- *Response time.* A measure of how quickly the software reacts to a given input or how quickly the software performs a function or activity.
- *Resource utilization.* A measure of the average or maximum amount of a given resource (for example, memory, disk space, bandwidth) used by the software to perform a function or activity.
- *Accuracy.* A measure of the level of precision, correctness, and/or freedom from error in the calculations and outputs of the software.
- *Capacity.* A measure of the maximum number of activities, actions, or events that can be concurrently handled by the software or system.

Maintainability

The ISO/IEC *Systems and Software Engineering—Vocabulary* (ISO/IEC 2009) defines maintainability as "The ease with which a software system or component can be modified to change or add capabilities, correct faults or defects, improve performance or other attributes, or adapt to a changed environment." Again, several metrics can be used to measure maintainability. Examples include *mean time to change* the software when an enhancement request is received or *mean time to fix* the software when a defect is detected. Another maintenance metric looks at the number of function points a maintenance programmer can support in a year.

2. PROCESS METRICS

> Measure the effectiveness and efficiency of software using functional verification tests (FVT), cost, yield, customer impact, defect detection, defect containment, total defect containment effectiveness (TDCE), defect removal efficiency (DRE), process capability and efficiency, etc. (Apply)
>
> **Body of Knowledge V.B.2**

Process entities include software-related activities and events. Process entities can be major activities, such as the entire software development process from requirements through delivery to operations, or small individual activities such as the inspection of a single piece of a document. Process entities can also be time intervals, which may not necessarily correspond to specific activities. Examples of time intervals include the month of January or the first six months of operations after delivery.

Examples of internal attributes associated with process entities include cycle time, effort, and the number of incidents that occurred during that process (for example, the number of defects found, the number of pages inspected, the number

of tasks completed). Examples of external attributes associated with process entities include controllability, efficiency, effectiveness, stability, and capability.

Cost

The *cost* of a process is typically measured as either the amount of money spent or the amount of effort expended to implement an occurrence of that process, for example, the number of U.S. dollars spent (money) for conducting an audit or the number of engineering hours (effort) expended preparing and conducting a peer review. Collecting metrics on average cost of a process and cost distributions over multiple implementations of that process can help identify areas of inefficiencies and low productivity that provide opportunities for improvement to that process. After those improvements are implemented, the same cost metrics can be used to measure the impacts of the improvements on process costs in order to evaluate the success of the improvement initiatives.

Cost metrics are used to estimate the costs of a project and project activities as part of project planning and to track actual costs against budgeted estimates as part of project tracking. The collection and evaluation of process cost metrics can help an organization understand where, when, and how they spend money and effort on their projects. This understanding can help improve the cost estimation models for predicting costs on future projects.

First-Pass Yield

First-pass yield evaluates the effectiveness of an organization's defect prevention techniques. First-pass yield looks at the percentage of products that do not require rework after the completion of a process. First-pass yield is calculated as a percentage of the number of items not requiring rework because of defects after the completion of the process, divided by the total number of items produced by that process. For example, if the coding process resulted in the creation and baselining of 200 source code units, and 35 of those units were later reworked because defects were found in those 35 units, then the first-pass yield for the coding process is $([200 - 35] / 200) \times 100\% = 82.5\%$. As another example, if 345 requirements are documented and baselined as part of the requirements development process, and 69 of those requirements were modified after baselining because of defects, then the first-pass yield of requirements development is $([345 - 69] / 345) \times 100\% \approx 80\%$.

Cycle Time

Cycle time is a measurement of the amount of calendar time it takes to perform a process from start to completion. Knowing the cycle times for the processes in an organization's software development life cycle allows better estimates to be done for schedules and required resources. It also enables the organization to monitor the impacts of process improvement activities on the cycle time for those processes. Cycle time can be measured as either static or dynamic cycle time.

Static cycle time looks at the average actual time it takes to perform the process. Cycle time helps answer questions such as "how long on average does it take to code a software unit, to correct a software defect, or to execute a test case?" For

example, if four source code units were programmed this week and they took five days, 10 days, seven days, and eight days, respectively, to program, the static cycle time = (5 + 10 + 7 + 8) / 4 = 7.5 days per unit.

Dynamic cycle time is calculated by dividing the number of items in progress (items that have only partially completed the process) by one-half of the number of new starts plus new completions during the period. For example, if 52 source code units were started this month, 68 source code units were completed this month, and 16 source code units were in progress at the end of the month, the dynamic cycle time = (16 / [(52 + 68) / 2]) × 23 days (the number of working days this month to convert months to days) ≈ 6.1 days per unit.

Customer Impact—Customer Satisfaction

Customer satisfaction is an essential element to staying in business in this modern world of global competition. An organization must satisfy and even delight its customers with the value of its software products and services to gain their loyalty and repeat business. So how satisfied are an organization's customers? The best ways to find out is to ask them using customer satisfaction surveys. Several metrics can result from the data collected during these surveys. These metrics can provide management with the information they need to determine their customer's level of satisfaction with their software products and with the services associated with those products. Software engineers and other members of the technical staff can use these metrics to identify opportunities for ongoing process improvements and to monitor the impact of those improvements.

Figure 19.12 illustrates an example of a metrics report that summarizes the customer satisfaction survey results and indicates the current customer satisfaction level. For each quality attribute polled on the survey, the average satisfaction and importance values are plotted as a numbered bubble on an *x*–*y* graph. It should be remembered that to make calculation of average satisfaction level valid, a ratio scale measure should be used (for example, a range of zero to five, with five being

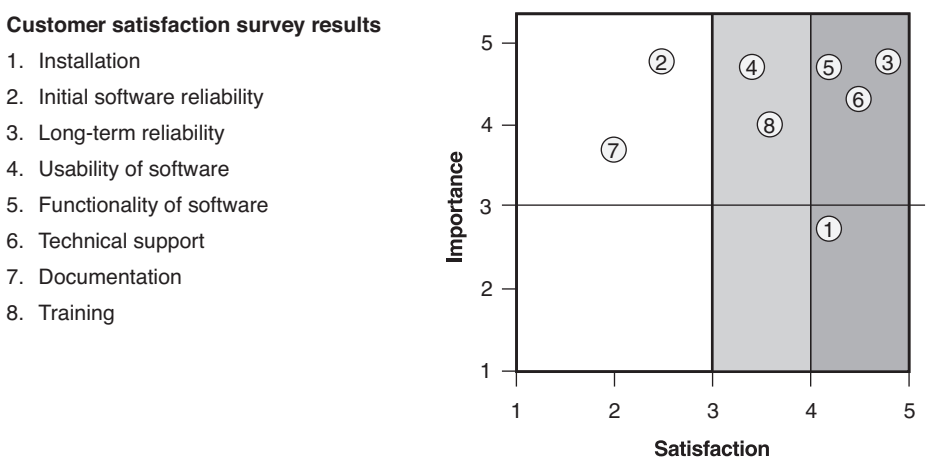


Figure 19.12 Customer satisfaction summary report—example.

very satisfied). If an ordinal scale metric is used, the median should be used as the measure of central tendency. The darker shaded area on this graph indicates the long-term goal of having an average satisfaction score of better than 4 for all quality attributes. The lighter shaded area indicates a shorter-term goal of having an average satisfaction score better than 3. From this summary report it is possible to quickly identify "initial software reliability" (bubble 2) and "documentation" (bubble 7) as primary opportunities to improve customer satisfaction. By polling importance as well as satisfaction level in the survey, the person analyzing this metric can see that even though documentation has a poorer satisfaction level, initial software reliability is much more important to the customers and therefore should probably be given a higher priority.

Figure 19.13 illustrates an example of a metrics report that shows the distribution of satisfaction scores for three questions. Graphs where the scores are tightly clustered around the mean (question A) indicate a high level of agreement among the customers on their satisfaction level. Distributions that are widely spread (question B), and particularly bimodal distributions (question C), are candidates for further detail analysis.

Another way to summarize the results of a satisfaction survey is to look at trends over time. Figure 19.14 illustrates an example of a metrics report that trends the initial software reliability based on quarterly surveys conducted over a period of 18 months. Again, the dark and light shaded areas on this graph indicate the long- and short-term satisfaction level goals. One note of caution is that to trend the results over time the survey must remain unchanged in the area being trended. Any rewording of the questions on the survey can have major impacts on the

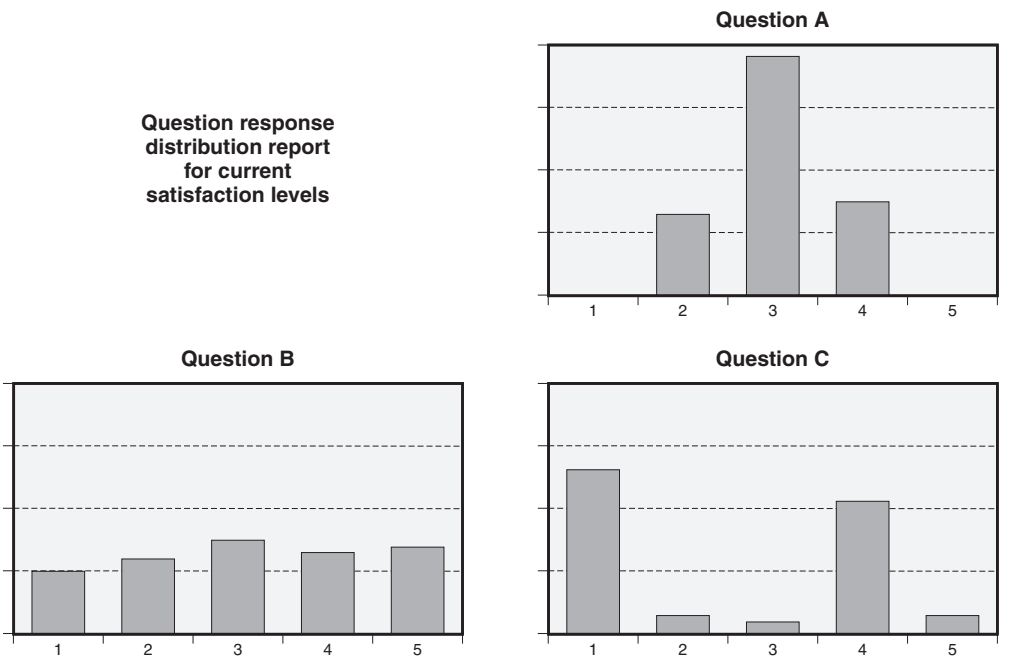


Figure 19.13 Customer satisfaction detailed reports—example.

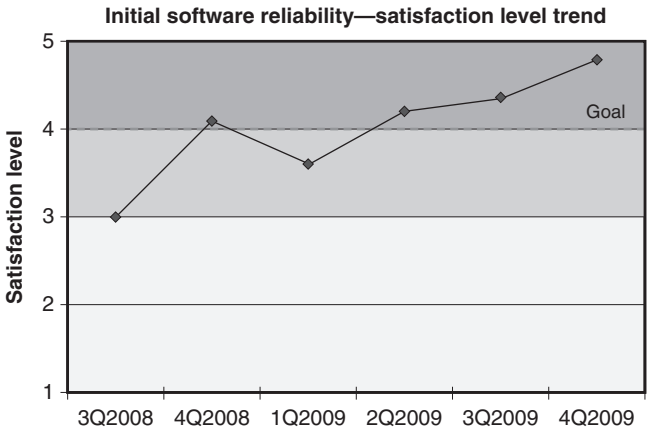


Figure 19.14 Customer satisfaction trending report—example.

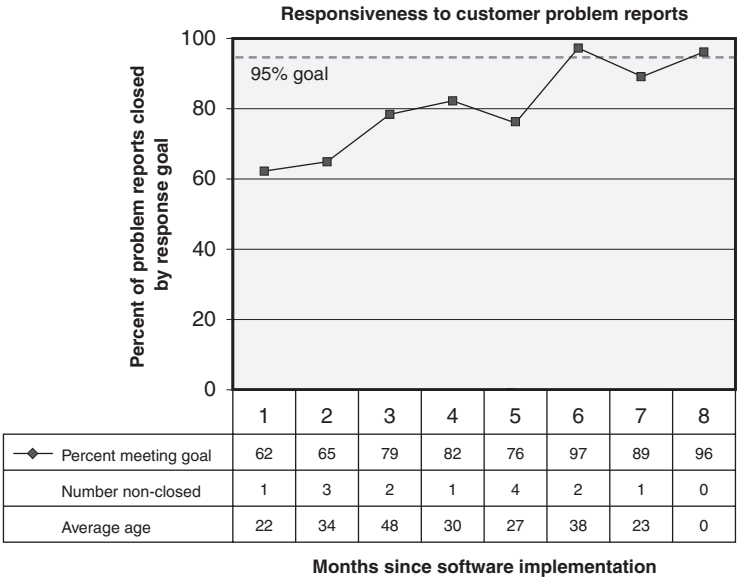
survey results. Therefore, historic responses from before wording changes should not be used in future trends.

The primary purpose of trend analysis is to determine if the improvements made to the products, services, or processes had an impact on the satisfaction level of the customers. It should be remembered, however, that satisfaction is a lagging indicator (it only provides information about what has already occurred). Customers have long memories; the dismal initial quality of a software version three releases back may still impact their perception of the product even if the last two versions have been superior.

Customer satisfaction is a subjective measure. It is a measure of perception, not reality, although when it comes to a happy customer, perception is more important than reality. One phenomenon that often occurs is that as the quality of software improves, the expectations of the customers also increase. The customers continue to demand bigger, better, faster software. This can result in a flat trend even though quality is continuously improving. Worse still, it can cause a declining graph because improvements to quality are not keeping up with the increases in the customer's expectations. Even though this impact can be discouraging, it is valuable information that the organization needs to know in the very competitive world of software.

Customer Impact—Responsiveness to Problem Reports

When a customer has a problem with software, the developer must respond quickly to resolve and close the reported problem. For example, service level agreements may define problem report response time goals based on the severity of the incident (for example, critical problems within 24 hours, major problems within 30 days, and minor problems within 120 days). The graph in Figure 19.15 illustrates an example of a metric to track actual performance against these service level agreements. This graph trends the percentage of problem reports closed within the service level agreement time frames each month. This metric is a lagging indicator, a view of the past. Using lagging indicator metrics is like painting



	1	2	3	4	5	6	7	8
Percent meeting goal	62	65	79	82	76	97	89	96
Number non-closed	1	3	2	1	4	2	1	0
Average age	22	34	48	30	27	38	23	0

Months since software implementation

Figure 19.15 Responsiveness to customer problems—example.

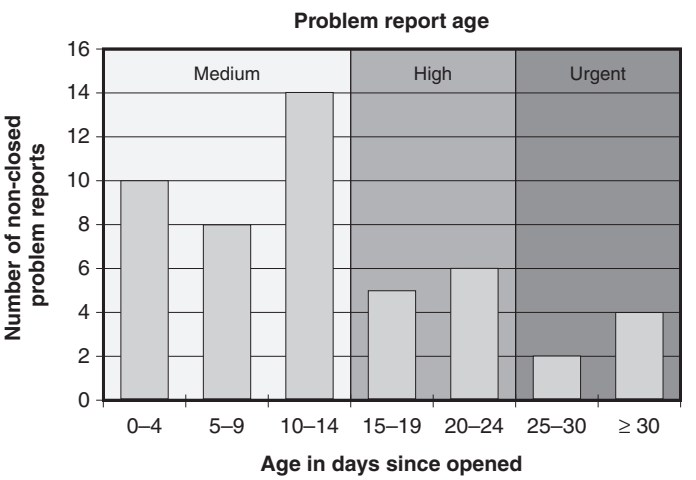


Figure 19.16 Defect backlog aging—example.

the front windshield of a car black and monitoring the quality of the driving by counting the dead bodies that can be seen in the rearview mirror. If an organization waits until the problem report is closed before tracking this response-time metric, they can not take proactive action to control their responsiveness.

The graph in Figure 19.16 shows all non-closed, major problem reports distributed by their age since they were opened. Analysis of this graph can quickly identify problem areas, including problem reports that are past their service level

agreement goal and reports approaching their goal. This information allows a proactive approach to controlling responsiveness to customer-reported problems. This metric is a leading indicator because it helps proactively identify issues. Similar graphs can also be created for minor problems (since critical problems must be corrected within 24 hours it is probably not cost-effective to track them with this metric).

Defect Detection

One way to measure the effectiveness of a defect detection process is to measure the number of *escapes* from that process. An analogy for escapes is to think of each defect detection process as a screen door that is catching bugs. Escapes are the bugs that make it through the screen door and get farther into the house. A person can not determine how effective the screen door is by looking at the bugs it caught; that person must go inside and count the number of bugs that got past the screen door.

In software there is no way of counting how many defects actually escaped a defect detection technique. The types of defects detected by subsequent defect detection techniques can be examined to approximate the number of actual escapes. For example, as illustrated in Figure 19.17a, at the end of the first defect detection technique (requirement defect detection—typically a requirement peer review) there are no known escapes.

However, as illustrated in Figure 19.17b, after analyzing the defects found by the second design defect detection technique, there are not only design-type defects (seven dark gray bugs) but also requirement-type defects (three light gray bugs), so three requirement escapes have been identified.

As illustrated in Figure 19.17c, after analyzing the defects found by the third coding defect detection technique, there are not only coding-type defects (seven black bugs) but also requirement-type defects (one light gray bug) and design-type

Requirement escapes | Design escapes | Code escapes

Figure 19.17a Measuring escapes—requirements example.

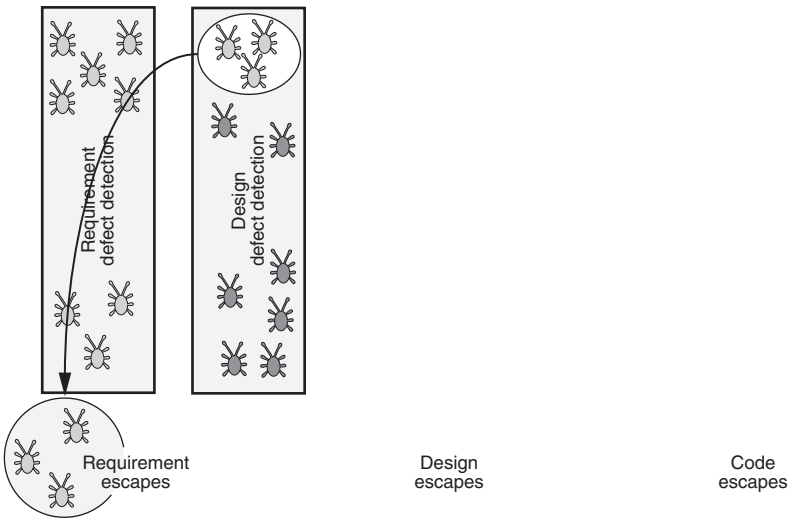


Figure 19.17b Measuring escapes—design example.

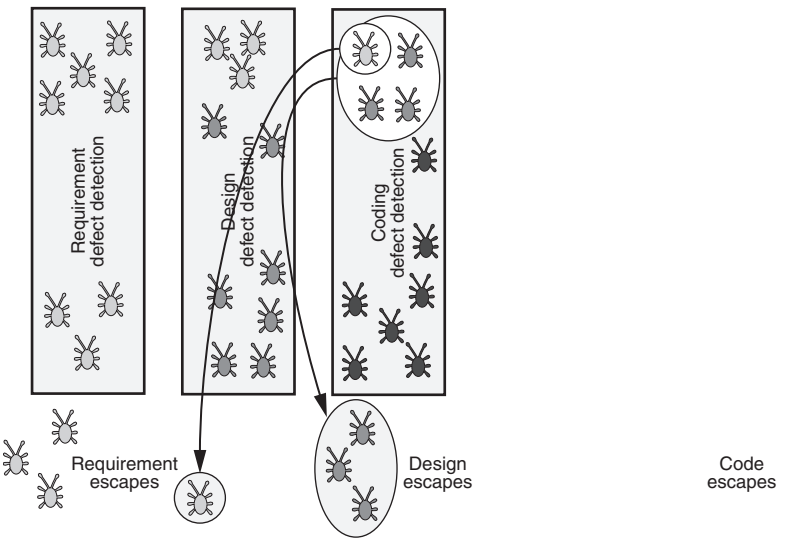


Figure 19.17c Measuring escapes—coding example.

defects (three dark gray bugs). So four requirement escapes and three design escapes have now been identified.

As illustrated in Figure 19.17d, everything found in the testing defect detection technique is an escape because no new defects are introduced through the testing process (new defects introduced during testing are the result of requirement, design, or code rework efforts). Analysis of defects found in testing shows requirement-type defects (two light gray bugs), design-type defects (five dark

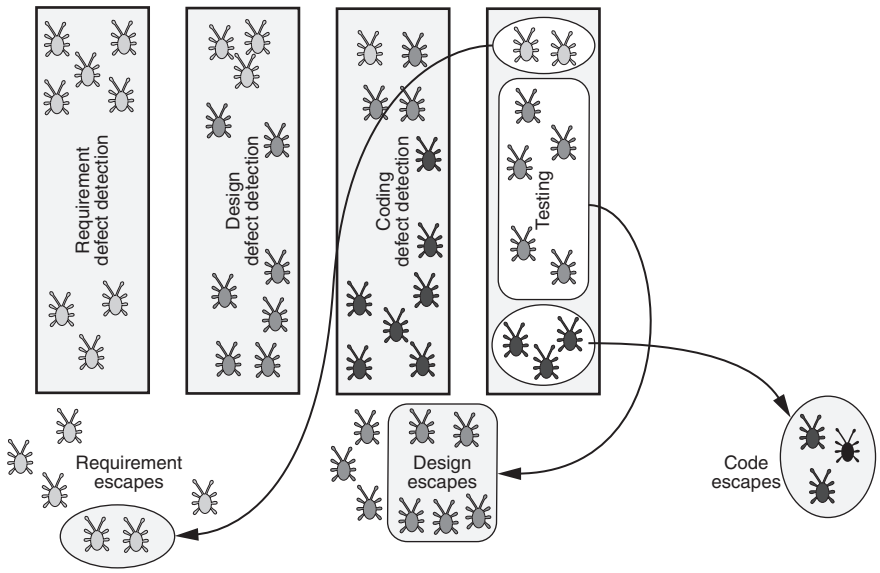


Figure 19.17d Measuring escapes—testing example.

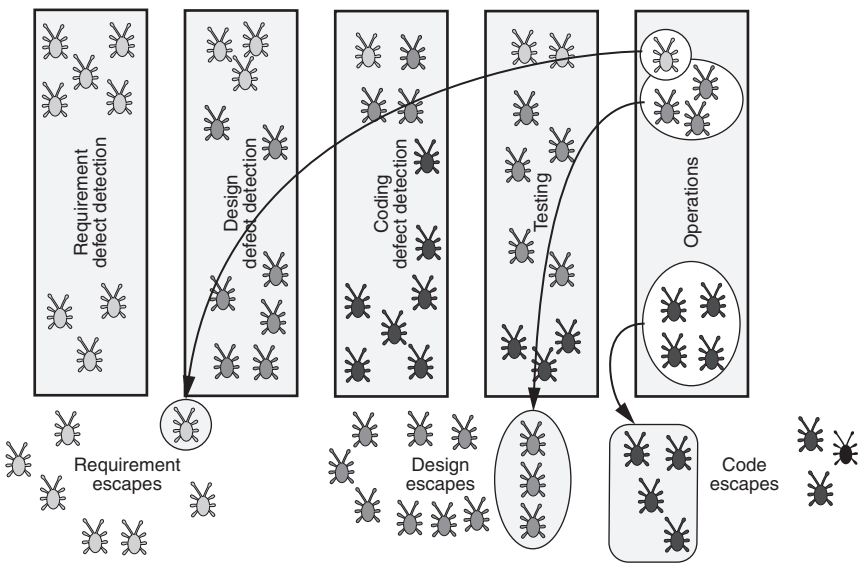


Figure 19.17e Measuring escapes—operations example.

gray bugs) and coding-type defects (three black bugs). There are now six requirement escapes, eight design escapes, and three coding escapes.

Finally, as illustrated in Figure 19.17e, everything found in operations is also an escape. Analysis of defects found in operations shows requirement-type defects (one light gray bug), design-type defects (three dark gray bugs), and coding-type

defects (four black bugs). There are now a total of seven requirement escapes, 11 design escapes, and seven coding escapes. Of course these counts will continue to change if additional defects are identified in operations.

Defect Containment and Total Defect Containment Effectiveness (TDCE)

Defect containment looks at the effectiveness of defect detection techniques to keep defects from escaping into later phases or into operations. *Phase containment effectiveness* metrics show the effectiveness of defect detection techniques in identifying defects in the same phase as they were introduced. Many studies have been done that demonstrate that defects that are not detected until later in the software development life cycle are much more costly to correct. By understanding which processes are allowing phase escapes, organizations can better target their defect detection improvement efforts. Phase containment is calculated by:

$$\frac{\text{Number of defects found that were introduced in the phase}}{\text{Total defects introduced during the phase (including those found later)}} \times 100\%$$

Figure 19.18 illustrates an example of calculating phase containment. For the requirements phase, 15 requirement-type defects were found and fixed during that phase, and 10 requirement-type defects were found in later phases for a total of 25 requirement-type defects. The requirements phase containment is 15 / 25 = 60%. For the design phase, 29 design-type defects were found and fixed during that phase, and 12 design-type defects were found in later phases for a total of 41 design-type defects. The design phase containment is 29 / 41 ≈ 71%. To continue this example, for the code phase, 86 code-type defects were found and fixed during that phase, and 26 code-type defects were found in later phases for a total of 112 code-type defects. The code phase containment is 86 / 112 ≈ 77%. Since the requirements phase containment percentage is the lowest, the requirements phase would be considered as a target for process improvement.

Phase detected	Phase detected defect was introduced		
	Requirements	Design	Code
Requirements	15		
Design	5	29	
Code	1	7	86
Test	3	3	19
Field	1	2	7
Total	25	41	112

Requirements: 15 / 25 = 60%
Design: 29 / 41 ≈ 71%
Code: __ / __ = __%
Prerelease: __ / __ = __%

Figure 19.18 Defect containment effectiveness—example.

While this example shows the phase containment metrics being calculated after the software has been in operations for some period of time—in real use—phase metrics should be calculated after each phase. At the end of the requirements phase, the requirement phase containment is 15/15 = 100%. At the end of the design phase, requirement phase containment is 15/20 = 75%. These ongoing calculations can be compared with average values baselined from other projects to determine if corrective action is necessary at any time. For example, if instead of five requirement-type defects being found in the design phase, assume that 30 requirement-type defects were found. In this case, the requirements phase containment would be calculated as 25/55 ≈ 45%. This much lower value might indicate that corrective action in terms of additional requirements defect detection activities should be performed before proceeding into the code phase.

The *total defect containment effectiveness* (TDCE) metric shows the effectiveness of defect detection techniques in identifying defects before the product is released into operation. TDCE is calculated as:

$$\frac{\text{Number of defects found prior to release}}{\text{Total defects found (including those found after release)}} \times 100\%$$

For the example in Figure 19.18, 24 requirement-type defects, 39 design-type defects, and 105 code-type defects, for a total of (24 + 39 + 105) = 168 defects, were found prior to release. Total defects found (including those found after release) is (25 + 41 + 112) = 178. TDCE for this example is 168/178 ≈ 94%. The TDCE for this project could then be compared with the TDCE for previous projects to determine if it is at an appropriate level. If process improvements have been implemented, this comparison can be used to determine if the improvements had a positive impact on the TDCE metric value.

Defect Removal Efficiency

Defect removal efficiency (DRE), also called *defect detection efficiency* or *defect detection effectiveness,* is a measure of the percentage of all defects in the software that were found and removed when a detection/rework process was executed. Unlike phase containment, this metric can be calculated for each defect detection technique (instead of by phase). For example, if both code reviews and unit testing were done in the coding phase, each technique would have its own DRE percentage. DRE also includes all defects that could have been found by that technique, not just the ones introduced during that phase. DRE is calculated by:

$$\frac{\text{Number of defects found and removed by the activity}}{\text{Total number of defects present at the activity}} \times 100\%$$

Figure 19.19a illustrates an example of the calculation of DRE. Note that the DRE for the requirements review process is 15/25 = 60%. Since this is the first defect detection/removal process and the only types of defects available to find are requirement-type defects, then the phase containment and DRE numbers are the same. However, in the design review processes, not only can design-type defects be found, but the defects that escaped from the requirements review can also be

found. As illustrated in Figure 19.19a, there were five requirement-type defects and 29 design-type defects found during the design review. However, an additional five requirement-type defects and 12 design-type defects escaped and were found later. The DRE for the design review is (5 + 29) / (5 + 29 + 5 + 12) ≈ 67%.

As illustrated in Figure 19.19b, the DRE for the code review is approximately equal to (1 + 3 + 54) / (1 + 3 + 54 + 4 + 9 + 58) ≈45%. To complete this example, the DRE for:

- Unit testing = (0 + 4 + 32) / (0 + 4 + 32 + 4 + 5 + 26) ≈51%
- Integration testing = (1 + 3 + 13) / (1 + 3 + 13 + 3 + 2 + 13) ≈49%
- System testing = (2 + 0 + 6) / (2 + 0 + 6 + 1 + 2 + 7) ≈44%

Detection technique	Phase detected defect was introduced		
	Requirements	Design	Code
Requirements review	15		
Design review	5	29	
Code review	1	3	54
Unit testing	0	4	32
Integration testing	1	3	13
System testing	2	0	6
Operations	1	2	7
Total	25	41	112

Design review: (5 + 29) / (5 + 29 + 5 + 12) = 34 / 51 ≈ 67%

Figure 19.19a Defect removal efficiency—design review example.

Detection technique	Phase detected defect was introduced		
	Requirements	Design	Code
Requirements review	15		
Design review	5	29	
Code review	1	3	54
Unit testing	0	4	32
Integration testing	1	3	13
System testing	2	0	6
Operations	1	2	7
Total	25	41	112

Code review: (1 + 3 + 54) / (1 + 3 + 54 + 4 + 9 + 58) = 58 / 129 ≈ 45%

Figure 19.19b Defect removal efficiency—code review example.