10

In-Process Metrics for Software Testing

In Chapter 9 we discussed quality management models with examples of in-process metrics and reports. The models cover both the front-end design and coding activities and the back-end testing phases of development. The focus of the in-process data and reports, however, are geared toward the design review and code inspection data, although testing data is included. This chapter provides a more detailed discussion of the in-process metrics from the testing perspective.[1] These metrics have been used in the IBM Rochester software development laboratory for some years with continual evolution and improvement, so there is ample implementation experience with them. This is important because although there are numerous metrics for software testing, and new ones being proposed frequently, relatively few are supported by sufficient experiences of industry implementation to demonstrate their usefulness. For each metric, we discuss its purpose, data, interpretation, and use, and provide a graphic example based on real-life data. Then we discuss in-process quality management vis-à-vis these metrics and revisit the metrics

1. This chapter is a modified version of a white paper written for the IBM corporate-wide Software Test Community Leaders (STCL) group, which was published as "In-process Metrics for Software Testing," in *IBM Systems Journal*, Vol. 40, No.1, February 2001, by S. H. Kan, J. Parrish, and D. Manlove. Copyright © 2001 International Business Machines Corporation. Permission to reprint obtained from *IBM Systems Journal*.

framework, the *effort/outcome model*, again with sufficient details on testing-related metrics. Then we discuss some possible metrics for a special test scenario, acceptance test with regard to vendor-developed code, based on the experiences from the IBM 2000 Sydney Olympics project by Bassin and associates (2002). Before we conclude the chapter, we discuss the pertinent question: *How do you know your product is good enough to ship?*

Because the examples are based on IBM Rochester's experiences, it would be useful to outline IBM Rochester's software test process as the context, for those who are interested. The accompanying box provides a brief description.

10.1 In-Process Metrics for Software Testing

In this section, we discuss the key in-process metrics that are effective for managing software testing and the in-process quality status of the project.

10.1.1 Test Progress S Curve (Planned, Attempted, Actual)

Tracking the progress of testing is perhaps the most important tracking task for managing software testing. The metric we recommend is a test progress S curve over time. The *X*-axis of the S curve represents time units and the *Y*-axis represents the number of test cases or test points. By "S curve" we mean that the data are cumulative over time and resemble an "S" shape as a result of the period of intense test activity, causing a steep planned test ramp-up. For the metric to be useful, it should contain the following information on one graph:

- ☐ Planned progress over time in terms of number of test cases or number of test points to be completed successfully by week (or other time unit such as day or hour)
- ☐ Number of test cases attempted by week (or other time unit)
- ☐ Number of test cases completed successfully by week (or other time unit)

The purpose of this metric is to track test progress and compare it to the plan, and therefore be able to take action upon early indications that testing activity is falling behind. It is well known that when the schedule is under pressure, testing, especially development testing, is affected most significantly. Schedule slippage occurs day by day and week by week. With a formal test progress metric in place, it is much more difficult for the team to ignore the problem. From the project planning perspective, an S curve forces better planning (see further discussion in the following paragraphs).

Figure 10.2 is an example of the component test metric at the end of the test of a major release of an integrated operating system. As can be seen from the figure, the testing plan is expressed in terms of a line curve, which is put in place before the test

IBM Rochester's Software Test Process

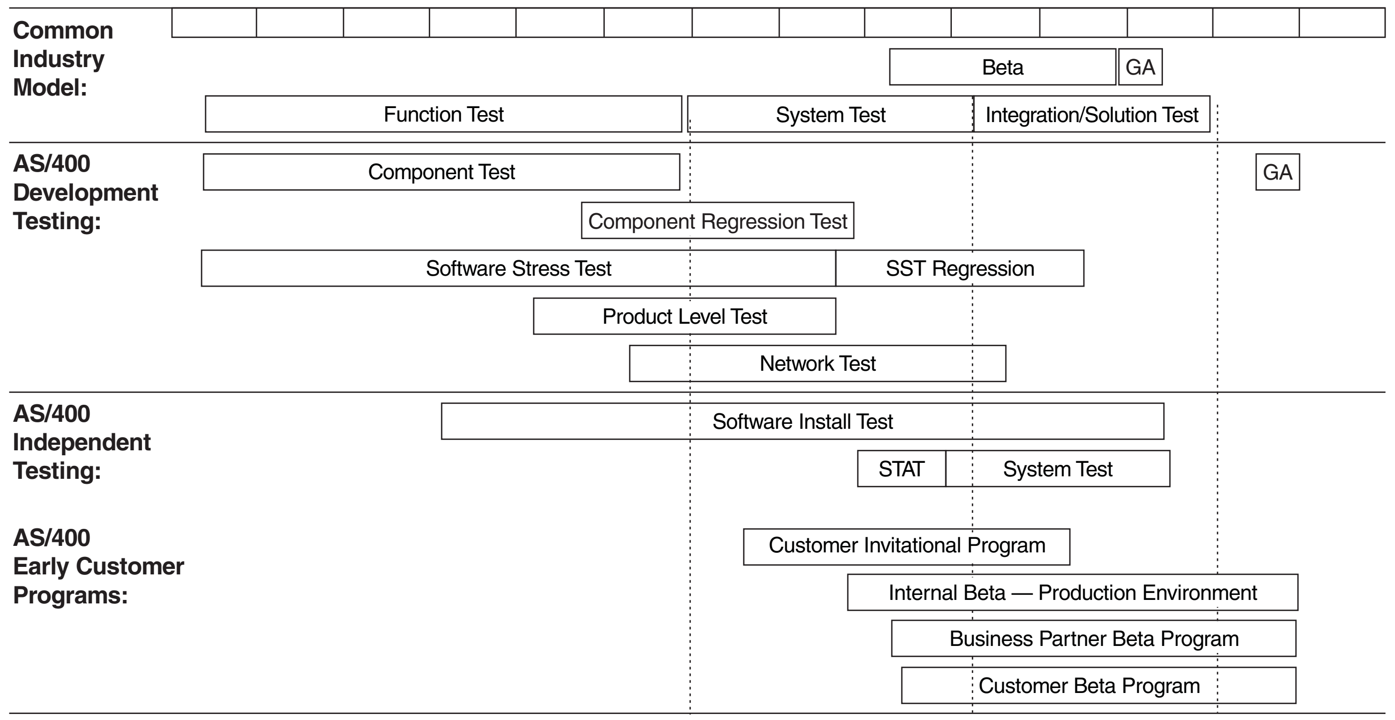
IBM Rochester's systems software development process has a strong focus on the front-end phases such as requirements, architecture, design and design verification, code integration quality, and driver builds. For example, the completion of high-level design review (I0) is always a key event in the system schedule and managed as an intermediate deliverable. At the same time, testing (development tests and independent tests) and customer validation are the key process phases with equally strong focus. As Figure 10.1 shows, the common industry model of testing includes functional test, system test, and customer beta test before the product is shipped. Integration and solution testing can occur before or after the product ships. It is often conducted by customers because the customer's integrated solution may consist of products from different vendors. For IBM Rochester, the first test phase after unit testing and code integration into the system library consists of component test (CT) and component regression test (CRT), which is equivalent to functional test. The next test phase is system test (ST), which is conducted by an independent test group. To ensure entry criteria is met, an acceptance test (STAT) is conducted before system test start. The main path of the test process is from CT → CTR → STAT → ST. Parallel to the main path are several development and independent tests:

- Along with component test, a stress test is conducted in a large network environment with performance workload running in the background to stress the system.
- When significant progress is made in component test, a product-level test (PLT), which focuses on the subsystems of an overall integrated software system (e.g., database, client access, clustering), starts.
- The network test is a specific product-level test focusing on communications subsystems and related error recovery processes.
- The independent test group also conducts a software installation test, which runs from the middle of the component test until the end of the system test.

The component test and the component regression test are done by the development teams. The stress test, the product-level test, and the network test are done by development teams in special test environments maintained by the independent test group. The install and system tests are conducted by the independent test team. Each of these different tests plays an important role in contributing to the high quality of an integrated software system for the IBM eServer iSeries and AS/400 computer system. Later in this chapter, another shaded box provides an overview of the system test and its workload characteristics.

As Figure 10.1 shows, several early customer programs occur at the back end of the development process:

- *Customer invitational program:* Selected customer invited to the development laboratory to test the new functions and latest technologies. This is done when component and component regression tests are near completion.
- *Internal beta:* The development site uses the latest release for its IT production operations (i.e., eating one's own cooking)
- *Beta program with business partners*
- *Customer beta program*


Common Industry Model:
Beta
GA
Function Test
System Test
Integration/Solution Test
AS/400 Development Testing:
Component Test
GA
Component Regression Test
Software Stress Test
SST Regression
Product Level Test
Network Test
AS/400 Independent Testing:
Software Install Test
STAT
System Test
AS/400 Early Customer Programs:
Customer Invitational Program
Internal Beta — Production Environment
Business Partner Beta Program
Customer Beta Program

GA: General Availability = product ship
STAT: System Test Acceptance Test
SST: Software Stress Test

FIGURE 10.1
IBM Rochester's Software Testing Phases

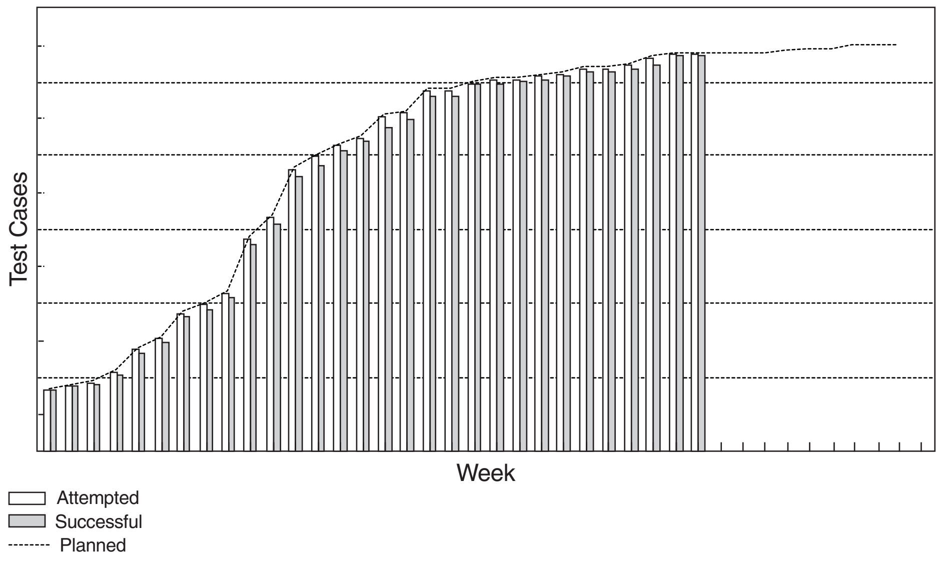


FIGURE 10.2
Sample Test Progress S Curve

begins. The empty bars indicate the cumulative number of test cases attempted and the solid bars represent the number of successful test cases. With the plan curve in place, each week when the test is in progress, two bars (one for attempted and one for successful) are added to the graph. This example shows that during the rapid test ramp-up period (the steep slope of the curve), for some weeks the test cases attempted were slightly ahead of plan (which is possible), and the successes were slightly behind plan.

Because some test cases are more important than others, it is not unusual in software testing to assign scores to the test cases. Using test scores is a normalization approach that provides more accurate tracking of test progress. The assignment of scores or points is normally based on experience, and at IBM Rochester, teams usually use a 10-point scale (10 for the most important test cases and 1 for the least). To track test points, the teams need to express the test plan (amount of testing done every week) and track the week-by-week progress in terms of test points. The example in Figure 10.3 shows test point tracking for a product level test, which was underway, for a systems software. It is noted that there is always an element of subjectivity in the assignment of weights. The weights and the resulting test points should be determined in the test planning stage and remain unchanged during the testing process. Otherwise, the purpose of this metric will be compromised in the reality of schedule pressures. In software engineering, weighting and test score assignment

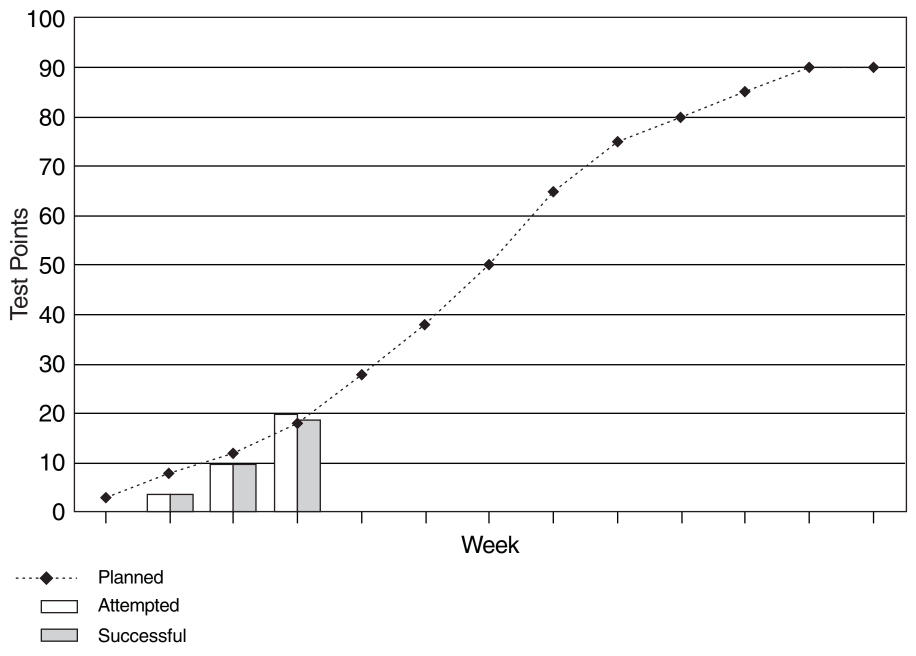


FIGURE 10.3
Test Progress S Curve—Test Points Tracking

remains an interesting area where more research is needed. Possible guidelines from such research will surely benefit the planning and management of software testing.

For tracking purposes, test progress can also be weighted by some measurement of coverage. Coverage weighting and test score assignment consistency become increasingly important in proportion to the number of development groups involved in a project. Lack of attention to tracking consistency across functional areas can result in a misleading view of the overall project's progress.

When a plan curve is in place, the team can set up an in-process target to reduce the risk of schedule slippage. For instance, a disparity target of 15% between attempted (or successful) and planned can be used to trigger additional actions. Although the test progress S curves, as shown in Figures 10.2 and 10.3, give a quick visual status of the progress against the total plan and plan-to-date (the eye can quickly determine if testing is ahead or behind on planned attempts and successes), it may be difficult to discern the exact amount of slippage. This is particularly true for large testing efforts, where the number of test cases is in the hundreds of thousands. For that reason, it is useful to also display the test status in tabular form, as in Table 10.1. The table also shows underlying data broken out by department and

TABLE 10.1
Test Progress Tracking—Planned, Attempted, Successful

	No. of Test Cases Planned to Date	Percent of Plan Attempted	Percent of Plan Successful	No. of Planned Test Cases Not Yet Attempted	Percent of Total Attempted	Percent of Total Successful
System	60577	90.19	87.72	5940	68.27	66.10
Dept A	1043	66.83	28.19	346	38.83	15.60
Dept B	708	87.29	84.46	90	33.68	32.59
Dept C	33521	87.72	85.59	4118	70.60	68.88
Dept D	11275	96.25	95.25	423	80.32	78.53
Dept E	1780	98.03	94.49	35	52.48	50.04
Dept F	4902	100.00	99.41	0	96.95	95.93
Product A	13000	70.45	65.10	3841	53.88	49.70
Product B	3976	89.51	89.19	417	66.82	66.50
Product C	1175	66.98	65.62	388	32.12	31.40
Product D	277	0	0	277	0	0
Product E	232	6.47	6.470	214	3.78	3.70

product or component, which helps to identify problem areas. In some cases, the overall test curve may appear to be on schedule, but when progress is viewed only at the system level, because some areas are ahead of schedule, they may mask areas that are behind schedule. Of course, test progress S curves are also used for functional areas and for specific products.

An initial plan curve should be subject to brainstorming and challenges. For example, if the curve shows a very steep ramp-up in a short period of time, the project manager may challenge the team with respect to how doable the plan is or the team's specific planned actions to execute the plan successfully. As a result, better planning will be achieved. *Caution:* Before the team settles on a plan curve and uses it to track progress, a critical evaluation of what the plan curve represents must be made. Is the total test suite considered effective? Does the plan curve represent high test coverage (functional coverage)? What are the rationales for the sequences of test cases in the plan? This type of evaluation is important because once the plan curve is in place, the visibility of this metric tends to draw the whole team's attention to the disparity between attempted, successful, and the planned testing.

Once the plan line is set, any proposed or actual changes to the plan should be reviewed. Plan slips should be evaluated against the project schedule. In general, the baseline plan curve should be maintained as a reference. Ongoing changes to the planned testing schedule can mask schedule slips by indicating that attempts are on track, while the plan curve is actually moving to the right.

In addition, this metric can be used for release-to-release or project-to-project comparisons, as the example in Figure 10.4 shows. For release-to-release comparisons, it is important to use time units (weeks or days) before product ship (or general availability, GA) as the unit for the *X*-axis. By referencing the ship dates, the comparison provides a true status of the project in process. In Figure 10.4, it can be observed that Release B, represented by the dotted line, is more back-end loaded than Release A, which is represented by the solid line. In this context, the metric is both a quality and a schedule statement for the testing of the project. This is because late testing causes late cycle defect arrivals and therefore negatively affects the quality of the final product. With this type of comparison, the project team can plan ahead (even before the testing starts) to mitigate the risks.

To implement this metric, the test execution plan needs to be laid out in terms of the weekly target, and actual data needs to be tracked on a weekly basis. For small to medium projects, such planning and tracking activities can use common tools such as Lotus 1-2-3 or other project management tools. For large and complex projects, a stronger tools support facility normally associated with the development environment may be needed. Many software tools are available for project management and quality control, including tools for defect tracking and defect projections. Testing tools usually include test library tools for keeping track of test cases and for test automation, test coverage analysis tools, test progress tracking, and defect tracking tools.

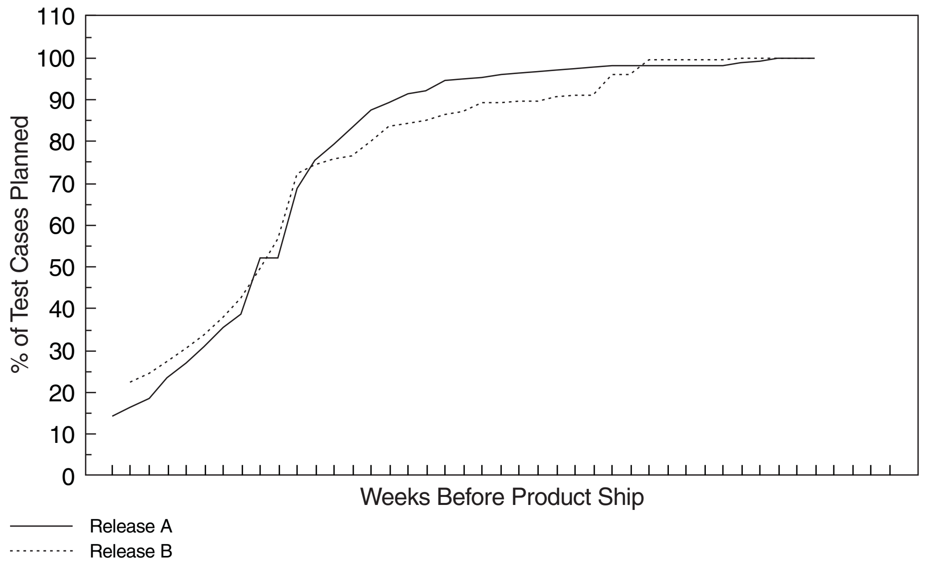


FIGURE 10.4
Test Plan Curve—Release-to-Release Comparison

10.1.2 Testing Defect Arrivals over Time

Defect tracking and management during the testing phase is highly recommended as a standard practice for all software testing. Tracking testing progress and defects are common features of many testing tools. At IBM Rochester, defect tracking is done via the problem tracking report (PTR) tool. We have discussed PTR-related models and reports previously. In this chapter we revisit two testing defect metrics (arrivals and backlog) with more details. We recommend tracking the defect arrival pattern over time, in addition to tracking by test phase. Overall defect density during testing, or for a particular test, is a summary indicator, but not really an in-process indicator. The pattern of defect arrivals over time gives more information. As discussed in Chapter 4 (section 4.2.2), even with the same overall defect rate during testing, different patterns of defect arrivals may imply different scenarios of field quality. We recommend the following for this metric:

- ☐ Always include data for a comparable baseline (a prior release, a similar project, or a model curve) in the chart if such data is available. If a baseline is not available, at the minimum, when tracking starts, set some expected level of defect arrivals at key points of the project schedule (e.g., midpoint of functional test, system test entry, etc.).

- The unit for the *X*-axis is weeks (or other time units) before product ship
- The unit for the *Y*-axis is the number of defect arrivals for the week, or its variants.

Figure 10.5 is an example of this metric for releases of an integrated operating system. For this example, the main goal is release-to-release comparison at the system level. The metric can be used for the defect arrival patterns based on the total number of defects from all test phases, and for defect arrivals for specific tests. It can be used to compare actual data with a PTR arrival model, as discussed in Chapter 9.

Figure 10.5 has been simplified for presentation. The real graph has much more information on it including vertical lines to depict the key dates of the development cycle and system schedules such as last new function integration, development test completion, start of system test, and so forth. There are also variations of the metric: total defect arrivals, severe defects (e.g., severity 1 and 2 defects in a 4-point severity scale), defects normalized to size of the release (new and changed code plus a partial weight for ported code), and total defect arrivals versus valid defects. The main, and the most useful, chart is the total number of defect arrivals. In our projects, we also include a high severity (severity 1 and 2) defect chart and a normalized view as mainstays of tracking. The normalized defect arrival chart can eliminate some of the

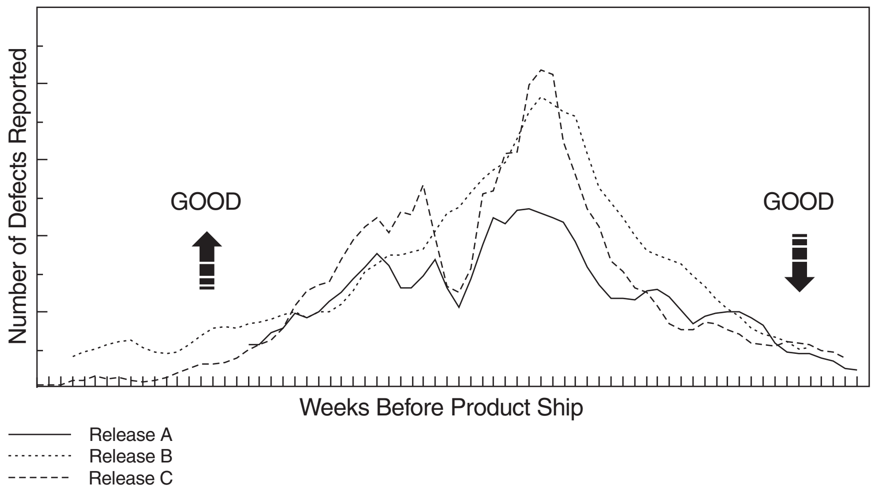


FIGURE 10.5
Testing Defect Arrival Metric

visual guesswork of comparing current progress to historical data. In conjunction with the severity chart, a chart that displays the percentage of severity 1 and 2 PTRs per week can be useful. As Figure 10.6 shows, the percentage of high severity problems increases as the release progresses toward the product ship date. Generally, this is because the urgency for problem resolution increases when approaching product delivery, therefore, the severity of the defects was elevated. Unusual swings in the percentage of high severity problems, however, could signal serious problems and should be investigated.

When do the defect arrivals peak relative to time to product delivery? How does this pattern compare to previous releases? How high do they peak? Do they decline to a low and stable level before delivery? Questions such as these are key to the defect arrival metric, which has significant quality implications for the product in the field. A positive pattern of defect arrivals is one with higher arrivals earlier, an earlier peak (relative to the baseline), and a decline to a lower level earlier before the product ship date, or one that is consistently lower than the baseline when it is certain that the effectiveness of testing is at least as good as previous testing. The tail end of

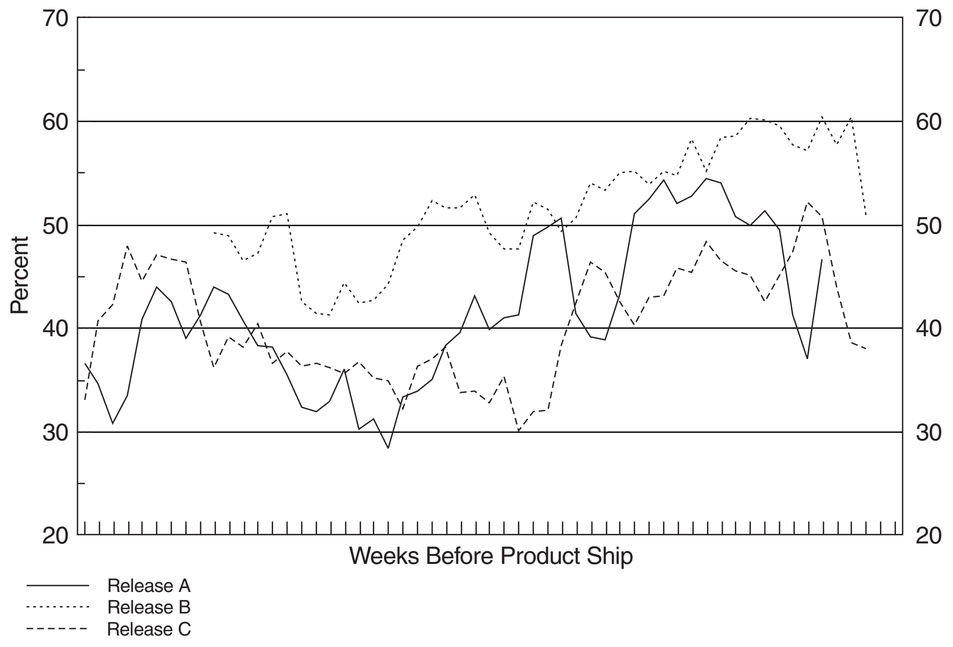


FIGURE 10.6
Testing Defect Arrivals—Percentage of Severity 1 and 2 Defects

the curve is especially important because it is indicative of the quality of the product in the field. High defect activity before product delivery is more often than not a sign of quality problems. To interpret the defect arrivals metrics properly, refer to the scenarios and questions discussed in Chapter 4 section 4.2.1.

In addition to being an important in-process metric, the defect arrival pattern is the data source for projection of defects in the field. If we change from the weekly defect arrival curve (a density form of the metric) to a cumulative defect curve (a cumulative distribution form of the metric), the curve becomes a well-known form of the software reliability growth pattern. Specific reliability models, such as those discussed in Chapters 8 and 9, can be applied to the data to project the number of residual defects in the product. Figure 10.7 shows such an example. The actual testing defect data represents the total cumulative defects removed when all testing is complete. The fitted model curve is a Weibull distribution with the shape parameter (m) being 1.8. The projected latent defects in the field is the difference in the *Y*-axis of the model curve between the product ship date and when the curve is approaching its limit. If there is a time difference between the end date of testing and the product ship date, such as this case, the number of latent defects represented by the section of the model curve for this time segment has to be included in the projected number of defects in the field.

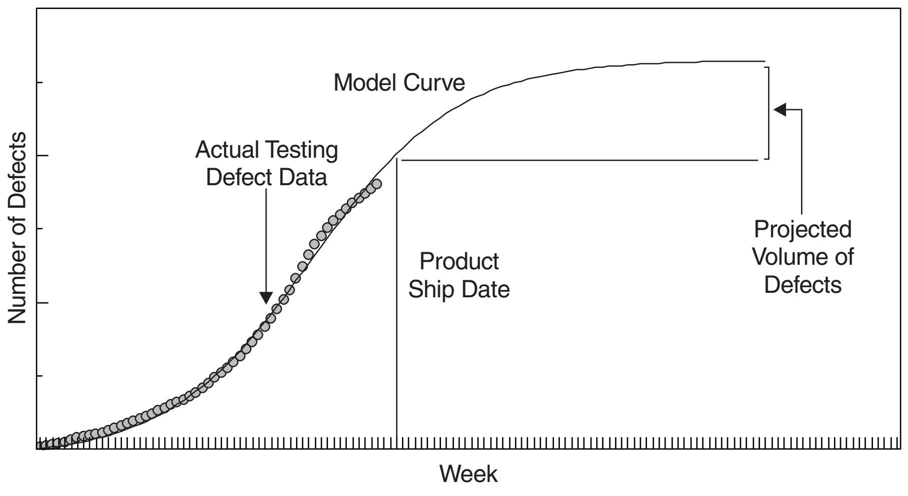


FIGURE 10.7
Testing Defect Arrival Curve, Software Reliability Growth Model, and Defect Projection

10.1.3 Testing Defect Backlog over Time

We define the number of testing defects (or problem tracking reports, PTRs) remaining at any given time as the defect backlog (PTR backlog). Simply put, defect backlog is the accumulated difference between defect arrivals and defects that were closed. Defect backlog tracking and management is important from the perspective of both test progress and customer rediscoveries. A large number of outstanding defects during the development cycle will impede test progress. When a product is about to ship to customers, a high defect backlog means more customer rediscoveries of the defects already found during the development cycle. For software organizations that have separate teams to conduct development testing and to fix defects, defects in the backlog should be kept at the lowest possible level at all times. For organizations that have the same teams responsible for development testing and fixing defects, however, there are appropriate timing windows in the development cycle for which the priority of focuses may vary. While the defect backlog should be managed at a reasonable level at all times, it should not be the highest priority during a period when making headway in functional testing is the critical-path development activity. During the prime time for development testing, the focus should be on test effectiveness and test execution, and defect discovery should be encouraged to the maximum possible extent. Focusing too early on overall defect backlog reduction may conflict with these objectives. For example, the development team may be inclined not to open defect records. The focus during this time should be on the fix turnaround of the critical defects that impede test progress instead of the entire backlog. Of course, when testing is approaching completion, strong focus for drastic reduction in the defect backlog should take place.

For software development projects that build on existing systems, a large backlog of "aged" problems can develop over time. These aged defects often represent fixes or enhancements that developers believe would legitimately improve the product, but which get passed over during development due to resource or design constraints. They may also represent problems that have been fixed or are obsolete as a result of other changes. Without a concerted effort, this aged backlog can build over time. This is one area of the defect backlog that warrants attention early in the development cycle, even prior to the start of development testing.

Figure 10.8 is an example of the defect backlog metric for several releases of a systems software product. Again, release-to-release comparisons and actual data versus targets are the main objectives. Target X was a point target for a specific event in the project schedule. Target Y was for the period when the product was being readied to ship.

Note that for this metric, a sole focus on the numbers is not sufficient. In addition to the overall reduction, deciding which specific defects should be fixed first is very

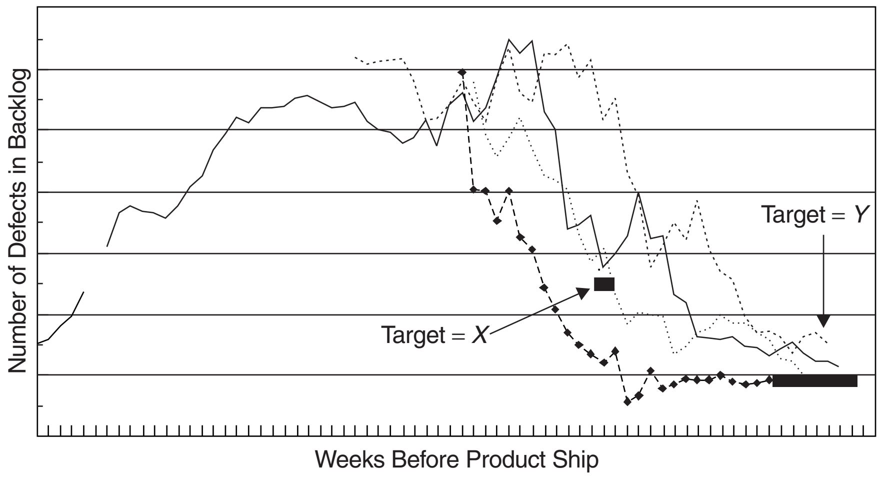


FIGURE 10.8
Testing Defect Backlog Tracking

important in terms of achieving early system stability. In this regard, the expertise and ownership of the development and test teams are crucial.

Unlike defect arrivals, which should not be controlled artificially, the defect backlog is completely under the control of the development organization. For the three metrics we have discussed so far, we recommend the following overall project management approach:

- When a test plan is in place and its effectiveness evaluated and accepted, manage test progress to achieve an early ramp-up in the S curve.
- Monitor defect arrivals and analyze the problems (e.g., defect cause analysis and Pareto analysis of problem areas of the product) to gain knowledge for improvement actions. *Do not* artificially control defect arrivals, which is a function of test effectiveness, test progress, and the intrinsic quality of the code (the amount of latent defects in the code). *Do* encourage opening defect records when defects are found.
- Strongly manage defect backlog reduction and achieve predetermined targets associated with the fix integration dates in the project schedule. Known defects that impede testing progress should be accorded the highest priority.

The three metrics discussed so far are obviously related, and they should be viewed together. We'll come back to this point in the section on the effort/outcome model.

10.1.4 Product Size over Time

Lines of code or another indicator of the project size that is meaningful to the development team can also be tracked as a gauge of the "effort" side of the development equation. During product development, there is a tendency toward growth as requirements and designs are fleshed out. Functions may continue to be added to meet late requirements or the development team wants more enhancements. A project size indicator, tracked over time, can serve as an explanatory factor for test progress, defect arrivals, and defect backlog. It can also relate the measurement of total defect volume to per unit improvement or deterioration. Figure 10.9 shows a project's release size pattern with rapid growth during release definition, stabilization, and then possibly a slight reduction in size toward release completion, as functions that fail to meet schedule or quality objectives are deferred. In the figure, the different segments in the bars represent the different layers in the software system. This metric is also known as an indicator of scope creep. Note that lines of code is only one of the size indicators. The number of function points is another common indicator, especially in application software. We have also seen the number of bytes of memory that the software will use as the size indicator for projects with embedded software.

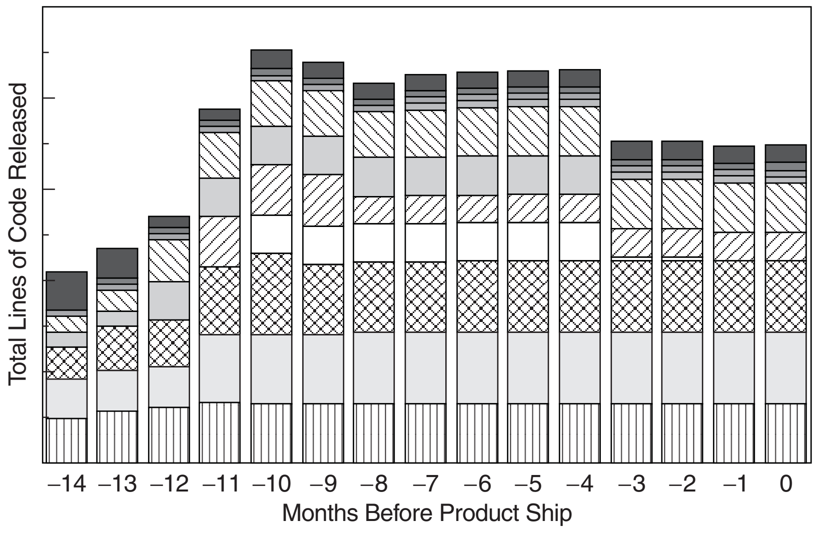


FIGURE 10.9
Lines of Code Tracking over Time

10.1.5 CPU Utilization During Test

For computer systems or software products for which a high level of stability is required to meet customers' needs, it is important that the product perform well under stress. In software testing during the development process, the level of CPU utilization is an indicator of the system's stress.

To ensure that its software testing is effective, the IBM Rochester software development laboratory sets CPU utilization targets for the software stress test and the system test. Stress testing starts at the middle of the component test phase and may run into the system test time frame with the purpose of stressing the system in order to uncover latent defects that cause system crashes and hangs that are not easily discovered in normal testing environments. It is conducted with a network of systems. System test is the final test phase with a customerlike environment. Test environment, workload characteristics, and CPU stress level are major factors contributing to the effectiveness of the test. The accompanying box provides an overview of the IBM Rochester system test and its workload characteristics.

System Test Overview and Workload Characteristics

IBM Rochester's system test serves as a means to provide a predelivery readiness assessment of the product's ability to be installed and operated in customerlike environments. These test environments focus on the total solution, including current release of the operating system, new and existing hardware, and customerlike applications. The resulting test scenarios are written to exercise the operating system and related products in a manner similar to customers' businesses. These simulated environments do not attempt to replicate a particular customer, but represent a composite of customer types in the target market.

The model used for simulating customerlike environments is referred to as the RAISE (Reliability, Availability, Installability, Serviceability, and Ease of use) environment. It is designed to represent an interrelated set of companies that use the IBM products to support and drive their day-to-day business activities. Test scenarios are defined to simulate the different types of end-user activities, workflow, and business applications. They include CPU-intensive applications and interaction-intensive computing. During test execution, the environment is run as a 24-hour-a-day, 7-day-a-week (24x7) operation.

Initially, work items are defined to address complete solutions in the RAISE environment. From these work items come more detailed scenario definitions. These scenarios are written to run in the respective test environment, performing a sequence of tasks and executing a set of test applications to depict some customerlike event. Scenario variations are used to cater test effort to different workloads, operating environments, and run-time duration. The resulting interaction of multiple scenarios executing across a network of systems provides a representation of real end-user environments. This provides an assessment of the overall functionality in the release, especially in terms of customer solutions.

Some areas that scenario testing concentrates on include:

- Compatibility of multiple products running together
- Integration and interoperability of products across a complex network
- Coexistence of multiple products on one hardware platform
- Areas of potential customer dissatisfaction:
  - –Unacceptable performance
  - –Unsatisfactory installation
  - –Migration/upgrade difficulties
  - –Incorrect and/or difficult-to-use documentation
  - –Overall system usability

As is the case for many customers, most system test activities require more than one system to execute. This fact is essential to understand, from both product integration and usage standpoints, and also because this represents a more realistic, customerlike setup. In driving multiple, interrelated, and concurrent activities across our network, we tend to "shake out" those hard-to-get-at latent problems. In such a complex environment, these types of problems tend to be difficult to analyze, debug, and fix, because of the layers of activities and products used. Additional effort to fix these problems is time well spent, because many of them could easily become critical situations to customers.

Workloads for the RAISE test environments are defined to place an emphasis on stressful, concurrent product interaction. Workload characteristics include:

- Stressing some of the more complex new features of the system
- Running automated tests to provide background workload for additional concurrence and stress testing and to test previous release function for regression
- Verifying that the software installation instructions are accurate and understandable and that the installation function works properly
- Testing release-to-release compatibility, including *n* to *n*–1 communications connectivity and system interoperability
- Detecting data conversion problems by simulating customers performing installations from a prior release
- Testing availability and recovery functions
- Artistic testing involving disaster and error recovery
- Performing policy-driven system maintenance (e.g., backup, recovery, and applying fixes)
- Defining and managing different security levels for systems, applications, documents, files, and user/group profiles
- Using the tools and publications that are available to the customer or IBM service personnel when diagnosing and resolving problems

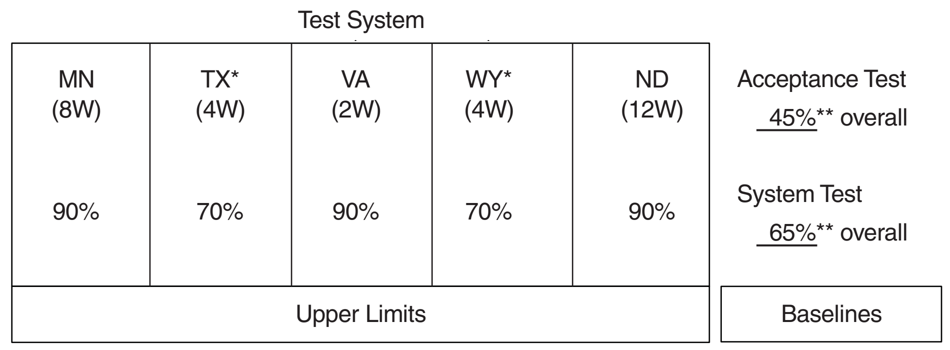
Another objective during the RAISE system test is to maintain customer environment systems at stable hardware and software levels for an extended time (one month or more). A guideline for this would be minimum number of unplanned initial program loads (IPL, or reboot) except for maintenance requiring an IPL. The intent is to simulate an active business and detect problems that occur only after the systems and network have been operating for an extended, uninterrupted period of time.

The data in Figure 10.10 indicate the recent CPU utilization targets for the IBM Rochester's system test. Of the five systems in the system test environment, there is one system with a 2-way processor (VA), two systems with 4-way processors (TX and WY), and one system each with 8-way and 12-way processors. The upper CPU utilization limits for TX and WY are much lower because these two systems are used for interactive processing. For the overall testing network, the baseline targets for system test and the acceptance test of system test are also shown.

The next example, shown in Figure 10.11, demonstrates the tracking of CPU utilization over time for the software stress test. There is a two-phase target as represented by the step-line in the chart. The original target was set at 16 CPU hours per system per day on the average, with the following rationale:

- The stress test runs 20 hours per day, with 4 hours of system maintenance.
- The CPU utilization target is 80% or higher.

The second phase of the target, set at 18 CPU hours per system per day, is for the back end of the stress test. As the figure shows, a key element of this metric, in addition to comparison of actual and target data, is release-to-release comparison. One can observe that the curve for release C had more data points in the early development cycle, which were at higher CPU utilization levels. This is because pretest runs were conducted prior to availability of the new release content. For all three releases, the CPU utilization metric shows an increasing trend with the stress test progress.



Test System

MN (8W)	TX* (4W)	VA (2W)	WY* (4W)	ND (12W)	Acceptance Test 45%** overall
90%	70%	90%	70%	90%	System Test 65%** overall
Upper Limits					Baselines

* Priorities set for interactive user response time; 70 percent seems to be the upper limit based on prior release testing.
** Average minimum needed to meet test case and system aging requirements.

FIGURE 10.10
CPU Utilization Targets for Testing Systems

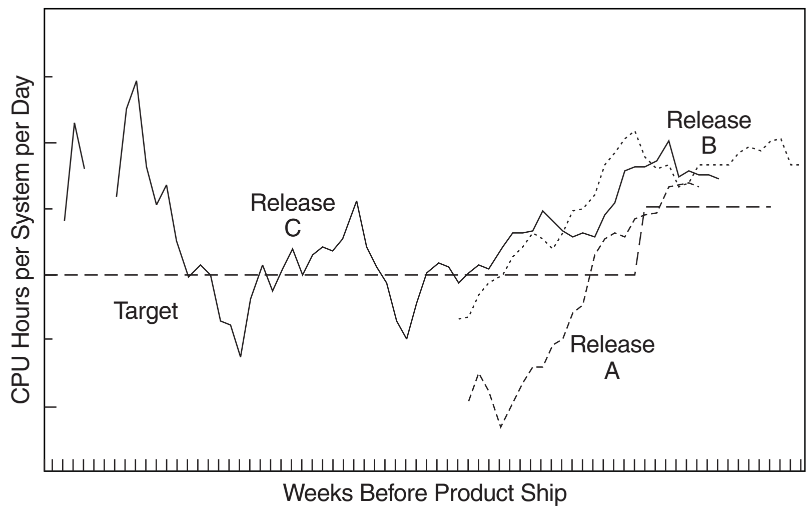


FIGURE 10.11
CPU Utilization Metrics

The CPU utilization metric is used together with the system crashes and hangs metric. This relationship is discussed in the next section.

To collect CPU utilization data, a performance monitor tool runs continuously (24x7) on each test system. Through the communication network, the data from the test systems are sent to a nontest system on a real-time basis. By means of a Lotus Notes database application, the final data can be easily tallied, displayed, and monitored.

10.1.6 System Crashes and Hangs

Hand in hand with the CPU utilization metric is the system crashes and hangs metric. This metric is operationalized as the number of unplanned initial program loads (IPLs, or reboots) because for each crash or hang, the system has to be re-IPLed (rebooted). For software tests whose purpose is to improve the stability of the system, we need to ensure that the system is stressed and testing is conducted effectively to uncover latent defects that would lead to system crashes and hangs, or in general any unplanned IPLs. When such defects are discovered and fixed, stability of the system improves over time. Therefore, the metrics of CPU utilization (stress level) and unplanned IPLs describe the effort aspect and the outcome aspect respectively, of the effectiveness of the test.

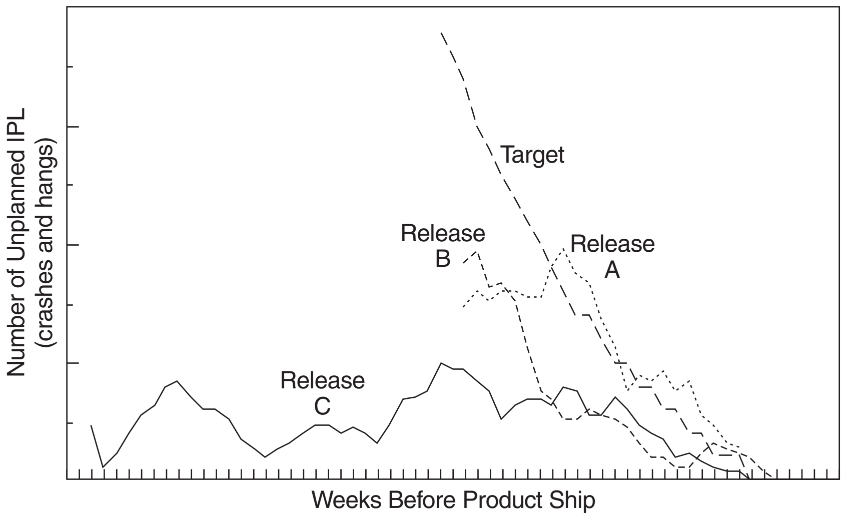


FIGURE 10.12
System Crashes and Hangs Metric

Figure 10.12 shows the system crashes and hangs metric for the same three releases shown in Figure 10.11. The target curve was derived based on data from prior releases by fitting an exponential model.

In terms of data collection, when a system crash or hang occurs and the tester reboots (re-IPLs) the system, the performance monitor and IPL tracking tool produces a screen prompt and requests information about the last system crash or hang. The tester can ignore the prompt temporarily, but it will reappear regularly after a certain time until the questions are answered. Information elicited via this tool includes test system, network ID, tester name, IPL code and reason (and additional comments), system reference code (SRC) if available, data and time system went down, release, driver, PTR number (the defect that caused the system crash or hang), and the name of the product. The IPL reason code consists of the following categories:

- 001 Hardware problem (unplanned)
- 002 Software problem (unplanned)
- 003 Other problem (unplanned)
- 004 Load fix (planned)

Because the volume and trend of system crashes and hangs are germane to the stability of the product in the field, we highly recommend this in-process metric for software for which stability is an important attribute. These data should also be used to make release-to-release comparisons and as leading indicators to product delivery readiness. While CPU utilization tracking definitely requires a tool, tracking of system crashes and hangs can start with pencil and paper if a disciplined process is in place.

10.1.7 Mean Time to Unplanned IPL

Mean time to failure (MTTF), or mean time between failures (MTBF), are the standard measurements of reliability. In software reliability literature, this metric and various models associated with it have been discussed extensively. Predominantly, the discussions and use of this metric are related to academic research or specific-purpose software systems. To the author's awareness, implementation of this metric is rare in organizations that develop commercial systems. This may be due to several reasons including issues related to single-system versus multiple-systems testing, the definition of a failure, the feasibility and cost in tracking all failures and detailed time-related data (Note: Failures are different from defects or faults; a single defect can cause multiple failures and in different machines) in commercial projects, and the value and return on investment of such tracking.

System crashes and hangs (unplanned IPLs) are the more severe forms of failure. Such failures are clear-cut and easier to track, and metrics based on such data are more meaningful. Therefore, at IBM Rochester, we use mean time to unplanned IPL (MTI) as the software reliability metric. This metric is used only during the system testing period, which, as previously described, is a customerlike system integration test prior to product delivery. Using this metric for other tests earlier in the development cycle is possible but will not be as meaningful because all the components of the system cannot be addressed collectively until the final system test. The formula to calculate the MTI metric is:

$$\text{Weekly MTI} = \sum_{i=1}^{n} W_i \bullet \left(\frac{H_i}{I_i + 1} \right)$$

where

n = Number of weeks that testing has been performed (i.e., the current week of test)

H = Total of weekly CPU run hours

W = Weighting factor

I = Number of weekly (unique) unplanned IPLs (due to software failures)

Basically the formula takes the total number of CPU run hours for each week (H_i), divides it by the number of unplanned IPLs plus 1 (I_i + 1), then applies a set of weighting factors to get the weighted MTI number, if weighting is desired. For example, if the total CPU run hours from all test systems for a specific week was 320 CPU hours and there was one unplanned IPL due to a system crash, then the unweighted MTI for that week would be 320/(1+1) = 160 CPU hours. In the IBM Rochester implementation, we apply a set of weighting factors based on results from prior baseline releases. The purpose of weighting factors is to take the outcome from the prior weeks into account so that at the end of the system test (with a duration of 10 weeks), the MTI represents an entire system test statement. It is the practitioner's decision whether to use a weighting factor or how to distribute the weights heuristically. Deciding factors may include type of products and systems under test, test cycle duration, and how the test period is planned and managed.

Figure 10.13 is an example of the MTI metric for the system test of a recent release of an integrated operating system. The *X*-axis represents the number of weeks before product ship. The *Y*-axis on the right side is MTI and on the left side is the number of unplanned IPLs. Inside the chart, the shaded areas represent the number of

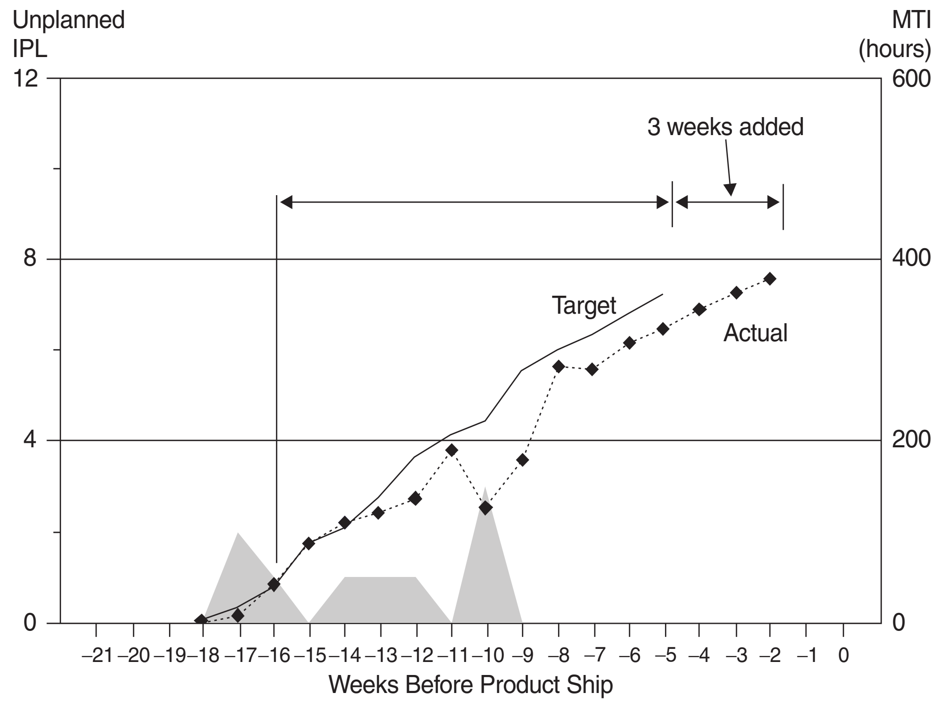


FIGURE 10.13
Mean Time to Unplanned IPL Metric

unique unplanned IPLs (crashes and hangs) encountered. From the start of the acceptance test of the system test, the MTI metric is shown tracking to plan until week 10 before product ship, when three system crashes occurred during one week. From the significant drop of the MTI, it was evident that with the original test plan, there would not be enough burn-in time for the system to reach the MTI target. Because this lack of burn-in time might result in undetected critical problems, additional testing was done and the system test was lengthened by three weeks. The product ship date remained unchanged.

Clearly, discrepancies between actual and targeted MTI should trigger early, proactive decisions to adjust testing plans and schedules to make sure that product ship criteria for burn-in can be achieved. At a minimum, the risks should be well understood and a risk mitigation plan should be developed. Action plans might include:

- Extending test duration and/or adding resources
- Providing for a more exhaustive regression test period if one were planned
- Adding a regression test if one were not planned
- Taking additional actions to intensify problem resolution and fix turnaround time (assuming that there is enough time available until the test cycle is planned to end)

10.1.8 Critical Problems: Showstoppers

This showstopper parameter is very important because the severity and impact of software defects varies. Regardless of the volume of total defect arrivals, it takes only a few showstoppers to render a product dysfunctional. This metric is more qualitative than the metrics discussed earlier. There are two aspects of this metric. The first is the number of critical problems over time, with release-to-release comparison. This dimension is quantitative. The second, more important, dimension is concerned with the types of the critical problems and the analysis and resolution of each problem.

The IBM Rochester's implementation of this tracking and focus is based on the general criteria that any problem that will impede the overall progress of the project or that will have significant impact on customer's business (if not fixed) belongs to such a list. The tracking normally starts at the middle of the component test phase when a critical problem meeting by the project management team (with representatives from all functional areas) takes place once a week. When it gets closer to system test and product delivery time, the focus intensifies and daily meetings take place. The objective is to facilitate cross-functional teamwork to resolve the problems swiftly. Although there is no formal set of criteria, problems on the critical problem list tend to be problems related to installation, system stability, security, data corruption, and so forth. All problems on the list must be resolved before product delivery.

10.2 In-Process Metrics and Quality Management

On the basis of the previous discussions of specific metrics, we have the following recommendations for implementing in-process metrics for software testing in general:

- Whenever possible, use calendar time, instead of phases of the development process, as the measurement unit for in-process metrics. There are some phase-based metrics or defect cause analysis methods available, which we also use. However, in-process metrics based on calendar time provide a direct statement on the status of the project with regard to whether it can be developed on time with desirable quality. As appropriate, a combination of time-based metrics and phase-based metrics may be desirable.
- For time-based metrics, use ship date as the reference point for the *X*-axis and use week as the unit of measurement. By referencing the ship date, the metric portrays the true in-process status and conveys a “marching toward completion” message. In terms of time units, we found that data at the daily level proved to have too much fluctuation and data at the monthly level lost its timeliness, and neither can provide a trend that can be spotted easily. Weekly data proved optimal in terms of both measurement trends and cycles for actions. Of course, when the project is approaching the back end of the development cycle, some metrics may need to be monitored and actions taken daily. For very small projects, the time units should be scaled according to the length of the test cycle and the pattern of defect arrivals. For instance, the example in Chapter 12 (Figure 12.5) shows the relationship between defect arrivals and hours of testing. The testing cycle was about 80 hours so the time unit was hour. One can observe that the defect arrival pattern by hour of testing shows a start, ramp-up, and then stabilizing pattern, which is a positive pattern.
- Metrics should indicate “good” or “bad” in terms of quality or schedule. To achieve these objectives, a comparison baseline (a model or some history) should always be established. Metrics should also have a substantial visual component so that “good” and “bad” are observable by the users without significant analysis. In this regard, we recommend frequent use of graphs and trend charts.
- Some metrics are subject to strong management actions, whereas a few specific ones should not be intervened with. For example, defect arrival pattern is an important quality indicator of the project. It is driven by test effectiveness and test progress. It should not be artificially controlled. When defects are discovered by testing, defect reports should be opened and tracked. On the other hand, testing progress can be managed. Therefore, defect arrival pattern can be influenced only indirectly via managing the testing. In contrast, defect backlog is completely subject to management and control.

- □ Finally, the metrics should be able to drive improvements. The ultimate questions for the value of metrics is, as a result of metrics, what kind and how much improvement will be made and to what extent will the final product quality be influenced?

With regard to the last item in the list, to drive specific improvement actions, sometimes the metrics have to be analyzed at a granular level. As a real-life example, for the test progress and defect backlog (PTR backlog) metrics, the following analysis was conducted and guidelines for action were provided for the component teams for an IBM Rochester project near the end of the component test (CT) phase.

- □ Components that were behind in the CT were identified using the following methods:
  - Sorting all components by "% of total test cases attempted" and selecting those that are less than 65%. In other words, with less than 3 weeks to component test complete, these components have more than one-third of testing left.
  - Sorting all components by "number of planned cases not attempted" and selecting those that have 100 or larger, and adding these components to those identified in step 1. In other words, these several additional components may be on track or not seriously behind percentage-wise, but because of the large number of test cases they have, a large amount of work remains.

  (Because the unit (test case, or test variation) is not of the same weight across components, step 1 was used as the major criterion, supplemented by step 2.)
- □ Components with double-digit PTR backlogs were identified.
- □ Guidelines for actions were devised:
  - If CT is way behind and PTR backlog is not high, the first priority is to focus on finishing CT.
  - If CT is on track and PTR backlog is high, the key focus is on reducing PTR backlog.
  - If CT is way behind and PTR backlog is high, then these components are really in trouble. GET HELP (e.g., extra resources, temporary help from other component teams who have experience with this component).
  - For the rest of the components, continue to keep a strong focus both on finishing CT and reducing PTR backlog.

Furthermore, analysis on defect cause, symptoms, defect origin (in terms of development phase), and where found can provide more information for possible improvement actions. Such analyses are discussed in previous chapters. Tables 10.2 and 10.3 show two examples on defect cause distribution and the distribution of defects found by test phase across development teams for a systems software project. The defect causes are categorized into initialization-related problems (INIT), data

TABLE 10.2
Percent Distribution of Defect Cause by Development Team

Defect Cause	Team A	Team B	Team C	Team D	Team E	Team F	Team G	Team H	Project Overall
Initialization (INIT)	11.5%	9.8%	12.3%	9.6%	10.6%	10.4%	**13.9%**	6.4%	10.6%
Definition (DEFN)	5.5	**34.9**	8.5	6.6	2.8	10.9	9.5	8.3	10.7
Interface (INTF)	10.6	16.3	15.8	**31.3**	8.3	19.3	12.0	11.3	15.6
Logic, algorithm (LGC)	**59.9**	**26.1**	54.2	41.4	54.4	49.7	48.6	**64.9**	50.4
Machine readable information (MRI)	**3.7**	1.4	**3.1**	0.5	0.9	1.8	0.7	1.1	1.7
Complex problems (CPLX)	8.8	11.6	6.1	10.6	**23.0**	7.9	**15.3**	7.9	11.0
TOTAL (*n*)	100.0% (217)	100.1% (215)	100.0% (260)	100.0% (198)	100.0% (217)	100.0% (394)	100.0% (274)	99.9% (265)	100.0% (2040)

TABLE 10.3
Percent Distribution of Defect Found by Testing Phase by Development Team

Team	UT	CT	CRT	Artistic	PLT	ST	Total (*n*)
A	26.7%	35.9%	9.2%	8.4%	6.9%	**12.9%**	100.0% (217)
B	25.6	24.7	7.4	**38.1**	2.8	**1.4**	100.0 (215)
C	31.9	33.5	9.2	12.3	5.4	7.7	100.0 (260)
D	**41.9**	29.8	11.1	12.1	**1.5**	3.6	100.0 (198)
E	38.2	23.5	11.1	5.0	11.1	**11.1**	100.0 (217)
F	**18.0**	39.1	7.4	3.3	**25.3**	6.9	100.0 (394)
G	**19.0**	29.9	18.3	**21.5**	4.4	6.9	100.0 (274)
H	26.0	36.2	17.7	12.8	4.2	3.1	100.0 (265)
Proejct Overall	27.1%	32.3%	11.4%	13.4%	9.1%	6.7%	100.0% (2040)

definition–related problems (DEFN), interface problems (INTF), logical and algorithmic problems (LGC), problems related to messages, translation, and machine-readable information (MRI), and complex configuration and timing problems (CPLX). The test phases include unit test (UT), component test (CT), component regression test (CRT), artistic test, product level test (PLT), and system test (ST). Artistic test is the informal testing done by developers during the formal CT, CRT, and PLT test cycles. It usually results from a "blitz test" focus on specific functions, additional testing triggered by in-process quality indicators, or new test cases in response to newly discovered problems in the field. In both tables, the percentages that are highlighted in bold numbers differ substantially from the pattern for the overall project.

Metrics are a tool for project and quality management. For many types of projects, including software development, commitment by the teams is very important. Experienced project managers know, however, that subjective commitment is not enough. Do you commit to the system schedules and quality goals? Will you deliver on time with desirable quality? Even with strong commitment by the development teams to the project manager, these objectives are often not met for a host of reasons, right or wrong. In-process metrics provide the added value of objective indication. It is the combination of subjective commitments and objective measurements that will make the project successful.

To successfully manage in-process quality and therefore the quality of the final deliverables, in-process metrics must be used effectively. We recommend an integrated approach to project and quality management vis-à-vis these metrics in which quality is managed as vigorously as factors such as schedule, cost, and content. Quality should always be an integral part of the project status report and checkpoint reviews. Indeed, many examples described here are metrics for both quality and schedules (those weeks to delivery date measurements) because the two parameters are often intertwined.

One common observation with regard to metrics in software development is that project teams often explain away the negative signs indicated by the metrics. There are two key reasons for this phenomenon. First, in practice many metrics are inadequate to measure the quality of the project. Second, project managers might not be action-oriented or not willing to take ownership of quality management. Therefore, the effectiveness, reliability, and validity of metrics are far more important than the quantity of metrics. We recommend using only a few important and manageable metrics during the project. When a negative trend is observed, an early urgent response can prevent schedule slips and quality deterioration. Such an approach can be supported by setting in-process metric targets. Corrective actions should be triggered when the measurements fall below a predetermined target.

10.2.1 Effort/Outcome Model

It is clear that some metrics are often used together to provide adequate interpretation of the in-process quality status. For example, test progress and defect arrivals (PTR arrivals), and CPU utilization and the number of system crashes and hangs are two obvious pairs. If we take a closer look at the metrics, we can classify them into two groups: those that measure the testing effectiveness or testing effort, and those that indicate the outcome of the test in terms of quality, or the lack thereof. We call the two groups the effort indicators (e.g., test effectiveness assessment, test progress S curve, CPU utilization during test) and the outcome indicators (PTR arrivals—total number and arrivals pattern, number of system crashes and hangs, mean time to unplanned initial program load (IPL)), respectively.

To achieve good test management, useful metrics, and effective in-process quality management, the effort/outcome model should be used. The 2x2 matrix in Figure 10.14 for testing-related metrics is equivalent to that in Figures 9.4 and 9.17 for inspection-related metrics. For the matrix on test effectiveness and the number of defects:

- Cell 2 is the best-case scenario. It is an indication of good intrinsic quality of the design and code of the software—low error injection during the development process—and verified by effective testing.
- Cell 1 is a good/not bad scenario. It represents the situation that latent defects were found via effective testing.
- Cell 3 is the worst-case scenario. It indicates buggy code and probably problematic designs—high error injection during the development process.
- Cell 4 is the unsure scenario. One cannot ascertain whether the lower defect rate is a result of good code quality or ineffective testing. In general, if the test effectiveness does not deteriorate substantially, lower defects is a good sign.

		Outcome (Defects Found)	
		Higher	Lower
Effort (Testing Effectiveness)	Better	Cell1 Good/Not Bad	Cell2 Best-Case
	Worse	Cell3 Worst-Case	Cell4 Unsure

FIGURE 10.14
An Effort/Outcome Matrix

It should be noted that in an effort/outcome matrix, the better/worse and higher/lower designations should be carefully determined based on project-to-project, release-to-release, or actual-to-model comparisons. This effort/outcome approach also provides an explanation of Myers (1979) counterintuitive principle of software testing as discussed in previous chapters. This framework can be applied to pairs of specific metrics. For testing and defect volumes (or defect rate), the model can be applied to the overall project level and in-process metrics level. At the overall project level, the effort indicator is the assessment of test effectiveness compared to the baseline, and the outcome indicator is the volume of all testing defects (or overall defect rate) compared to the baseline, when all testing is complete. As discussed earlier, it is difficult to derive a quantitative indicator of test effectiveness. But an ordinal assessment (better, worse, about equal) can be made via test coverage (functional or some coverage measurements), extra testing activities (e.g., adding a separate phase), and so forth.

At the in-process status level, the test progress S curve is the effort indicator and the defect arrival pattern (PTR arrivals) is the outcome indicator. The four scenarios will be as follows:

- □ Positive Scenarios
  - The test progress S curve is the same as or ahead of baseline (e.g., a previous release) and the defect arrival curve is lower (than that of a previous release). This is the cell 2 scenario.
  - The test progress S curve is the same as or ahead of the baseline and the defect arrival is higher in the early part of the curve—chances are the defect arrivals will peak earlier and decline to a lower level near the end of testing. This is the cell 1 scenario.

- □ Negative Scenarios
  - The test progress S curve is significantly behind and the defect arrival curve is higher (compared with baseline)—chances are the PTR arrivals will peak later and higher and the problem of late cycle defect arrivals will emerge. This is the cell 3 scenario.
  - The test S curve is behind and the defect arrival is lower in the early part of the curve —this is an unsure scenario. This is the cell 4 scenario.

Both cell 3 (worst case) and cell 4 (unsure) scenarios are unacceptable from quality management's point of view. To improve the situation at the overall project level, if the project is still in early development the test plans have to be more effective. If testing is almost complete, additional testing for extra defect removal needs to be done. The improvement scenarios take three possible paths:

1. If the original scenario is cell 3 (worst case), the only possible improvement scenario is cell 1 (good/not bad). This means achieving quality via extra testing.
2. If the original scenario is cell 4 (unsure), the improvement scenario can be one of the following two:
   - □ Cell 1 (good/not bad) means more testing leads to more defect removal, and the original low defect rate was truly due to insufficient effort.
   - □ Cell 2 (best case) means more testing confirmed that the intrinsic code quality was good, that the original low defect rate was due to lower latent defects in the code.

For in-process status, the way to improve the situation is to accelerate the test progress. The desirable improvement scenarios take two possible paths:

1. If the starting scenario is cell 3 (worst case), then the improvement path is cell 3 to cell 1 to cell 2.
2. If the starting scenario is cell 4 (unsure), improvement path could be:
   - □ Cell 4 to cell 2
   - □ Cell 4 to cell 1 to cell 2

The difference between the overall project level and the in-process status level is that for the latter situation, cell 2 is the only desirable outcome. In other words, to ensure good quality, the defect arrival curve has to decrease to a low level when active testing is still going on. If the defect arrival curve stays high, it implies that there are substantial latent defects in the software. One must keep testing until the defect arrivals show a genuine pattern of decline. At the project level, because the volume of defects (or defect rate) is cumulative, both cell 1 and cell 2 are desirable outcomes from a testing perspective.

Generally speaking, outcome indicators are fairly common; effort indicators are more difficult to establish. Moreover, different types of software and tests may need different effort indicators. Nonetheless, the effort/outcome model forces one to establish appropriate effort measurements, which in turn, drives the improvements in testing. For example, the metric of CPU utilization is a good effort indicator for systems software. In order to achieve a certain level of CPU utilization, a stress environment needs to be established. Such effort increases the effectiveness of the test. The level of CPU utilization (stress level) and the trend of the number of system crashes and hangs are a good pair of effort/outcome metrics.

For integration type software where a set of vendor software are integrated together with new products to form an offering, effort indicators other than CPU stress level may be more meaningful. One could look into a test coverage-based metric including the major dimensions of testing such as:

- ☐ Setup
- ☐ Install
- ☐ Min/max configuration
- ☐ Concurrence
- ☐ Error-recovery
- ☐ Cross-product interoperability
- ☐ Cross-release compatibility
- ☐ Usability
- ☐ Double-byte character set (DBCS)

A five-point score (1 being the least effective and 5 being the most rigorous testing) can be assigned for each dimension and their sum can represent an overall coverage score. Alternatively, the scoring approach can include the "should be" level of testing for each dimension and the "actual" level of testing per the current test plan based on independent assessment by experts. Then a "gap score" can be used to drive release-to-release or project-to-project improvement in testing. For example, assume the test strategy for a software offering calls for the following dimensions to be tested, each with a certain sufficiency level: setup, 5; install, 5; cross-product interoperability, 4; cross-release compatibility, 5; usability, 4; and DBCS, 3. Based on expert assessment of the current test plan, the sufficiency levels of testing are setup, 4; install, 3; and cross-product interoperability, 2; cross-release compatibility, 5; usability, 3; DBCS, 3. Therefore the "should be" level of testing would be 26 and the "actual" level of testing would be 20, with a gap score of 6. This approach may be somewhat subjective but it also involves in the assessment process the experts who can make the difference. Although it would not be easy in real-life implementation, the point here is that the effort/outcome paradigm and the focus on effort metrics have direct linkage to test improvements. Further research in this area or implementation experience will be useful.

For application software in the external user test environment, usage of key features of the software and hours of testing would be good effort indicators, and the number of defects found can be the outcome indicator. Again to characterize the quality of the product, the defect curve must be interpreted with data about feature usage and effort of testing. *Caution:* To define and develop effort indicators, the focus should be on the effectiveness of testing rather than on the person-hour (or person-month) effort in testing per se. A good testing strategy should strive for efficiency (via tools and automation) as well as effectiveness.

10.3 Possible Metrics for Acceptance Testing to Evaluate Vendor-Developed Software

Due to business considerations, a growing number of organizations rely on external vendors to develop the software for their needs. These organizations typically conduct an acceptance test to validate the software. In-process metrics and detailed information to assess the quality of the vendors' software are generally not available to the contracting organizations. Therefore, useful indicators and metrics related to acceptance testing are important for the assessment of the software. Such metrics would be different from the calendar-time–based metrics discussed in previous sections because acceptance testing is normally short and there may be multiple code drops and, therefore, multiple mini acceptance tests in the validation process.

The IBM 2000 Sydney Olympics project was one such project, in which IBM evaluated vendor-delivered code to ensure that all elements of a highly complex system could be integrated successfully (Bassin, Biyani, and Santhanam, 2002). The summer 2000 Olympic Games was considered the largest sporting event in the world. For example, there were 300 medal events, 28 different sports, 39 competition venues, 30 accreditation venues, 260,000 INFO users, 2,000 INFO terminals, 10,000 news records, 35,000 biographical records, and 1.5 million historical records. There were 6.4 million INFO requests per day on the average and the peak Internet hits per day was 874.5 million. For the Venue Results components of the project, Bassin, Biyani, and Santhanam developed and successfully applied a set of metrics for IBM's testing of the vendor software. The metrics were defined based on test case data and test case execution data; that is, when a test case was attempted for a given increment code delivery, an execution record was created. Entries for a test case execution record included the date and time of the attempt, and the execution status, test phase, pointers to any defects found during execution, and other ancillary information. There were five categories of test execution status: pass, completed with errors, fail, not implemented, and blocked. A status of "failed" or "completed with errors" would result in the generation of a defect record. A status of "not implemented" indicated that the test case did not succeed because the targeted function had not yet been implemented, because this was in an incremental code delivery environment. The "blocked" status was used when the test case did not succeed because access to the targeted area was blocked by code that was not functioning correctly. Defect records would not be recorded for these latter two statuses. The key metrics derived and used include the following:

Metrics related to test cases

- *Percentage of test cases attempted*—used as an indicator of progress relative to the completeness of the planned test effort
- *Number of defects per executed test case*—used as an indicator of code quality as the code progressed through the series of test activities
- *Number of failing test cases without defect records*—used as an indicator of the completeness of the defect recording process

Metrics related to test execution records

- *Success rate*—The percentage of test cases that passed at the last execution was an important indicator of code quality and stability.
- *Persistent failure rate*—The percentage of test cases that consistently failed or completed with errors was an indicator of code quality. It also enabled the identification of areas that represented obstacles to progress through test activities.
- *Defect injection rate*—The authors used the percentage of test cases whose status went from pass to fail or error, fail to error, or error to fail, as an indicator of the degree to which inadequate or incorrect code changes were being made. Again, the project involves multiple code drops from the vendor. When the status of a test case changes from one code drop to another, it is an indication that a code change was made.
- *Code completeness*—The percentage of test executions that remained "not implemented" or "blocked" throughout the execution history was used as an indicator of the completeness of the coding of component design elements.

With these metrics and a set of in-depth defect analysis referenced as orthogonal defect classification, Bassin and associates were able to provide value-added reports, evaluations, and assessments to the project team.

These metrics merit serious considerations for software projects in similar environments. The authors contend that the underlying concepts are useful, in addition to vendor-delivered software, for projects that have the following characteristics:

- Testers and developers are managed by different organizations.
- The tester population changes significantly, for skill or business reasons.
- The development of code is iterative.
- The same test cases are executed in multiple test activities.

It should be noted these test case execution metrics require tracking at a very granular level. By definition, the unit of analysis is at the execution level of each test case. They also require the data to be thorough and complete. Inaccurate or incomplete data will have much larger impact on the reliability of these metrics than on metrics based on higher-level units of analysis. Planning the implementation of these metrics therefore must address the issues related to the test and defect tracking system as part of the development process and project management system. Among the most important issues are cost and behavioral compliance with regard to the recording of accurate data. Finally, these metrics measure the outcome of test executions. When using these metrics to assess the quality of the product to be shipped, the effectiveness of the test plan should be known or assessed a priori, and the framework of effort/outcome model should be applied.

10.4 How Do You Know Your Product Is Good Enough to Ship?

Determining when a product is good enough to ship is a complex issue. It involves the types of products (e.g., a shrink-wrap application software versus an operating system), the business strategy related to the product, market opportunities and timing, customers requirements, and many more factors. The discussion here pertains to the scenario in which quality is an important consideration and that on-time delivery with desirable quality is the major project goal.

A simplistic view is that one establishes a target for one or several in-process metrics, and if the targets are not met, then the product should not be shipped per schedule. We all know that this rarely happens in real life, and for legitimate reasons. Quality measurements, regardless of their maturity levels, are never as black and white as meeting or not meeting a delivery date. Furthermore, there are situations where some metrics are meeting targets and others are not. There is also the question of how bad is the situation. Nonetheless, these challenges do not diminish the value of in-process measurements; they are also the reason for improving the maturity level of software quality metrics.

In our experience, indicators from at least the following dimensions should be considered together to get an adequate picture of the quality of the product.

- ☐ System stability, reliability, and availability
- ☐ Defect volume
- ☐ Outstanding critical problems
- ☐ Feedback from early customer programs
- ☐ Other quality attributes that are of specific importance to a particular product and its customer requirements and market acceptance (e.g., ease of use, performance, security, and portability.)

When various metrics are indicating a consistent negative message, the product will not be good enough to ship. When all metrics are positive, there is a good chance that the product quality will be positive in the field. Questions arise when some of the metrics are positive and some are not. For example, what does it mean to the field quality of the product when defect volumes are low and stability indicators are positive but customer feedback is less favorable than that of a comparable release? How about when the number of critical problems is significantly higher and all other metrics are positive? In those situations, at least the following points have to be addressed:

- Why is this and what is the explanation?
- What is the influence of the negative in-process metrics on field quality?
- What can be done to control and mitigate the risks?
- For the metrics that are not meeting targets, how bad is the situation?

Answers to these questions are always difficult, and seldom expressed in quantitative terms. There may not even be right or wrong answers. On the question of how bad is the situation for metrics that are not meeting targets, the key issue is not one of statistical significance testing (which helps), but one of predictive validity and possible negative impact on field quality after the product is shipped. How adequate the assessment is and how good the decision is depend to a large extent on the nature of the product, experience accumulated by the development organization, prior empirical correlation between in-process metrics and field performance, and experience and observations of the project team and those who make the GO or NO GO decision. The point is that after going through all metrics and models, measurements and data, and qualitative indicators, the team needs to step back and take a big-picture view, and subject all information to its experience base in order to come to a final analysis. The final assessment and decision making should be analysis driven, not data driven. Metric aids decision making, but do not replace it.

Figure 10.15 is an example of an assessment of in-process quality of a release of a systems software product when it was near the ship date. The summary table outlines the indicators used (column 1), key observations of the status of the indicators (column 2), release-to-release comparisons (columns 3 and 4), and an assessment (column 5). Some of the indicators and assessments are based on subjective information. Many parameters are based on in-process metrics and data. The assessment was done about two months before the product ship date, and actions were taken to address the areas of concern from this assessment. The release has been in the field for more than two years and has demonstrated excellent field quality.

Indicator	Observation	Versus Release A	Versus Release B	Assessment
Component Test	Base complete. Product X to complete 7/31.	⇔	⇔	Green
PTR Arrivals	Peak earlier than Release A and Release B, and lower at back end — for both absolute numbers and normalized (to size) rates.	⇑	⇑	Green
PTR Severity Distribution	Lower than Release A and Release B at back end.	⇑	⇑	Green
PTR Backlog	Excellent backlog management, lower than Release A and Release B, and achieved targets at Checkpoint Z. Needs focus for final take-down before product ship.	⇔	⇔	Green
Number of Pending Fixes	Higher than Release B at same time before product ship. Need focus to minimize customer rediscovery.	⇔	⇓	Yellow
Critical Problems	Strong problem management. Number of problems on the critical list similar to Release B.	⇑	⇑	Green
System Stability – Unplanned IPLs – CPU Run Time	Stability similar to, maybe slightly better than, Release B.	⇑	⇑	Green
Plan Change	Plan changes not as pervasive as Release B	N/A	⇑	Green
Timeliness of Translation and National Language Testing	Early and proactive build daily meetings. National language testing behind, but schedules achievable.	⇔	⇔	Green
Hardware System Test	Target complete: 7/31/xx. Focusing on backlog reduction. XX is a known problem area but receiving focus.	⇑	⇔	Green

Hardware Reliability	Projected to meet target (better than prior releases) for all models.	⇑	⇑	Green
Product Level Test	Testing continues for components DD and WWDatabase, but no major problems.	⇑	⇔	Green
Install Test	Phase II testing ahead of plan. One of the cleanest releases in install test.	⇑	⇑	Green
Serviceability and Upgrade Testing	Concern with configurator readiness, software order structure in manufacturing.	⇔	⇓	Red: Concern
Software System Test	Release looks good overall.	⇔	⇔	Green
Service Readiness	Worldwide service community is on track to be ready to support the release.	⇑	⇑	Green
Early Customer Programs	Good early customer feedback on the release.	⇔	⇔	Green
Manufacturing Build and Test	Still early, but no major problems.	⇔	⇔	Green

Key: ⇑ : Better than comparison release
⇔ : Same as comparison release
⇓ : Worse than comparison release

FIGURE 10.15
A Quality Assessment Summary

10.5 Summary

In this chapter we discuss a set of in-process metrics for the testing phases of the software development process. We provide real-life examples based on implementation experiences at the IBM Rochester software development laboratory. We also revisit the effort/outcome model as a framework for establishing and using in-process metrics for quality management.

There are certainly many more in-process metrics for software test that are not covered here; it is not our intent to provide a comprehensive coverage. Furthermore, not every metric we discuss here is applicable universally. We recommend that the several metrics that are basic to software testing (e.g., the test progress curve, defect arrivals density, critical problems before product ship) be integral parts of all software testing.

It can never be overstated that it is the effectiveness of the metrics that matters, not the number of metrics used. There is a strong temptation for quality practitioners to establish more and more metrics. However, ill-founded metrics are not only useless, they are actually counterproductive and add costs to the project. Therefore, we must take a serious approach to metrics. Each metric should be subjected to the examination of basic principles of measurement theory and be able to demonstrate empirical value. For example, the concept, the operational definition, the measurement scale, and validity and reliability issues should be well thought out. At a macro level, an overall framework should be used to avoid an ad hoc approach. We discuss the effort/outcome framework in this chapter, which is particularly relevant for in-process metrics. We also recommend the Goal/Question/Metric (GQM) approach in general for any metrics (Basili, 1989, 1995).

Recommendations for Small Organizations

For small organizations that don't have a metrics program in place and that intend to practice a minimum number of metrics, we recommend these metrics as basic to software testing: test progress S curve, defect arrival density, and critical problems or showstoppers.

For any projects and organizations we strongly recommend the effort/outcome model for interpreting the metrics for software testing and in managing their in-process quality. Metrics related to the effort side of the equation are especially important in driving improvement of software tests.

Finally, the practice of conducting an evaluation on whether the product is good enough to ship is highly recommended. The metrics and data available to support the evaluation may vary, and so may the quality criteria and the business strategy related to the product. Nonetheless, having such an evaluation based on both quantitative metrics and qualitative assessments is what good quality management is about.

At the same time, to enhance success, one should take a dynamic and flexible approach, that is, tailor the metrics to the needs of a specific team, product, and organization. There must be buy-in by the team (development and test) in order for the metrics to be effective. Metrics are a means to an end—the success of the project—not an end itself. The project team that has intellectual control and thorough understanding of the metrics and data they use will be able to make the right decisions. As such, the use of specific metrics cannot be mandated from the top down.

While good metrics can serve as a useful tool for software development and project management, they do not automatically lead to improvement in testing and in quality. They do foster data-based and analysis-driven decision making and provide objective criteria for actions. Proper use and continued refinement by those involved (e.g., the project team, the test community, the development teams) are therefore crucial.

References

1. Basili, V. R., "Software Development: A Paradigm for the Future," *Proceedings 13th International Computer Software and Applications Conference (COMPSAC),* Keynote Address, Orlando, Fla., September 1989.
2. Basili, V. R., "Software Measurement Workshop," University of Maryland, 1995.
3. Bassin, K., S. Biyani, and P. Santhanam, "Metrics to Evaluate Vendor Developed Software Based on Test Case Execution Results," *IBM Systems Journal,* Vol. 41, No. 1, 2002, pp. 13–30.
4. Hailpern, B., and P. Santhanam, "Software Debugging, Testing, and Verification," *IBM Systems Journal,* Vol. 41, No. 1, 2002, pp. 4–12.
5. McGregor, J. D., and D. A. Sykes, *A Practical Guide to Testing Object-Oriented Software,* Boston: Addison-Wesley, 2001.
6. Myers, G. J., *The Art of Software Testing,* New York: John Wiley & Sons, 1979.
7. Ryan, L., "Software Usage Metrics for Real-World Software Testing," *IEEE Spectrum,* April 1998, pp. 64–68.