Doing the review first will cut your compile time from about 6% to 16% of development to about 2% (as shown in Figure 9.2 on p. 166).
- On average, it takes several times longer to fix syntax defects in testing than it does in the code review. In testing, you will quickly find most of the syntax defects the compiler misses, but some will occasionally take much longer.
- Unit testing typically finds about half of a program's defects. If you count on the compiler to find all of the syntaxlike mistakes, about 4% of them will likely escape both the compiler and unit testing and will have to be caught by integration testing, system testing, or the users. One such defect could cost you or your teammates many hours.
- Typically, subsequent test phases have even lower yields than unit testing (Thayer et al. 1978).
- If you are going to review the code anyway to find syntaxlike defects, you won't save time by compiling first. However, you will save time by reviewing first.

- Because you know that the compiler will find nearly all of the syntaxlike defects, you can treat them like seeded defects (Knight and Ammann 1985). That is, you can use the compiler to evaluate the thoroughness of your reviews. If you found 80% of the syntax defects, you probably also found about 80% of the other coding defects and many of the design defects. Of course, some design defects cannot be found by a typical code review.
- You will likely get great satisfaction from a clean first compile.
- Reviewing before compiling is more rewarding and more effective. If you don't review before compiling, your reviews of the compiled code will find only about one-tenth as many syntax and naming defects. Reviewing would then seem like a waste of time and you could easily get discouraged and do a poor review. After a while, you would likely stop doing code reviews entirely.

Review Objectives

If your objective is to get into testing as quickly as possible, there is probably no way to convince you to do a review before compiling. This attitude, however, confuses speed with progress. Although getting to testing would seem like progress, the testing time for defective code is highly unpredictable, and for large programs it can take many weeks or months. If you measure your overall performance both with reviews and without, you will find that by reviewing first, you may take a little longer to get into testing but the testing time will be much less.

Conversely, if your goal is to remove the maximum number of defects, you will want to do the code reviews when they are most effective. The compiler is equally effective before or after the review, so if you find any of the defects it would miss, you are ahead. My preference is to review the code before compiling and then use the compiler as a check on the quality of my code review. If the compiler finds more than a very few defects, I probably have a quality problem and need to reexamine my review practices. This is also a good guide for when to update your Code Review Checklist.

The Relationship between Reviews and Inspections

In a review, you personally review your own program. An inspection is a team review of a program. After personal reviews, inspections are the most valuable quality technique a software team can use (Fagan 1976; Gilb and Graham 1993; Humphrey 2000). If your objective is to achieve 100% process yield, incorporate inspections into your personal process. You could even use them for the exercises in this book. To do that, either change the process to include inspections or record your and your inspectors' times in your respective time logs. Then record all of

the defects you and everyone else find in *your* defect log. Also note in the comments section of the time and defect logs that these data were for inspections. Then, record your and your inspectors' inspection data on your Process Improvement Proposal (PIP) form, both to have a personal record and to tell your PSP instructor what you did.

When using inspections, you must decide where in the process to put them. The principal questions are whether to review the code before the inspection and whether to compile and test the code before the inspection. The central issue here is one of personal standards. Would you ask someone to review the rough draft of a technical report? You would presumably read it first to eliminate the obvious mistakes. Programs are much the same. Most first-draft programs have obvious defects that are quickly found in even a cursory review. Do you want each of your reviewers to separately wade through these obvious defects just because you were too lazy to find and correct them first?

The people who inspect your code are taking their precious time to help you improve the quality of your product. To show your appreciation, treat their time as important and carefully review your code before the inspection. This is not just a question of courtesy; obvious defects are distracting. When there are many simple problems in the code, the reviewers are less likely to see the important issues. Because you can easily find most of the simple problems yourself, thoroughly review your designs and code and make them as clean as you can before having them inspected.

Whether or not you compile before the inspection is more debatable. The issue here concerns where you need help. While you are gaining fluency with a language, it might help to have associates inspect the code before you compile. Your objective, however, is to improve your reviewing skills so that you can later find most of the syntax mistakes yourself. Once you become more fluent with the language, it will be more effective to review and compile first and have the inspection focus on the more subtle requirements, design, and coding issues.

If the quality of the code entering an inspection is so important, why not also unit-test it first? This is a question of inspection psychology. When the inspectors know that the code has been tested, they are not as likely to do a thorough analysis. It would seem pointless, for example, to struggle through a complex while-loop verification when you know that it has already been tested. Even though most programmers understand that an initial unit test will not find all of the defects in a complex program, the fact that the program is known to run can be demotivating. I recommend that you not test your programs before the first code inspection. The inspection objective should be to have no defects found in testing. Your personal review goal, of course, is to have no defects found in compiling, inspections, or testing.

9.10 Summary

This chapter deals with personal design and code reviews. The purpose of a review is to efficiently produce a high-quality program. The principal kinds of reviews are inspections, walk-throughs, and personal reviews. Reviews should be done for the requirements, the design, the documentation, and all other product elements. Many software projects spend nearly half of their development time in testing. This is very inefficient. Design and code reviews are much more cost-effective for finding and fixing many defects. Reviews find defects directly, whereas tests only provide symptoms. When reviewing a program, you will know where you are and what the logic is supposed to do. Your fixes are therefore likely to be complete and correct. In testing, you must figure out the source of the problem. However, because you will almost certainly be rushed, there is a good chance that you will introduce new defects when correcting an existing one.

Reviews provide a greater return on your personal time than any other single thing you can do. Read each program, study it, and understand it. Then fix the defects, the logic, the structure, and the clarity. In addition, learn to rewrite programs. When areas are unclear or confusing, add comments, or better yet, do a complete rewrite. Make it easy to read and to understand. Produce something that you would be proud to publish or to show to friends and associates. The basic principles of personal reviews are to establish review goals, follow disciplined methods, and measure and improve the review process.

Your decision to do reviews is driven by your desire to be productive and to create quality products. To determine whether your reviews are helping to achieve these goals, measure them. Use a Pareto distribution to establish review priorities and to develop a review checklist. Reexamine this Pareto distribution periodically to ensure that you are still focusing on the most significant defects.

Although yield provides the best measure of review quality, you cannot calculate yield until after you have completed development and testing. To manage your personal reviews, you need current or instant measures that correlate with yield, such as LOC reviewed per hour. You can then use the review rate data to improve the quality of your reviews. A review rate of under 200 LOC per hour is generally required for high-yield reviews.

One of the more contentious PSP issues is whether to review code before or after compiling it. Although there are valid arguments for both options, my preference is to review the code before compiling. I then use the compiler as a statistical check on the quality of my review. If the compiler finds more than a very few defects, there is likely to be a quality problem.

9.11 Exercises

The assignment for this chapter is to use PSP2 to write one or two programs. If you produced a Code Review Checklist while writing the R4 report, use it. If not, before writing these programs, use your defect data to produce one. The PSP2 process and the program specifications are given in the assignment kits, which you can get from your instructor or at **www.sei.cmu.edu/tsp/psp**. The kit contains the assignment specifications, an example of the calculations, and the PSP2 process scripts. In completing this assignment, follow the PSP2 process, record all required data, and produce the program reports according to the specifications in the assignment kits.

References

Fagan, M. "Design and Code Inspections to Reduce Errors in Program Development." *IBM Systems Journal* 15, no. 3 (1976).

———. "Advances in Software Inspections." *IEEE Transactions on Software Engineering,* vol. SE-12, no. 7 (July 1986).

Gilb, T., and D. Graham. *Software Inspections.* Reading, MA: Addison-Wesley, 1993.

Humphrey, W. S. *Managing the Software Process.* Reading, MA: Addison-Wesley, 1989.

———. *Introduction to the Team Software Process.* Boston: Addison-Wesley, 2000.

Knight, J. C., and P. E. Ammann. "An Experimental Evaluation of Simple Methods for Seeding Program Errors." *Proceedings of the Eighth International Conference on Software Engineering,* August 28–30, 1985, Imperial College, London.

Thayer, T. A., M. Lipow, and E. C. Nelson. *Software Reliability: A Study of Large Project Reality.* TRW Series of Software Technology 2 (Amsterdam: North-Holland, 1978).