CHAPTER 13

System Test Execution

Execute every act of thy life as though it were thy last.
— *Marcus Aurelius*

13.1 BASIC IDEAS

Preparing for and executing system-level tests are a critical phase in a software development process because of the following: (i) There is pressure to meet a tight schedule close to the delivery date; (ii) there is a need to discover most of the defects before delivering the product; and (iii) it is essential to verify that defect fixes are working and have not resulted in new defects. It is important to monitor the processes of test execution and defect fixing. To be able to monitor those test processes, we identify two key categories of metrics: (i) *system test execution status* and (ii) *defects status*. We provide a detailed *defect schema*, which includes a general FSM model of defects for ease of collecting those metrics. Analyzing defects is a crucial activity performed by the software development team while fixing defects. Therefore, in this chapter, we describe three types of defect analysis techniques: *causal, orthogonal*, and *Pareto methodology*. In addition, we provide three metrics, namely, *defect removal efficiency, spoilage*, and *fault seeding*, to measure test effectiveness. The objective of a test effectiveness metric is to evaluate the effectiveness of a system testing effort in the development of a product in terms of the number of defects escaped from the system testing effort.

The product is ready for beta testing at the customer site during the system testing. We provide a framework for beta testing and discuss how beta testing is conducted at the customer's site. Moreover, we provide a detailed structure of the system test execution report, which is generated before a product's *general availability* is declared.

Software Testing and Quality Assurance: Theory and Practice, Edited by Kshirasagar Naik and Priyadarshi Tripathy

13.2 MODELING DEFECTS

The key to a successful defect tracking system lies in properly modeling defects to capture the viewpoints of their many stakeholders, called cross-functionality groups. The cross-functionality groups in an organization are those groups that have different stakes in a product. For example, a marketing group, a customer support group, a development group, a system test group, and a product sustaining group are collectively referred to as cross-functionality groups in an organization. It is not enough to merely report a defect from the viewpoint of software development and product management and seek to understand it by means of reproduction before fixing it. In reality, a reported defect is an evolving entity that can be appropriately represented by giving it a life-cycle model in the form of a state transition diagram, as shown in Figure 13.1. The states used in Figure 13.1 are briefly explained in Table 13.1.

The state transition model allows us to represent each phase in the life cycle of a defect by a distinct state. The model represents the life cycle of a defect from its initial reporting to its final closing through the following states: new, assigned, open, resolved, wait, FAD, hold, duplicate, shelved, irreproducible, postponed, and closed. When a defect moves to a new state, certain actions are taken by the owner of the state. By "owner" of a state of a defect we mean the person or group of people who are responsible for taking the required actions in that state. Once the associated actions are taken in a state, the defect is moved to a new state.

Two key concepts involved in modeling defects are the levels of *priority* and *severity*. On one hand, a priority level is a measure of how soon the defect needs to be fixed, that is, urgency. On the other hand, a severity level is a measure of the extent of the detrimental effect of the defect on the operation of the product. Therefore, priority and severity assignments are separately done. In the following, four levels of defect priority are explained:

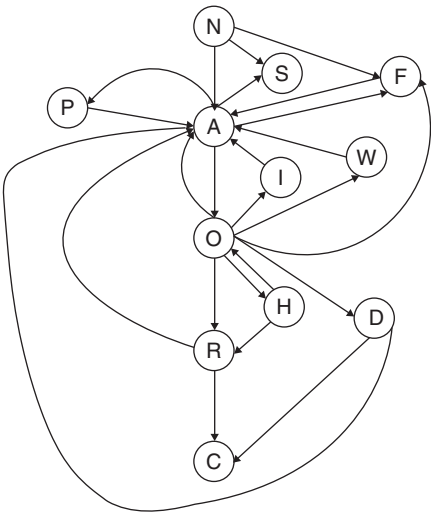


Figure 13.1 State transition diagram representation of life cycle of defect.

TABLE 13.1 States of Defect Modeled in Figure 13.1

State	Semantic	Description
N	New	A problem report with a severity level and a priority level is filed.
A	Assigned	The problem is assigned to an appropriate person.
O	Open	The assigned person is actively working on the problem to resolve it.
R	Resolved	The assigned person has resolved the problem and is waiting for the submitter to verify and close it.
C	Closed	The submitter has verified the resolution of the problem.
W	Watt	The assigned person is waiting for additional information from the submitter.
F	FAD	The reported defect is not a true defect. Rather, it is a function as designed.
H	Hold	The problem is on hold because this problem cannot be resolved until another problem is resolved.
S	Shelved	There is a problem in the system, but a conscious decision is taken that this will not be resolved in the near future. Only the development manager can move a defect to this state.
D	Duplicate	The problem reported is a duplicate of a problem that has already been reported.
I	Irreproducible	The problem reported cannot be reproduced by the assigned person.
P	Postponed	The problem will be resolved in a later release.

1. *Critical*: Defects at this level must be resolved as soon as possible. Testing should not progress until a known critical defect is fixed.
2. *High*: Defects at this level must be resolved with high priority. A subset of the system functionalities cannot be tested unless this level of defect is resolved.
3. *Medium*: Defects at this level will not have any impact on the progress of testing. Medium-priority defects stand by themselves.
4. *Low*: Defects at this level should be resolved whenever it is possible.

Intuitively, a critical-priority defect affects a large number of functionalities of a system, a high-priority defect affects a smaller subset of functionalities, a medium-priority defect affects an individual functionality, and a low-priority defect is considered to be a minor irritation. In the following, four levels of severity are explained:

1. *Critical*: A defect receives a "critical" severity level if one or more critical system functionalities are impaired by a defect with is impaired and there is no workaround.
2. *High*: A defect receives a "high" severity level if some fundamental system functionalities are impaired but a workaround exists.
3. *Medium*: A defect receives a "medium" severity level if no critical functionality is impaired and a workaround exists for the defect.

4. *Low*: A defect receives a "low" severity level if the problem involves a cosmetic feature of the system.

Intuitively, defects with high severity should receive a high priority in the resolution phase, but a situation may arise where a high-severity defect may be given a low priority. For example, let a system crash when the system date is set to December 32. Though a crash defect has severe consequences, an occurrence of December 32 as the system date is unlikely in normal circumstances. The said defect receives a critical severity but a low priority. In other words, if a critically severe defect occurs with an extremely low probability, then it may not be fixed immediately, that is, it may be treated with low priority.

Someone reporting a defect makes a preliminary assessment of the severity and priority levels of the defect. If there is a dispute among the different cross-functional groups concerning the priority and severity levels of a defect, an agreement can be reached in a regular review meeting. Note that the severity of a defect remains the same throughout its life cycle, whereas the priority level may change as the software product approaches its release date. A defect may be assigned a low priority at the beginning of system testing if it may not have any impact on the progress of testing, but it must be fixed before the software is released. For example, a spelling mistake in the GUI may be a low-priority defect at the beginning of system testing but its priority becomes high as the release date approaches.

In the following, we describe the states of a defect modeled as a state transition diagram, as shown in Figure 13.1, and explain how a defect moves from state to state. When a new defect is found, it is recorded in a defect tracking system, and the defect is put in its initial state new. The owner of this state is the software development manager. In this state, the following fields, explained as parts of a schema given in Table 13.2, are initialized: defect_id, submit_date, product, submitter, group, owner, headline, keywords, severity, priority, reproducible, category, software_version, build, description, h/w_configuration, s/w_configuration, attachments, notes, and number_tc_fail, tc_id.

The software development manger, who is the owner of the new state, moves the defect from the new state to the assigned state by changing the ownership either to a software developer or to himself. The software development manager may reject this defect by explaining why it is not a true defect and move the defect to the FAD state with a change in ownership back to the submitter. The development manager may change the state to the shelved state, which means that it is a true defect and a conscious decision has been made to the effect that this defect will not be fixed in the near future. All the relevant parties, especially the customer support group of the organization, must agree to shelve the defect before a defect is moved to the shelved state. There are several reasons to move a defect to the shelved state. For example, a defect may not have much impact on system operation in the customer's operational environment, and it may consume a lot of resources to fix it. In practice, very few defects land in the shelved state.

The assigned state means that someone has been assigned to fix the defect but the actual work required to fix the defect has not begun. The software development

TABLE 13.2 Defect Schema Summary Fields

Field Name	Description
defect_id	A unique, internally generated identifier for the defect
state	Current state of the defect; takes a value from the set {New, Assigned, Open, Resolved, Information, FAD, Hold, Duplicate, Shelved, Irreproducible, Postponed, Closed}
headline	One-line summary of defect
severity	Severity level of defect; takes a value from the set {critical, high, medium, low}
priority	Priority level of defect; takes a value from the set {critical, high, medium low}
submitter	Name of the person who submits the defect
group	Group affiliation of submitter; takes a value from the set {ST, SIT, software, hardware, customer support}
owner	Current owner of defect
reproducible	Says, in terms of yes or no, whether or not defect can be reproduced
crash	Says, in terms of yes or no, whether or not defect causes system to crash
keywords	Some common words that can be associated with this defect for searching purpose
product	Name of product in which defect was found
category	Test category name that revealed defect
software_version	Software version number in which defect was observed
build	Build number in which defect was observed
submit_date	Date of submission of defect
description	Brief description of defect
h/w_configuration	Description of hardware configuration of test bed
s/w_configuration	Description of software configuration of test bed
attachments	Attachments, in the form of log files, configuration files, etc., useful in understanding the defect
notes	Additional notes or comments, if there are any
number_tc_fail	Number of test cases failed or blocked because of defect
tc_id	List of test case identifiers of those test cases, from test factory database, which will fail because of defect
forecast_fix_version	Software version in which fix for defect will be available
forecast_build_number	Build number in which fix for defect will be available
actual_fix_version	Software version in which fix is actually available
actual_build_number	Build number in which fix is actually available
fix_description	Brief description of fix for defect
fix_date	Date when fix was checked in to code
duplicate_defect_id	Present defect considered to be duplicate of duplicate defect_id
requirement_id	Requirement identifier from requirement database that is generated as result of agreement that defect be turned into requirement

manager, or a developer, is the owner of the defect in the assigned state. The ownership may change to a different software developer who will fix the defect. The assigned software developer may move the defect from the assigned state to the open state as soon as the process of defect fixing begins. Note that only the software development manager moves a defect from the assigned state to one the following states: shelved, postponed, and FAD. The ownership is changed to the submitter when a defect is moved to the FAD state.

A software developer, the owner of the defect, is actively working to fix the defect in the open state. The developer initializes the forecast_fix_version and forecast_build_number fields of the defect schema given in Table 13.2 so that the submitter can plan and schedule a retest of the fix. A defect may lie in the open state for several weeks. The software developer moves the defect from this state to five possible next states—irreproducible, resolved, wait, hold, and duplicate—which are explained in Table 13.3 along with the actions that need to be taken when the state is changed.

If a software developer working on a defect is satisfied that the defect has been fixed, he or she moves the defect to a resolved state and changes the ownership

TABLE 13.3 State Transitions to Five Possible Next States from Open State

Possible *Next* State	Actions
Irreproducible	If the owner, i.e., a developer, cannot reproduce a defect, the defect is moved to irreproducible state with the ownership changed back to the submitter.
Wait	If the amount of information given by the submitter is not enough to understand and/or reproduce the defect, then the owner asks for more information on the defect from the submitter. Essentially, the developer is asking for more information about the defect in the notes field. The ownership field is changed back to the submitter.
Resolved	**1.** The ownership field is changed back to the submitter. **2.** The following fields need to be filled out: **3.** Actual_fix_version: Software version where the fix is available **4.** Actual_build_number: Build number where the fix is available. **5.** Fixed_description: Brief description of the fix. The developer must list all the files that are modified and what has been changed in the files in order to fix the problem. **6.** Fixed_date: Date the defect was fixed on and moved to the resolved state.
Hold	It has to be explained in the notes field why the defect was moved to this state. The hold state means that no firm decision has been made regarding the defect because the problem depends on some other issues, such as a software defect in a third-party product. Unless the other issues are resolved, the problem described in the defect report cannot be resolved.
Duplicate	If this defect is identical to, i.e., a duplicate of another defect, then the identifier of the original defect is recorded in the duplicate_defect_id field.

of the defect back to the submitter. A defect in the resolved state merely implies that the defect has been fixed to the satisfaction of the developer, and the associated tests need to be executed to further verify the claim in a wider context. The submitter moves the defect from the resolved state to the closed state by initializing the following fields of the defect schema after it is verified that the defect has been fixed: verifier, closed_in_version, closed_in_build, closed_date, and verification_description. Essentially, verification_description describes the details of the verification procedure. The closed state signifies that the defect has been resolved and verified. On the other hand, if the verifier is convinced that the defect has not been actually or completely fixed, the fix is rejected by moving the defect back to the assigned state, and the ownership of the defect is changed back to the responsible software developer. The verifier provides the following information while rejecting a fix:

- Explanation for rejection of fix
- Any new observation and/or problem encountered

A defect is moved to the FAD state by the development team if the team concludes that it is not a true defect and the system behaves as expected. The submitter, the owner of the defect in the FAD state, may involve the development team, the customer support group, the product management, and the project leadear to review defects in this state. There are three possible alternative outcomes of a defect review meeting:

- The reporting of the defect was due to a procedural error on the part of the submitter. The defect remains in the FAD state for this outcome. The idea behind keeping the defect in the FAD state, and not in the closed state, is to record the fact that the submitter had designated it as a defect.
- The reported defect is a true defect. The defect is moved to the assigned state with a note for the software development manager saying that this is a defect agreed to by all the parties.
- A suggestion is made that a new feature enhancement request be made to accommodate the system behavior perceived to be a "defect" at this moment. The submitter moves the defect to the assigned state with a note for the development manager saying that the defect may be resolved by creating a new requirement and submitting the requirement identifier in the requirement_id field of the defect schema.

If a defect cannot be fixed in a particular release, it is moved to the postponed state, where the development manager is the owner of the defect. Fixing a defect is postponed due to lack of time to fix it for a certain release. The defect is moved, after the release for its fix is known, to the assigned state, where the following fields of the defect schema are initialized: (i) forecast_build_number—the build in which a fix will be available for testing—and (ii) forecast_fix_version—the version in which a fix will be available.

All the outstanding defects that are still open are moved to the postponed state after the product is released to the customer, so that the defects can be scheduled

to be fixed in future software releases. The defects are moved to the assigned state after it is finalized in which release these defects will be fixed. The postponed state is useful in tracking and scheduling all the outstanding defects in the system that must be fixed in future releases.

If a developer finds that the information provided by the submitter is not adequate to fix a defect, the developer asks for additional information from the submitter by moving the defect from the open state to the wait state. In the wait state, the submitter is the owner of the defect. The submitter provides the information asked for by the developer by initializing the notes field of the defect schema, moves the defect to the assigned state, and changes the ownership back to the software developer.

The developer may try to reproduce the symptom to understand the defect. If the developer cannot reproduce the reported symptom, the defect is moved to the irreproducible state with the ownership changed back to the submitter. The submitter makes an effort to reproduce the defect, collect all the relevant information into the defect report, and move the defect to the assigned state if the defect is reproduced. If the submitter cannot reproduce the defect, the defect stays in the irreproducible state.

It may be observed that the defect is caused by an extraneous factor, such as a defect in a third-party product which is being used. The reported defect cannot be fixed unless the defect in the outside component is resolved. In such a case, the defect is put on hold by moving it to the hold state, where the software development manager is the owner of the defect. The software development manager moves the defect to either the resolved state or the open state depending on if the primary defect in the outside component has been resolved or is being fixed, respectively.

A defect in the duplicate state, where the submitter is the owner, means that the problem is a duplicate of a defect reported earlier. The original defect identifier is used to initialize the schema field duplicate_defect_id. Once the duplicate defect is in the closed state, the submitter must verify this defect. If the verification is successful, the submitter moves the defect to the closed state. On the other hand, if the verification fails, the submitter rejects the duplicity claim by moving the defect back to the assigned state, where the software developer is the owner of the defect. The verifier must give a reason for rejecting a defect as a duplicate defect. This information contains any observation and/or problem encountered while verifying the defect.

13.3 PREPAREDNESS TO START SYSTEM TESTING

The status of the entry criteria outlined in Chapter 13 must be tracked at least four weeks before the start of the first system test cycle. Any exceptions to these criteria must be noted and discussed at the weekly project status review meeting. We discuss the last item of the entry criteria, namely that the test execution working document is in place and complete. A framework of such a document is outlined in Table 13.4 and is explained subsequently. This working document is created, controlled, and tracked by the test team leader.

TABLE 13.4 Outline of Test Execution Working Document

1. Test engineers
2. Test cases allocation
3. Test beds allocation
4. Automation progress
5. Projected test execution rate
6. Execution of failed test cases
7. Development of new test cases
8. Trial of system image
9. Schedule of defect review meeting

The test engineers section contains the names of the test engineers assigned to this project as well as their availability and expertise in the specific areas of the project. The training requirement for each team members that may be necessary to successfully execute the test project is noted in detail. The action plan and training progress for each team member are tracked in this section. Any issue related to human resource availability or training is referred to the software project lead.

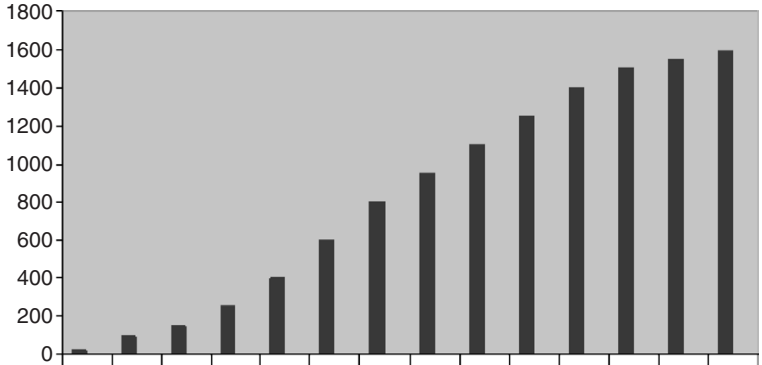
The test cases are allocated among test engineers based on their expertise and interest after the test suite is created in the test factory. The software development engineers and system test engineers are identified and included in the sw_dev and tester fields of the test cases, respectively. The idea here is to give the ownership of the test case execution to the individual test engineers. A test engineer reviews the test cases allocated to him or her to understand them and prioritize them in consultation with the software developers. Priority of a test case is set to one of the values from the set {high, medium, low}. If necessary, a test engineer may update the test cases to fine tune the test procedures and the pass–fail criteria as discussed in Section 11.6. The activities of prioritization and allocation of test cases among engineers and software developers are tracked in the test case allocation section, and this must be completed before the start of the system test cycle. This is achieved by calling several meetings with the team members of the software project.

The test beds allocation section of the document states the availability of the number and kinds of test beds and the ways these test beds are distributed so that the test cases are executed concurrently. A mapping is established between the available test beds and the subgroups of test cases from the test suite. The mapping is based on the configuration requirements to execute the groups of test cases. The test engineer responsible for executing a subgroup of test cases is then assigned to that test bed. The test engineer owns that particular test bed and must ensure that the test bed is up and ready to go before the start of system test cycles. Precaution must be taken to ensure that each test engineer is allocated the same test bed throughout the entire test execution cycle. The idea here is to maximize the execution efficiency of the engineer. If necessary, new test beds are created, or allocated from other projects, so that the idle time is minimized.

In the automation progress section of the document, a table is created to track the progress of the automation of the test cases and the availability of those automated test cases for execution. This is of particular interest in the case of regression test cases that were developed earlier and executed manually, but their automation is in progress. The test team leader interacts with the responsible test automation engineers to gather the information on the test automation progress. In case the automated test cases are available on time, that is, before the start of the test cycle, then the scheduled manual execution of these test cases may be skipped. Instead, the automated test cases can be executed.

In the projected test execution rate section of the document, a table is created to show the number of test cases that are anticipated to be executed on a weekly basis during a test cycle. The projected number of test cases is tracked against the actual execution of test cases during the test cycle. As an example, let us consider a test project name, Bazooka, for which the total number of test cases selected is 1592. The projected execution of 1592 test cases on a weekly basis for a 14-week test cycle is shown in Table 13.2 in the form of a cumulative chart. The execution of the 1592 test cases is expected to take 14 weeks; 25 test cases will be executed in the first week; 75 test cases will be executed in the second week; thus, 100 test cases will executed by the end of the second week. As the execution proceeds, the second row of Figure 13.2, namely, actually executed test cases, is updated. This gives the test group an idea of the progress of the testing. If the number of actually executed test cases falls far behind the projected number for several weeks, the test manager will know that the project is going to be delayed.

A strategy for the execution of failed test cases and verification of defect fixes is outlined in the execution of failed test cases section of the document. There are two possibilities here: (i) The failed test cases are executed in the next test cycle and (ii) the failed test cases are executed in the current cycle as soon



	wk1	wk2	wk3	wk4	wk5	wk6	wk7	wk8	wk9	wk10	wk11	wk12	wk13	wk14
■ Projected execution	25	100	150	250	400	600	800	950	1100	1250	1400	1500	1550	1592
□ Actually executed														

Figure 13.2 Projected execution of test cases on weekly basis in cumulative chart form.

as the corresponding fixes are available. In the first case, all the fixes, which are already integrated into the system, are subject to a regression test. No fixes are inserted into the system midway through a test cycle which are available at the beginning of a test cycle. In the second scenario, the fixes are checked in to the version control system, and a new build is created. The build is then sanity tested by the system integration group, and then it is released to the system testing group to be retested by running the test cases which failed earlier. If a test case now passes, then the fix has been verified to have resolved the defect. However, any collateral damage may not have been detected. A fix causing collateral damage is called a "bad fix," A handful of passed test cases around the fix may be reexecuted after a previously failed test case is observed to have passed irrespective of a fix that has caused collateral damage. The idea is to identify the high-impact region of the fix, select a set of previously passed test cases in consultation with the software developer who fixed the defect, and then execute those tests as a regression test. The following information is collected and analyzed in selecting the regression test cases:

- Parts of the software modules that were modified in order to incorporate the fix
- Functionality areas that are closely related to the fix and susceptible to defects
- Kinds of defects likely to be introduced because of the fix

Based on the above information, existing or new test cases are selected that are most likely to uncover any possible new defects that might have been introduced.

The test engineers learn more about the system and are in a better position to develop additional test cases during the first test cycle of testing. This is in particular important when a repeatable problem is observed and there are no test scenarios in the test suite. In this case, a new test case needs to be added to the test factory since the long-term goal is to improve the coverage and effectiveness of the test suite. However, a decision must be made regarding the addition of a test case into the test factory during the test cycle or after completion of the test cycle.

The development of new test cases section of the document clarifies the strategy for writing new test cases during the execution of test cases. It should be communicated to the test engineers before the start of the test cycle so that one of the following actions is taken:

- Develop new test cases during the test cycle.
- Include a note in the defect report saying that a test case needs to be developed later for this defect and continue with the execution of test cases as per schedule.

It is recommended to take the second approach, since it is difficult to find time to write the test cases during test execution unless it is in the schedule. Therefore, the test engineer should go back to the individual defects and add test cases to the test factory based on the defect report after completion of the test cycle.

Quite often the test engineers download a software image from the system integration group to their allocated test beds at least two or three weeks prior to the official start of the first test cycle. This allows the test engineer to be familiar with the software image and to ensure that the test bed is fully operational. This activity is tracked for each system test engineer in the trial of system image section of the working document, even though the build is not officially released to start system testing. There are several advantages of using the trial software image before the official start date of the system test cycle:

- The test engineers are trained to become familiar with the system so that test execution productivity is higher during the system test cycle.
- To ensure that the test bed is operational without any downtime at the beginning of the system test cycle.
- The basic test cases can be executed earlier to validate the test steps.

A schedule of the defect review meeting must be in place before the start of the system test cycle. The schedule is established for the entire test cycle duration before the start of the system test. The test leader may use a Gantt chart to represent a schedule and include it in the schedule the defect review meeting section of the test execution working document. The cross-functional team members from software development, hardware development, and the customer support groups are invited to this meeting.

13.4 METRICS FOR TRACKING SYSTEM TEST

The system test execution brings forth (three) different facets of software development. The developers would like to know the degree to which the system meets the explicit as well as implicit requirements. The delivery date cannot be precisely predicted due to the uncertainty in fixing the problems. The customer is excited to take the delivery of the product. It is therefore a highly visible and exciting activity. At this stage, it is desirable to monitor certain metrics which truly represent the progress of system testing and reveal the quality level of the system. Based on those metrics, the management can trigger actions for corrective and preventive measures. By putting a small but critical set of metrics in place executive management is in a position to know whether they are on the right track [1]. We make a difference between *statistics* and *in-process* metrics: Statistics are used for postproject analysis to gather experience and use it in future projects, whereas in-process metrics let us monitor the progress of the project while we still have an opportunity to steer the course of the project. By considering three large, real-life test projects, we categorize execution metrics into two classes:

- For monitoring test execution
- For monitoring defects

The first class of metrics concerns the process of executing test cases, whereas the second class concerns the defects found as a result of test execution. These

metrics need to be tracked and analyzed on a periodic basis, say, daily or weekly. It is important to gather valid and accurate information about the project for the leader of a system test group to effectively control a test project. One example of effective control of a test project is to precisely determine the time to trigger the revert criteria for a test cycle and initiate root cause analysis of the problems before more tests are performed. A test manager can effectively utilize the time of test engineers by triggering such a revert criterion, and possibly money, by suspending a test cycle on a product with too many defects to carry out a meaningful system test. Therefore, our objective is to identify and monitor the metrics while system testing is in progress so that the nimble decisions can be taken by the management team [2].

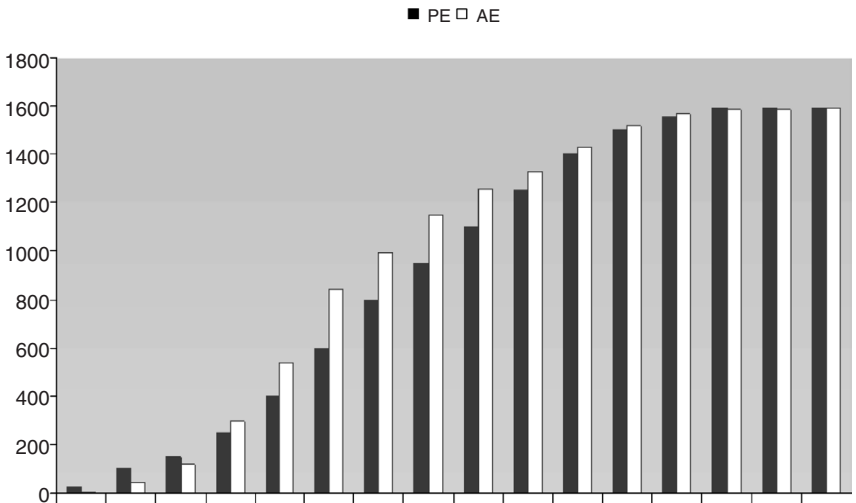
13.4.1 Metrics for Monitoring Test Execution

It is desirable to monitor the execution of system test cases of a large project involving tens of test engineers and taking several months. The system test execution for large projects is monitored weekly in the beginning and daily toward the finish. It is strongly recommended to use automated tools, such as a test factory, for monitoring and reporting the test case execution status. Different kinds of queries can be generated to obtain the *in-process* metrics after a database is in place. The following metrics are useful in successfully tracking test projects:

- **Test Case Escapes (TCE):** The test engineers may find the need to design new test cases during the testing, called test case escapes, as testing continues. The number of test case escapes is monitored as it rises. A significant increase in the number of test case escapes implies the deficiencies in the test design process, and this may adversely affect the project schedule.
- **Planned versus Actual Execution (PAE) Rate:** The actual number of test cases executed every week is compared with the planned number of test cases [3]. This metric is useful in representing the productivity of the test team in executing test cases. If the actual rate of execution falls far short of the planned rate, managers may have to take preventive measures so that the time required for system testing does not adversely affect the project schedule.
- **Execution Status of Test (EST) Cases:** It is useful to periodically monitor the number of test cases lying in different states—failed, passed, blocked, invalid, and untested—after their execution. It is also useful to further subdivide those numbers by test categories, such as basic, functionality, and robustness.

13.4.2 Test Execution Metric Examples

We give examples of test case execution metrics from a real-life test project called Bazooka. A total of 1592 test cases were designed to be executed. The projected execution of the 1592 test cases on a weekly basis for a 14-week test cycle is shown



	wk1	wk2	wk3	wk4	wk5	wk6	wk7	wk8	wk9	wk10	wk11	wk12	wk13	wk14	wk15	wk16
PE	25	100	150	250	400	600	800	950	1100	1250	1400	1500	1550	1592	1592	1592
AE	5	45	121	297	542	841	991	1148	1254	1324	1426	1515	1566	1581	1581	1591

Figure 13.3 PAE metric of Bazooka (PE: projected execution; AE: actually executed) project.

in Figure 13.3 in the form of a cumulative bar chart for the PAE metric. Twenty-five test cases are planned to be executed in the first week, 75 in the second week. A total of 100 test cases are planned to be executed by the end of first 2 weeks. The second row at the bottom of Figure 13.3, those actually executed, is updated. One should monitor the numbers of projected and actually executed test cases on a weekly basis during the execution cycles. If there is a large difference between the two, immediate action can be taken to understand its root cause and improve the execution rate before it is too late. The test execution rate was lower than the planned rate, as shown in the table, for the first 3 weeks. It took approximately 3 weeks for the test engineers to understand the whole system and come up to speed. The test project took 16 weeks, instead of the planned 14, to complete one execution cycle for all the test cases, including reexecution of all the failed ones. Initially, the test engineers did not run the test cases for the stress, load, and stability categories because some of the fixes concerning memory leak were checked in only at the end of week 13. Therefore, the test cycle was extended for 2 more weeks to ensure that the system could stay up without any memory leaks for an extended period of time.

Let us consider our previous example, the Bazooka test project, for which we had selected 1592 test cases. The EST metric that we monitor on a weekly basis for each test group concerns the number of test cases in the following states: passed, failed, invalid, blocked, and untested. As an example, the test execution status of week 4 is shown in Table 13.5.

TABLE 13.5 EST Metric in Week 4 of Bazooka Project

Test Groups	Total Number of Test Cases	Passed	Failed	Blocked	Invalid	Executed	Untested	Passed/ Executed (%)
Basic	156	64	9	0	0	73	83	87.67
Functionality	480	56	7	0	0	63	417	88.89
Robustness	230	22	1	0	0	23	207	95.65
Interoperability	149	15	2	0	0	17	132	88.24
Load and stability	43	1	0	0	0	1	42	100.00
Performance	54	0	0	0	0	0	54	0.00
Stress	36	0	0	0	0	0	36	0.00
Scalability	54	5	0	0	0	5	49	100.00
Regression	356	93	13	0	0	106	250	87.74
Documentation	34	8	1	0	0	9	25	88.89
Total	1592	264	33	0	0	297	1295	88.89

We explain how the Bazooka team controlled the progress of system testing. A revert criterion for a test cycle is a predicate on the quality level of a product in terms of percentage of executed test cases that have passed. If such a criterion holds, the test cycle is suspended. An example of a revert criteria is: *At any instant during the test cycle, if the cumulative failure rate reaches 20%, the cycle will restart all over after the reported defects are claimed to have been fixed*. We show the percentage of executed test cases that have passed in the rightmost column of Table 13.6. The pass rate was 71.11% at the end of the second week of Bazooka, which implies that the failed rate was 28.89%. Considering the example revert criteria given above, the test cycle should have been abandoned. However, as a special case, the Bazooka project management team made a decision not to abandon the test cycle for the following reasons:

- Only 45 out of 1592 test cases had been executed.
- Some of the test cases were considered to have failed due to procedural errors on the part of test engineers; these test cases were eventually considered to have passed.

A unanimous decision was made by the Bazooka team to wait for one more week and to monitor the test results on a daily basis. The pass rate went up to 89.26% by the end of the third week, as shown in Table 13.6. Since the failure rate was well below the 20% threshold for the revert criteria to be activated, the test cycle was continued until its completion.

The above example tells us that it is important to analyze the test metrics, rather than take decisions based on the raw data. It is interesting to note that the highest cumulative failure rate of 19.5% was observed in week 6, and 841 test cases—or 52% of the total number of test cases to be executed—were already executed. The system test cycle would have been abandoned with an increase of

TABLE 13.6 EST Metric in Bazooka Monitored on Weekly Basis

Week	Executed	Passed	Failed	Executed/ Total (%)	Passed/ Total (%)	Passed/ Executed (%)
1	5	5	0	0.31	0.31	100.00
2	45	32	13	2.83	2.01	71.11
3	121	108	13	7.60	6.78	89.26
4	297	264	33	18.66	16.58	88.89
5	542	451	91	34.05	28.33	83.21
6	841	677	164	52.83	42.53	80.50
7	991	835	156	62.25	52.45	84.26
8	1148	1009	139	72.11	63.38	87.89
9	1254	1096	158	78.77	68.84	87.40
10	1324	1214	110	83.17	76.26	91.69
11	1426	1342	84	89.57	84.30	94.11
12	1515	1450	65	95.16	91.08	95.71
13	1566	1519	47	98.37	95.41	97.00
14	1581	1567	14	99.31	98.43	99.11
15	1581	1570	11	99.31	98.62	99.30
16	1591	1580	11	99.94	99.25	99.31

Note: Total valid test cases 1592.

just 0.5% in the failure rate. Test cases concerning the most vulnerable areas of the system were executed on a priority basis in the beginning of the cycle. Therefore, a maximum number of 164 test cases failed within the first 6 weeks of testing. New images with fixes were released for system testing after the week 6. The failed test cases were reexecuted, and the system passed those tests, thus improving the pass rate, as shown in the rightmost column of Table 13.6. Only one test case from the stress group was still in the untested state by the end of week 16.

13.4.3 Metrics for Monitoring Defect Reports

Queries can be developed to obtain various kinds of information from the database after a defect tracking system is put in place. As the system test engineers submit defects, the defects are further analyzed. Useful data can be gathered from the analysis process to quantify the quality level of the product in the form of the following metrics:

- **Function as Designed (FAD) Count:** Often test engineers report defects which are not really true defects because of a misunderstanding of the system, called FADs. If the number of defects in the FAD state exceeds, say, 10% of the submitted defects, we infer that the test engineers have an inadequate understanding of the system. The lower the FAD count, the higher the level of system understanding of the test engineers.

- **Irreproducible Defects (IRD) Count:** One must be able to reproduce the corresponding failure after a defect is reported so that developers can understand the failure to be able to fix it. If a defect cannot be reproduced, then the developers may not be able to gain useful insight into the cause of the failure. Irreproducibility of a defect does not mean that the defect can be overlooked. It simply means that the defect is an intricate one and it is very difficult to fix it. The number of defects in the irreproducible state, or hidden-defect count, is a measure of the unreliability of the system.
- **Defects Arrival Rate (DAR) Count:** Defect reports arrive from different sources during system testing: system testing group (ST), software development group (SW), SIT group, and others with their own objectives in mind. The "others" group includes customer support, marketing, and documentation groups. Defects reported by all these groups are gathered and the percentage of defects reported by each group on a weekly basis computed. This is called the rate of defects reported by each group.
- **Defects Rejected Rate (DRR) Count:** The software development team makes an attempt to fix the reported defects by modifying the existing code and/or writing more code. A new software image is available for further system testing after these defects are fixed. The system testing group verifies that the defects have been actually fixed by rerunning the appropriate tests. At this stage the system testing team may observe that some of the supposedly fixed defects have not actually been fixed. Such defects are used to produce the DRR count. The DRR count represents the extent to which the development team has been successful in fixing defects. It also tells us about the productivity of the software development team in fixing defects. A high DRR count for a number of weeks should raise an alert because of the high possibility of the project slipping out of schedule.
- **Defects Closed Rate (DCR) Count:** The system testing group verifies the resolution by further testing after a defect is claimed to be resolved. The DCR metric represents the efficiency of verifying the claims of defect fixes.
- **Outstanding Defects (OD) Count:** A reported defect is said to be an outstanding defect if the defect continues to exist. This metric reflects the prevailing quality level of the system as system testing continues. A diminishing OD count over individual test cycles is an evidence of the increasingly higher level of quality achieved during the system testing of a product.
- **Crash Defects (CD) Count:** The defects causing a system to crash must be recognized as an important category of defects because a system crash is a serious event leading to complete unavailability of the system and possibly loss of data [4]. This metric, generally known as a stability metric, is very useful in determining whether to release the system with its current level of stability.

- **Arrival and Resolution of Defects (ARD) Count:** It is useful to compare the rate of defects found with the rate of their resolution [5]. Defect resolution involves rework cost which should be minimized by doing the first time development work in a better way. Hence, this metric is useful for the development team and helps them to estimate the time it may take to fix all the defects. Actions are taken to expedite the defect resolution process based on this information.

13.4.4 Defect Report Metric Examples

We give examples for most of the above metrics by using data from a second test project called Stinger. The weekly DAR from each group of the organization performing system-level testing in the second test cycle of execution for 16 weeks is shown in Table 13.7. The total remaining open defects from the first test cycle are given in the first row of the table as week 0. The average number of defects filed per week by ST, SW, SIT, and others are 57, 15, 10, and 18, respectively. Here, the term "others" includes members from customer support, marketing, and documentation groups. The arrival rates of the defects in the first 6 weeks are much higher than the rest of the test cycle. This is due to the fact that early in the test cycle test cases are prioritized to flush out defects in the more vulnerable portions of the system.

The DRR count for the Stinger project is shown in Table 13.8. The average rate of defect rejection was 5.24%. The rejected rate was almost 6% during the

TABLE 13.7 DAR Metric for Stinger Project

	ST		SW		SIT		Others		Total
Week	Number of Defects	Percentage	Number of Defects	Percentage	Number of Defects	Percentage	Number of Defects	Percentage	Number of Defects
0	663	69.21	172	17.95	88	9.19	35	3.65	958
1	120	79.47	28	18.54	2	1.32	1	0.66	151
2	99	55.00	44	24.44	31	17.22	6	3.33	180
3	108	53.73	54	26.87	31	15.42	8	3.98	201
4	101	65.16	16	10.32	22	14.19	16	10.32	155
5	107	57.22	21	11.23	26	13.90	33	17.65	187
6	108	56.25	16	8.33	35	18.23	33	17.19	192
7	59	57.84	0	0.00	1	0.98	42	41.18	102
8	15	31.91	9	19.15	3	6.38	20	42.55	47
9	28	59.57	0	0.00	1	2.13	18	38.30	47
10	14	50.00	4	14.29	0	0.00	10	35.71	28
11	12	48.00	1	4.00	0	0.00	12	48.00	25
12	59	53.64	5	4.55	9	8.18	37	33.64	110
13	28	54.90	5	9.80	0	0.00	18	35.29	51
14	23	51.11	0	0.00	0	0.00	22	48.89	45
15	12	31.58	21	55.26	3	7.89	2	5.26	38
16	24	38.71	23	37.10	0	0.00	15	24.19	62
Sum	917		247		164		293		1,621
Avearge	57	56.57	15	15.24	10	10.12	18	18.08	

TABLE 13.8 Weekly DRR Status for Stinger Test Project

Week	Number of Defects Verified	Number of Defects Closed	Number of Defects Rejected	Defects Rejected (%)
1	283	269	14	4.95
2	231	216	15	6.49
3	266	251	15	5.64
4	304	284	20	6.58
5	194	183	11	5.67
6	165	156	9	5.45
7	145	136	9	6.21
8	131	122	9	6.87
9	276	262	14	5.07
10	160	152	8	5.00
11	52	49	3	5.77
12	80	74	6	7.50
13	71	71	0	0.00
14	60	58	2	3.33
15	121	119	2	1.65
16	97	96	1	1.03
Sum	2636	2498	138	
Avearge	165	156	9	5.24

first 8 weeks, and for the next 4 weeks it was about 5%. For the remainder of the test cycle, the rejection rate dropped near 1.5%.

The OD counts are given in Table 13.9. Critical-, high-, medium-, and low-priority defects are denoted by P1, P2, P3, and P4, respectively. The *priority* level is a measure of how soon the defects are to be fixed. The outstanding defect counts carried forward from the first test cycle into the second cycle are given in the first row of the table as week 0 statistics. It is evident from this metric that the total number of outstanding defects gradually diminished from 958 to 81 within 16 weeks.

The number of crashes observed by different groups during the final 8 weeks of testing is shown in Table 13.10. It is evident from the table that the total number of crashes gradually diminished from 20 per week to 1 per week. One has to give these crashes a closer look to determine the root cause of each of the individual crashes. The development team must investigate these crashes individually.

Finally, we give an example of the ARD metric by considering a third test project called Bayonet. The projected number of defects submitted, resolved, and remaining open in the first four weeks of a test cycle are given in the upper half of Table 13.11. The projected numbers were derived from an earlier test project [6] that had very similar characteristics to Bayonet. At the start of the test cycle, the total number of open defects is 184. Out of these 184 defects, 50 defects are either of P1 or of P2 priority level, 63 defects are of level P3, and 71 defects are of

TABLE 13.9 Weekly OD on Priority Basis for Stinger Test Project

	P1		P1		P3		P4		Total
Week	Number of Defects	Percentage	Number of Defects	Percentage	Number of Defects	Percentage	Number of Defects	Percentage	Number of Defects
0	14	1.46	215	22.44	399	41.65	330	34.45	958
1	19	2.26	164	19.52	280	33.33	377	44.88	840
2	17	2.11	138	17.16	236	29.35	413	51.37	804
3	11	1.46	108	14.32	181	24.01	454	60.21	754
4	7	1.12	44	7.04	123	19.68	451	72.16	625
5	5	0.79	28	4.45	123	19.55	473	75.20	629
6	10	1.50	33	4.96	123	18.50	499	75.04	665
7	11	1.74	83	13.15	52	8.24	485	76.86	631
8	5	0.90	34	6.12	32	5.76	485	87.23	556
9	8	2.35	21	6.16	39	11.44	273	80.06	341
10	4	1.84	21	9.68	36	16.59	156	71.89	217
11	4	2.07	26	13.47	38	19.69	125	64.77	193
12	3	1.31	28	12.23	84	36.68	114	49.78	229
13	3	1.44	40	19.14	84	40.19	82	39.23	209
14	10	5.10	42	21.43	68	34.69	76	38.78	196
15	2	1.74	34	29.57	38	33.04	41	35.65	115
16	3	3.70	25	30.86	24	29.63	29	35.80	81

TABLE 13.10 Weekly CD Observed by Different Groups for Stinger Test Project

	S/W		ST		SIT		Others		Total
Week	Number of Defects	Percentage	Number of Defects	Percentage	Number of Defects	Percentage	Number of Defects	Percentage	Number of Crashes
9	6	30.00	8	40.00	6	30.00	0	0.00	20
10	5	31.25	6	37.50	3	18.75	2	12.50	16
11	4	25.00	9	56.25	0	0.00	3	18.75	16
12	3	33.33	4	44.44	0	0.00	2	22.22	9
13	2	22.22	6	66.67	0	0.00	1	11.11	9
14	1	14.29	2	28.57	1	14.29	3	42.86	7
15	0	0.00	2	66.67	1	33.33	0	0.00	3
16	0	0.00	1	100.00	0	0.00	0	0.00	1
Sum	21		38		11		11		81
Average	3	25.93	5	46.91	1	13.58	1	13.58	10

level P4. The actual numbers of submitted and resolved defects are shown in the lower half of the same table. The actual numbers of open defects are slightly lower than the projected numbers after the completion of the first week. However, the actual number of resolved defects is much lower than the projected number in the second week. This suggests that the software developers were slower in resolving the defects. The total number of open defects at the end of the second week is higher than the estimated number. At this stage the software development manager must take actions to improve the defect resolution rate by either increasing the working hours or moving software developers from another project to Bayonet. In

TABLE 13.11 ARD Metric for *Bayonet*

		Submitted				Resolved				Open			
Week	Build	Total	P1+P2	P3	P4	Total	P1+P2	P3	P4	Total	P1+P2	P3	P4
							Projected						
										184	50	63	71
1	build10	75	24	36	15	118	38	60	20	141	36	39	66
2	build11	75	24	36	15	118	38	60	20	98	22	15	61
3	build12	75	24	36	15	95	38	37	20	78	8	14	56
4	build13	14	5	7	2	20	10	5	5	72	3	16	53
							Actual						
1	build10	67	23	27	17	119	37	49	33	132	36	41	55
2	build11	77	26	36	15	89	37	37	15	120	25	40	55
3	build12	62	18	32	12	100	36	52	12	82	7	20	55
4	build13	27	11	12	4	33	15	8	10	76	3	24	49

this case, the manager moved an experienced software developer to this project to expedite the resolution of outstanding defects. It is advisable to have the upper portion of the table in place before the start of a test cycle. The second half of the table evolves as the test cycle progresses so that everyone in the development team becomes aware of the progress of defect resolution.

13.5 ORTHOGONAL DEFECT CLASSIFICATION

Orthogonal defect classification (ODC) [7] is a methodology for rapid capturing of the semantics of each software defect. The ODC methodology provides both a classification scheme for software defects and a set of concepts that provide guidance in the analysis of the classified aggregate defect data. Here, orthogonality refers to the nonredundant nature of the information captured by the defect attributes and their values that are used to classify defects. The classification of defects occurs at two different points in time during the life cycle of a defect. First, a defect is put in its initial new state when it is discovered, where the circumstances leading up to the exposure of the defect and the likely impact are typically known. Second, a defect is moved to the resolved state, where the exact nature of the defect and the scope of the fix are known. ODC categories capture the semantics of a defect from these two perspectives. In the new state, the submitter needs to fill out the following ODC attributes, or fields:

- **Activity:** This is the actual activity that was being performed at the time the defect was discovered. For example, the developer might decide to do a code review of a particular procedure during the system testing phase. In this case, the term would be "system testing," whereas the activity is "code review."

- **Trigger:** The environment or condition that had to exist for the defect to surface. Triggers describe the requirements to reproduce the defect.
- **Impact:** This refers to the effect the defect would have on the customer if the defect had escaped to the field.

The owner of the defect moves the defect to the resolved state when the defect has been fixed and needs to fill out the following ODC attributes, or fields:

- **Target:** The target represents the high-level identity, such as design, code, or documentation, of the entity that was fixed.
- **Defect Type:** The defect type represents the actual correction that was made.
- **Qualifier:** The qualifier specifies whether the fix was made due to missing, incorrect, or extraneous code.
- **Source:** The source indicates whether the defect was found in code developed in-house, reused from a library, ported from one platform to another, or provided by a vendor.
- **Age:** The history of the design or code that had the problem. The age specifies whether the defect was found in new, old (base), rewritten, or refixed code:
  1. **New:** The defect is in a new function which was created by and for the current project.
  2. **Base:** The defect is in a part of the product which has not been modified by the current project and is not part of a standard reuse library. The defect was not injected by the current project, and therefore it was a latent defect.
  3. **Rewritten:** The defect was introduced as a direct result of redesigning and/or rewriting an old function in an attempt to improve its design or quality.
  4. **Refixed:** The defect was introduced by the solution provided to fix a previous defect.

The ODC attributes can be collected by modifying an existing defect tracking tool that is being used for modeling defects. As the data are collected, those must be *validated* and *assessed* on a regular basis. The individually classified defects are reviewed to ensure the consistency and correctness of the classification. The data are ready for assessment after those have been validated. The assessment is not done against each individual defect; rather trends and patterns in the aggregate data are studied. Data assessment of ODC classified data is based on the relationships of the ODC attributes to one another and to non-ODC attributes, such as category, severity, and defect submit date [8]. For example, the relationships among the attributes of defect type, qualifier, category, submit date, and severity of defects need to be considered to evaluate the product stability.

The ODC assessment must be performed by an analyst who is familiar with the project and has an interest in data analysis. A good user-friendly tool for

visualizing data is needed. The ODC analyst must provide regular feedback, say, on a weekly basis, to the software development team so that appropriate actions can be taken. Once the feedback is given to the software development team, they can then identify and prioritize actions to be implemented to prevent defects from recurring.

The ODC along with application of the Pareto analysis technique [9, 10] gives a good indication of the parts of the system that are error prone and, therefore, require more testing. Juran [9] stated the Pareto principle very simply as "concentrate on the vital few and not the trivial many." An alternative expression of the principle is to state that 80% of the problems can be fixed with 20% of the effort, which is generally called the 80–20 rule. This principle guides us in efficiently utilizing the effort and resources. As an example, suppose that we have data on test *category* and the frequency of occurrence of defects for a hypothetical Chainsaw system test project as shown in Table 13.12. The data plotted on a Pareto diagram, shown in Figure 13.4, which is a bar graph showing the frequency of occurrence of defects with the most frequent ones appearing first. Note that the *functionality* and the *basic* groups have high concentration of defects. This information helps system test engineers to focus on the functionality and the basic category parts of the Chainsaw test project. The general guideline for applying Pareto analysis is as follows:

- Collect ODC and non-ODC data relevant to problem areas.
- Develop a Pareto diagram.
- Use the diagram to identify the vital few as issues that should be dealt with on an individual basis.
- Use the diagram to identify the trivial many as issues that should be dealt with as classes.

TABLE 13.12 Sample Test Data of Chainsaw Test Project

Category	Number of Defect Occurrences
1. Basic	25
2. Functionality	48
3. Robustness	16
4. Interoperability	4
5. Load and stability	6
6. Performance	12
7. Stress	6
8. Scalability	7
9. Regression	3
10. Documentation	2

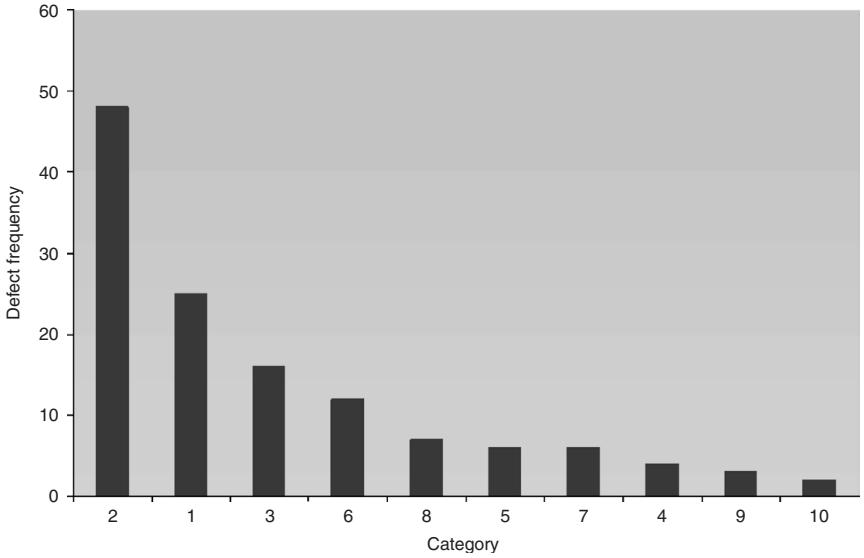


Figure 13.4 Pareto diagram for defect distribution shown in Table 13.12.

Remark. Vilfredo Federico Damaso Pareto was a French–Italian sociologist, economist, and philosopher. He made several important contributions, especially in the study of income distribution and in the analysis of individuals' choices. In 1906 he made the famous observation that 20% of the population owned 80% of the property in Italy, later generalized by Joseph M. Juran and others into the so-called Pareto principle (also termed the 80–20 rule) and generalized further to the concept of a Pareto distribution.

13.6 DEFECT CAUSAL ANALYSIS

The idea of defect causal analysis (DCA) in software development is effectively used to raise the quality level of products at a lower cost. Causal analysis can be traced back to the quality control literature [11] as one of the quality circle activities in the manufacturing sector. The quality circle concept is discussed in Section 1.1. The causes of manufacturing defects are analyzed by using the idea of a quality circle, which uses *cause–effect* diagrams and Pareto diagrams. A cause–effect diagram is also called an *Ishikawa* diagram or a *fishbone* diagram. Philip Crosby [12] described a case study of an organization called "HPA Appliance" involving the use of causal analysis to prevent defects on manufacturing lines. Causal analysis of software defects is practiced in a number of Japanese companies, usually in the context of quality circle activities [13, 14].

The idea of DCA was developed at IBM [15]. Defects are analyzed to (i) determine the cause of an error, (ii) take actions to prevent similar errors from

occurring in the future, and (iii) remove similar defects that may exist in the system or detect them at the earliest possible point in the software development process. Therefore, DCA is sometimes referred to as defect prevention or defect RCA [16]. The importance of DCA has been succinctly expressed by Watts S. Humphrey [17] as follows: "While detecting and fixing defects is critically important, it is an inherently defensive strategy. To make significant quality improvements, you should identify the causes of these defects and take steps to eliminate them."

In his tutorial article [18], David N. Card explains a five-step process for conducting DCA. Card suggests three key principles to drive DCA:

- **Reduce the number of defects to improve quality:** One can define software quality in terms of *quality factors* and *attributes*, but it is needless to say that a product with many defects lacks quality. Software quality is improved by focusing on defect prevention and early detection of defects.
- **Apply local expertise where defects occur:** The responsibility of DCA should be given to the software developers who unwittingly contributed to the mistakes. They are best qualified to identify what went wrong and how to prevent it. DCA should take place at the actual location where the failure occurred, instead of in a remote conference room. In other words, DCA should be performed at the source.
- **Focus on systematic errors:** Intuitively, an error is said to be a systematic error if the same error or similar errors occur on many occasions. Systematic errors account for a significant portion of the defects found in software projects. Identifying and preventing systematic errors do have a huge impact on the quality of the product.

As the term suggests, DCA focuses on understanding the cause–effect relationship [19]. Three conditions are established to demonstrate a causal relationship between a cause and a symptom in the form of a failure:

- There must be a correlation between the hypothesized cause and the effect.
- The hypothesized cause must precede the effect in time.
- The mechanism linking the cause to the effect must be identified.

The first condition indicates that the effect is likely to be observed when a cause occurs. The second condition is obvious. The third condition requires investigation, which consists of five steps to select systematic errors associated with common failures and tracing them back to their original cause:

(i) *When was the failure detected and the corresponding fault injected?* Note the development phase, that is, code review, unit testing, integration testing, or system testing, where the failure is observed. Next, determine the phase of development in which the corresponding fault was injected. Developers identify the more problematic areas by considering the above information.

(ii) *What scheme is used to classify errors?* We identify the classes of important errors in this step. Grouping errors together helps us in identifying

clusters in which common problems are likely to be found. One can use Pareto diagrams to identify problem clusters. Unlike the ODC scheme, DCA does not use a predefined set of error categories. The objective of DCA is not to analyze percentages of errors in each categories, but rather to analyze each error to understand the reason the error occurred and take preventive measures. Categorization of errors is based on the nature of the work performed, the opportunities for making mistakes, and the current dynamics of the project. Suppose that during our analysis we notice that there are many errors related to the program interface. Then we have two levels of error classification: (i) a coarse-grain classification called interface errors and (ii) many fine-grained subcategories, such as construction, misuse of interface, and initialization, as discussed in Section 7.2.

(iii) *What is the common problem (systematic error)*? The set of problem reports comprising a cluster of errors, identified in the second step above, gives an indication of a systematic error. In general, systematic errors are associated with the specific functionality of a product. Ignore the effects of individual errors in seeking out systematic errors during the identification process.

(iv) *What is the principal cause of the errors*? We endeavor to trace the error back to its cause and understand the root cause associated with many errors in the cluster. Finding the causes of errors requires (i) thorough understanding of the process and the product and (ii) experience and judgment. In order to achieve this goal, we may need to draw a cause–effect diagram as shown in Figure 13.5. First, the problem being analyzed must be stated and then the main branch of the diagram connected to the problem statement is drawn. Second, the generic causes to be used in the diagram are added. The generic causes usually fall into four categories as Ishikawa has identified [11]:

- **Tools:** The tools used in developing the system may be clumsy, unreliable, or defective.
- **Input:** It may be incomplete, ambiguous, or defective.
- **Methods:** The methods may be incomplete, ambiguous, wrong, or unenforced.
- **People:** They may lack adequate training or understanding.

We need to brainstorm to collect specific causes and attach the specific causes to the appropriate generic causes. The significance of the causes identified is discussed after all the ideas have been explored, focusing on the principal causes contributing to the systematic errors. Often this is found in the denser cluster of causes in the diagram. The cause–effect diagram provides a focused way to address the problem.

(v) *How could the error be prevented in the future?* Actions, which are preventive, corrective, or mitigating, are taken after the principal cause of a systematic error has been identified:

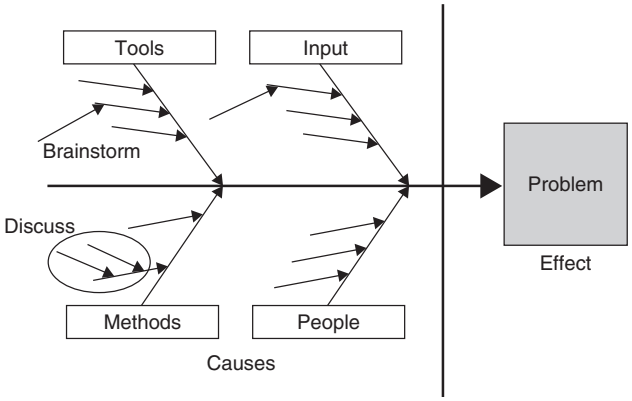


Figure 13.5 Cause–effect diagram for DCA.

- **Preventive:** A preventive action reduces the chances of similar problems occurring in the future. The idea is to detect a problem earlier and fix it.
- **Corrective:** A corrective action means fixing the problems. Attacking the cause itself may not be cost-effecive in all situations.
- **Mitigating:** A mitigating action tries to counter the adverse consequences of problems. Mitigating actions are a part of a risk management activity [20].

A combination of preventive, corrective, and mitigating actions may be applied, depending on the cost of taking actions and the magnitude of the symptoms observed. A comprehensive action plan is created to promote either prevention or detection of systematic errors at the earliest possible time in the software development process. The following are typical questions to ask at this point to elicit recommendations:

- What is the earliest a defect could have been recognized?
- What symptoms suggest that the defect may occur again?
- What did we need to know to avoid the defect?
- How could we have done things differently?

The recommended actions need to be specific and precise for others to be able to follow. Some common examples of preventive and corrective actions are as follows:

- **Training:** This allows people to improve their knowledge about the product. Training may include concrete actions such as organizing a seminar series on the components of the product, preparing a technical write-up on a complex aspect of the product, and writing an article about repetitive errors.

- **Improvement in Communication:** These are actions that improve communication procedures within the product organizations. One can think of notifying programmers and testers via email when the interface specification changes. Holding weekly team meetings helps in clarifying and disseminating useful information.
- **Using Tools:** Actions are taken to develop new tools or enhance existing tools that support the processes. One can develop or buy a tool to check memory utilization or add a new coverage metric to the module coverage tool.
- **Process Improvements:** These are actions that improve existing processes and define new processes. Such actions might modify the design change process or add new items to a common error list.

13.7 BETA TESTING

Beta testing is conducted by the potential buyers prior to the official release of the product. The purpose of beta testing is not to find defects but to obtain feedback from the field about the usability of the product. There are three kinds of beta tests performed based on the relationships between the potential buyers and the sellers:

- **Marketing Beta:** The purpose here is to build early awareness and interest in the product among potential buyers.
- **Technical Beta:** The goal here is to obtain feedback about the usability of the product in a real environment with different configurations from a small number of friendly customers. The idea is to obtain feedback from a limited number of customers who commit a considerable amount of time and thought to their evaluation.
- **Acceptance Beta:** The idea of this test is to ensure that the product meets its acceptance criteria. It is the fulfillment of the contract between a buyer and a seller. The system is released to a specific buyer, who has a contractual original equipment manufacturer (OEM) agreement with the supplier. Acceptance beta includes the objective of technical beta as well. Acceptance testing is discussed in Chapter 14.

The defects and the progress of test execution provide useful information about the quality of the system. The quality of the system under development is a moving target during test execution. A decision about when to release the system to the beta customers is made by software project team members. The consequences of early and delayed releases are as follows:

- If it is released too early, too many defects in the system may have a negative impact on the prospective customers, and it may generate bad publicity for both the product and the company.
- If the release is delayed, other competitors may capture the market.

The beta release criteria are established by the project team, and the criteria are less stringent than the final system test cycle criteria given in Chapter 13. A framework for writing beta release criteria is given in Table 13.13. All the beta release criteria in Table 13.13 are self-explanatory, except the last item, namely, that all the identified beta blocker defects are in the closed state. The beta blocker defects are the set of defects that must be in the closed state before the software is considered for beta testing. Three weeks before the beta release, the beta blocker defects are identified at the cross-functional project meeting. These defects are tracked on a daily basis to ensure that the defects are resolved and closed before the beta release. In the meantime, as new defects are submitted, these defects are evaluated to determine whether or not these are beta blockers. If it is concluded that a new defect is a beta blocker, then the defect is included on the list of beta blockers and all the defects are tracked for closure.

The system test cycle continues concurrently with beta testing. Weekly meetings are conducted with beta customers by the beta support team, which consists of cross-functional team members of the project. The weekly test status reports from the beta customers are analyzed, and actions are taken to resolve the issues by either fixing the defects or providing them a workaround of the issues raised by the beta customers. If a beta customer decides to deploy the system, an agreement can be reached to provide patch releases by fixing any outstanding deficiency in the system by a certain time frame. A final test cycle with a scaled-down number of test cases is executed as a regression test suite after an agreement is reached, and the product is prepared for first customer shipment (FCS).

TABLE 13.13 Framework for Beta Release Criteria

1. Ninety-eight percent all the test cases have passed.
2. The system has not crashed in the last week of all testing.
3. All the *resolved* defects must be in the closed state.
4. A release note with all the defects that are still in the open state along with workaround must be available. If the workaround does not exist, then an explanation of the impact on the customer must be incorporated in the release note.
5. No defect with "critical" severity is in the open state.
6. No defect with "high" severity has a probability of hitting at a customer site of more than 0.1.
7. The product does not have more than a certain *number* of defects with severity "medium." The number may be determined by the software project team members.
8. All the test cases for performance testing must have been executed, and the results must be available to the beta test customer.
9. A beta test plan must be available from all potential beta customers. A beta test plan is nothing but a user acceptance test plan.
10. A draft of the user guide must be available.
11. The training materials on the system are available for field engineers.
12. The beta support team must be available for weekly meetings with each beta customer.
13. All the identified beta blocker defects are in the closed state.

13.8 FIRST CUSTOMER SHIPMENT

The exit criterion of the final test cycle must be satisfied before the FCS which is established in the test execution strategy section of Chapter 13. An FCS readiness review meeting is called to ensure that the product meets the shipment criteria. The shipment criteria are more than just the exit criteria of the final test cycle. This review should include representatives from the key function groups responsible for delivery and support of the product, such as engineering, operation, quality, customer support, and product marketing. A set of generic FCS readiness criteria is as follows:

- All the test cases from the test suite should have been executed. If any of the test cases is unexecuted, then an explanation for not executing the test case should have been provided in the test factory database.
- Test case results are updated in the test factory database with passed, failed, blocked, or invalid status. Usually, this is done during the system test cycles.
- The requirement database is updated by moving each requirement from the verification state to either the closed or the decline state, as discussed in Chapter 11, so that compliance statistics can be generated from the database. All the issues related to the EC must be resolved with the development group.
- The pass rate of test cases is very high, say, 98%.
- No crash in the past two weeks of testing has been observed.
- No known defect with *critical* or *high* severity exists in the product.
- Not more than a certain number of known defects with *medium* and *low* levels of severity exist in the product. The threshold number may be determined by the software project team members.
- All the identified FCS blocker defects are in the closed state.
- All the resolved defects must be in the closed state.
- All the outstanding defects that are still in either the open or assigned state are included in the release note along with the workaround, if there is one.
- The user guides are in place.
- A troubleshooting guide is available.
- The test report is completed and approved. Details of a test report are explained in the following section.

Once again, three weeks before the FCS, the FCS blocker defects are identified at the cross-functional project meeting. These defects are tracked on a daily basis to ensure that the defects are resolved and closed before the FCS. In the meantime, as new defects are submitted, these are evaluated to determine whether these are FCS blockers. If it is concluded that a defect is a blocker, then the defect is included in the blocker list and tracked along with the other defects in the list for its closure.

13.9 SYSTEM TEST REPORT

During the test execution cycles test reports are generated and distributed as discussed in Sections 13.4.2 and 13.4.4. A final summary report is created after completion of all the test cycles. The structure of the final report is outlined in Table 13.14.

The introduction to the test project section of the test report summarizes the purpose of the report according to the test plan. This section includes the following information:

- Name of the project
- Name of the software image
- Revision history of this document
- Terminology and definitions
- Testing staff
- Scope of the document
- References

The summary of test results section summarizes the test results for each test category identified in the test plan. In each test cycle, the test results are summarized in the form of the numbers of test cases passed, failed, invalid, blocked, and untested and the reasons for not executing some test cases. A test case may not be tested because of several reasons, such as unavailability of equipment and difficulty in creating the test scenario using the available test bed setup. The latter may only be possible at a real beta testing site. For each test cycle, the following data are given in this section:

- Start and completion dates
- Number of defects filed
- Number of defects in different states, such as irreproducible, FAD, closed, shelved, duplicate, postponed, assigned, and open

TABLE 13.14 Structure of Final System Test Report

1. Introduction to the test project
2. Summary of test results
3. Performance characteristics
4. Scaling limitations
5. Stability observations
6. Interoperability of the system
7. Hardware/software compatible matrix
8. Compliance requirement status

Problems still remaining in the system, including any shortcoming or likelihood of failure, are summarized in this section in terms of their levels of severity and their potential impact on the customers. For example, if the password is case insensitive, then it should be stated so. As another example, if an operating system puts a limit of 20 on the number of windows that can be simultaneously opened, then it should be stated that the user should not open more than 20 windows simultaneously; exceeding the limit may cause the system to crash.

In the performance characteristics section of the document, the system response time, throughput, resource utilization, and delay measurement results are described along with the test environments and tools used in the measurement process.

The limits of the system are stated in the scaling limitation section of the document based on the findings of scalability testing. Once again the facilities and tools that are used in scaling testing are described here.

The stability observations section states the following information:

- Uptime in number of hours of the system
- Number of crashes observed by different groups, such as software developer, ST, and SIT, in each week of the test cycles
- Descriptions of symptoms of the defects causing system crash that are in the irreproducible state

We state in the interoperability section of the document those third-party software and hardware systems against which the system interoperability tests were conducted. The list may include software and hardware against which the system will not interoperate.

A table shows what software versions are compatible with what hardware revisions in the hardware/software compatible matrix section. Generation of this table is based on the different kinds of hardware systems and different versions of the software used during system test cycles.

Finally, in the compliance requirement status section of the document compliance statistics are summarized by generating them from the requirements database. The following reports are generated by creating appropriate queries on the database:

- Requirements that are in compliance, partial compliance, or noncompliance with the software image
- Requirements that are verified by testing, analysis, demonstration, and inspection
- Requirements that are covered by each test case of the test suite along with the status of the test case

13.10 PRODUCT SUSTAINING

Once a product is shipped to one of the paying customers and if there is no major issue reported by the customer within a time frame of, say, three weeks,

then the product's general availability (GA) is declared. After that the software project is moved to a new phase called the sustaining phase. The goal of this phase is to maintain the software quality throughout the product's market life. Software maintenance activities occur because software testing cannot uncover all the defects in a large software system. The following three software maintenance activities were coined by Swanson [21]:

- **Corrective:** The process that includes isolation and correction of one or more defects in the software.
- **Adaptive:** The process that modifies the software to properly interface with a changing environment such as a new version of hardware or third-party software.
- **Perfective:** The process that improves the software by the addition of new functionalities, enhancements, and/or modifications to the existing functions.

The first major task is to determine the type of maintenance task to be conducted when a defect report comes in from a user through the customer support group. The sustaining team, which includes developers and testers, is assigned immediately to work on the defect if the defect reported by the customer is considered as *corrective* in nature. The status of the progress is updated to the customer within 24 hours. The group continues to work until a patch with the fix is released to the customer. If the defect reported is considered as either *adaptive* or *perfective* in nature, then it is entered in the requirement database, and it goes though the usual software development phases.

The sustaining test engineers are different from the system test engineers. Therefore, before a product transition occurs, the sustaining engineers are provided with enough training about the product to carry out sustaining test activities. In reality, the test engineers are rotated back and forth between the system test group and the sustaining group. The major tasks of sustaining test engineers are as follows:

- Interact with customers to understand their real environment
- Reproduce the issues observed by the customer in the laboratory environment
- Once the root cause of the problem is known, develop new test cases to verify it
- Develop upgrade/downgrade test cases from the old image to the new patch image
- Participate in the code review process
- Select a subset of regression tests from the existing pool of test cases to ensure that there is no side effect because of the fix
- Execute the selected test cases
- Review the release notes for correctness
- Conduct experiments to evaluate test effectiveness

13.11 MEASURING TEST EFFECTIVENESS

It is useful to evaluate the effectiveness of the testing effort in the development of a product. After a product is deployed in the customer environment, a common measure of test effectives is the number of defects found by the customers that were not found by the test engineers prior to the release of the product. These defects had escaped from our testing effort. A metric commonly used in the industry to measure test effectiveness is the defect removal efficiency (DRE) [22], defined as

$$\text{DRE} = \frac{\text{number of defects found in testing}}{\text{number of defects found in testing} + \text{number of defects not found}}$$

We obtain the number of defects found in testing from the defect tracking system. However, calculating the number of defects not found during testing is a difficult task. One way to approximate this number is to count the defects found by the customers within the first six months of its operation. There are several issues that must be understood to interpret the DRE measure in an useful way:

- Due to the inherent limitations of real test environments in a laboratory, certain defects are very difficult to find during system testing no matter how thorough we are in our approach. One should include these defects in the calculation if the goal is to measure the effectiveness of the testing effort including the limitations of the test environments. Otherwise, these kinds of defects may be eliminated from the calculation.
- The defects submitted by customers that need *corrective* maintenance are taken into consideration in this measurement. The defects submitted that need either *adaptive* or *perfective* maintenance are not real issues in the software; rather they are requests for new feature enhancements. Therefore, defects that need adaptive and perfective maintenance may be removed from the calculation.
- There must be a clear understanding of the duration for which the defects are counted, such as starting from unit testing or system integration testing to the end of system testing. One must be consistent for all test projects.
- This measurement is not for one project; instead it should be a part of the long-term trend in the test effectiveness of the organization.

Much work has been done on the *fault seeding* approach [23] to estimate the number of escaped defects. In this approach the effectiveness of testing is determined by estimating the number of actual defects using an extrapolation technique. The approach is to inject a small number of representative defects into the system and measure the percentage of defects that are uncovered by the sustaining test engineers. Since the sustaining test team remains unaware of the seeding, the extent to which the product reveals the known defects allows us to extrapolate the extent to which it found unknown defects. Suppose that the product contains N defects and K defects are seeded. At the end of the test experiments, the sustaining test team has found n unseeded and k seeded defects. The fault seeding theory

asserts the following:

$$\frac{k}{K} = \frac{n}{N} \Longrightarrow N = n\left(\frac{K}{k}\right)$$

For example, consider a situation where 25 known defects have been deliberately seeded in a system. Let us assume that the sustaining testers detect 20 (80%) of these seeded defects and uncovers 400 additional defects. Then, the total number of estimated defects in the system is 500. Therefore, the product still has $500 - 400 = 100$ defects waiting to be discovered plus 5 seed defects that still exist in the code. Using the results from the seeding experiment, it would be possible to obtain the total number of escaped defects from the system testing phase. This estimation is based on the assumption that the ratio of the number of defects found to the total number of defects is 0.80—the same ratio as for the seeded defects. In other words, we assume that the 400 defects constitute 80% of all the defects found by the sustaining test engineers in the system. The estimated remaining number of defects are still hidden in the system. However, the accuracy of the measure is dependent on the way the defects are injected. Usually, artificially injected defects are manually planted, but it has been proved difficult to implement defect seeding in practice. It is not easy to introduce artificial defects that can have the same impact as the actual defects in terms of difficulty of detection. Generally, artificial defects are much easier to find than actual defects.

Spoilage Metric Defects are injected and removed at different phases of a software development cycle [24]. Defects get introduced during requirements analysis, high-level design, detailed design, and coding phases, whereas these defects are removed during unit testing, integration testing, system testing, and acceptance testing phases. The cost of each defect injected in phase X and removed in phase Y is not uniformly distributed; instead the cost increases with the increase in the distance between X and Y. The delay in finding the dormant defects cause greater harms, and it costs more to fix because the dormant defects may trigger the injection of other related defects, which need to be fixed in addition to the original dormant defects. Therefore, an effective testing method would find defects earlier than a less effective testing method would. Hence an useful measure of test effectiveness is defect age, known as *PhAge*. As an example, let us consider Table 13.15, which shows a scale for measuring age. In this example, a requirement defect discovered during high-level design review would be assigned a PhAge of 1, whereas a requirement defect discovered at the acceptance testing phase would be assigned a PhAge of 7. One can modify the table to accommodate different phases of the software development life cycle followed within an organization, including the PhAge numbers.

If the information about a defect introduction phase can be determined, it can be used to create a matrix with rows corresponding to the defect injected in each phase and columns corresponding to defects discovered in each phase This is often called a *defect dynamic model*. Table 13.16 shows a defect injected–v discovered matrix from an imaginary test project called Boomerang. In this example, there were

TABLE 13.15 Scale for Defect Age

	Phase Discovered							
Phase Injected	Requirements	High-Level Design	Detailed Design	Coding	Unit Testing	Integration Testing	System Testing	Acceptance Testing
Requiremants	0	1	2	3	4	5	6	7
High-level design		0	1	2	3	4	5	6
Detailed design			0	1	2	3	4	5
Coding				0	1	2	3	4

TABLE 13.16 Defect Injection versus Discovery on Project Boomerang

	Phase Discovered								
Phase Injected	Requirements	High-Level Design	Detailed Design	Coding	Unit Testing	Integration Testing	System Testing	Acceptance Testing	Total Defects
Requiremants	0	7	3	1	0	0	2	4	17
High-level design		0	8	4	1	2	6	1	22
Detailed design			0	13	3	4	5	0	25
Coding				0	63	24	37	12	136
Summary	0	7	11	18	67	30	50	17	200

seven requirement defects found in high-level design, three in detailed design, one in coding, two in system testing, and four in acceptance testing phases.

Now a new metric called *spoilage* [25, 26] is defined to measure the defect removal activities by using the defect age and defect dynamic model. The spoilage metric is calculated as

$$\text{Spoilage} = \frac{\sum(\text{number of defects} \times \text{discovered PhAge})}{\text{total number of defects}}$$

Table 13.17 shows the spoilage values for the Boomerang test project based on the number of defects found (Table 13.15) weighted by defect age (Table 13.16). During acceptance testing, for example, 17 defects were discovered, out of which 4 were attributed to defects injected during the requirements phase of the Boomerang project. Since the defects that were found during acceptance testing could have been found in any of the seven previous phases, the requirement defects that were dormant until the acceptance testing were given a PhAge of 7. The weighted number of requirement defects revealed during acceptance testing is 28, that is, $7 \times 4 = 28$. The spoilage values for requirements, high-level design, detail design, and coding phases are $3.2, 2.8, 2.0,$ and 1.98, respectively. The spoilage value for the Boomerang test project is 2.2. A spoilage value close to 1 is an indication of

TABLE 13.17 Number of Defects Weighted by Defect Age on Project Boomerang

	Phase Discovered										
Phase Injected	Requirements	High-Level Design	Detailed Design	Coding	Unit Testing	Integration Testing	System Testing	Acceptance Testing	Weight	Total Defects	Spoilage as Weight/Total Defects
Requiremants	0	7	6	3	0	0	12	28	56	17	3.294117647
High-level design		0	8	8	3	8	30	6	63	22	2.863636364
Detailed design			0	13	6	12	20	0	51	25	2.04
Coding				0	63	48	111	48	270	136	1.985294118
Summary	0	7	14	24	72	68	173	82	440	200	2.2

a more effective defect discovery process. As an absolute value, the spoilage metric has little meaning. This metric is useful in measuring the long-term trend of test effectiveness in an organization.

13.12 SUMMARY

This chapter began with a state transition model which allows us to represent each phase in the life cycle of a defect by a state. At each state of the transition model, certain actions are taken by the owner; the defect is moved to a new state after the actions are completed. We presented a defect schema that can be used to monitor various defect metrics. Two key concepts involved in modeling defects are priority and severity. On the one hand, a priority level is a measure of how soon a defect needs to be fixed, that is, urgency. On the other hand, a severity level is a measure of the extent of the detrimental effect of the defect on the operation of the product.

Then we explored the idea of test preparedness to start system-level testing. We discussed the idea of a test execution working document that needs to be maintained before the start of the system test execution in order to ensure that system test engineers are ready to start the execution of system tests. We recommend two types of metrics to be monitored during the execution of system-level test cases:

- Metrics for monitoring test execution
- Metrics for monitoring defects

The first kind of metric concerns the process of executing test cases, whereas the second kind concerns the defects found as a result of test execution. We provided examples of test case execution and defect report metrics from different real-life test projects.

Next, we examined three kinds of defect analysis techniques: ODC, causal, and the Pareto methodology. Causal analysis is conducted to identify the root cause of the defects and to take actions to eliminate the sources of defects; this is done at the time of fixing defects. In the ODC method, assessment is not done against individual defects; rather, trends and patterns in the aggregate data are studied. The Pareto principle is based on the postulate that 80% of the problems can be fixed with 20% of the effort. We showed that ODC along with the application of Pareto analysis can give a good indication of which parts of the system are error prone and may require more testing.

We provided frameworks for beta releases and discussed the process of beta testing as is conducted at the customer's site. We classified three types of beta testing: marketing beta, technical beta, and acceptance beta. The purpose of marketing beta is to build early awareness and interest in the product among potential buyers. A technical beta test is performed to obtain feedback about the usability of the product in a real environment with different configurations. The idea is to obtain feedback from a limited number of users who commit a considerable amount of time and thought to their evaluation. An acceptance beta test is performed to

ensure that the product is acceptable, that is, the product meets its acceptance criteria. Acceptance testing is the fulfillment of the contract between a buyer and a seller.

Next, we provided a detailed structure of the system test report, which must be generated before the product's general availability is declared. Once the product is declared as generally available, the software project is moved to a new phase called the sustaining phase for maintenance. During this phase sustaining test engineers are responsible for carrying out sustaining test activities. We explained the tasks of sustaining test engineers in this chapter.

Finally, we provided two metrics used to measure test effectiveness: defect removal efficiency and spoilage. The objective of a test effectiveness metric is to evaluate the effectiveness of the system testing effort in the development of a product, that is, the number of defects that had escaped from our testing effort. We presented the fault seeding approach in order to estimate the number of escaped defects.

LITERATURE REVIEW

In reference [4] a set of in-process metrics for testing phases of the software development process are described. For each metric the authors describe its purpose, data, interpretation, and use and present graphical examples with real-life data. These metrics are applicable to most software projects and are an integral part of system-level testing.

In reference [8] three case studies are presented to measure test effectiveness using ODC. All three case studies highlight how technical teams can use ODC data for objective feedback on their development processes and the evolution of their product. The case studies include background information on the products, how their ODC process was implemented, and the details of the assessments, including actions that resulted.

Several excellent examples of defect RCA efforts in software engineering have been published. Interested readers are recommended to go through the following articles for more information:

M. Leszak, D. E. Perry, and D. Stoll, "Classification and Evaluation of Defects in a Retrospective," *Journal of Systems and Software*, Vol. 61, April 2002, pp. 173–187.

W. D. Yu, "A Software Prevention Approach in Coding and Root Cause Analysis," *Bell Labs Technical Journal*, Vol. 3, April 1998, pp. 3–21.

The article by Leszak et al. describes a retrospective process with defect RCA, process metric analysis, and software complexity analysis of a network element as a part of an optical transmission network at Lucent Technologies (now Alcatel-Lucent). The article by Yu discusses the guidelines developed by the team to conduct defect RCA at various Lucent switching development organizations.

Mills's seminal article on *fault seeding* is reprinted in *Software Productivity* (H. D. Mills, Little, Brown and Company, Boston, 1983). The book consists of articles on the subject of software process written by the author over many years.

REFERENCES

1. D. Lemont. *CEO Discussion—From Start-up to Market Leader—Breakthrough Milestones*. Ernst and Young Milestones, Boston, May 2004, pp. 9–11.
2. G. Stark, R. C. Durst, and C. W. Vowell. Using Metrics in Management Decision Making. *IEEE Computer*, September 1994, pp. 42–48.
3. S. H. Kan. *Metrics and Models in Software Quality Engineering*, 2nd ed. Addison-Wesley, Reading, MA, 2002.
4. S. H. Kan, J. Parrish, and D. Manlove. In-Process Metrics for Software Testing. *IBM Systems Journal*, January 2001, pp. 220–241.
5. R. Black. *Managing Testing Process*, 2nd ed. Wiley, New York, 2002.
6. E. F. Weller. Using Metrics to Manage Software Projects. *IEEE Computer*, September 1994, pp. 27–33.
7. R. Chillarege, I. S. Bhandari, J. K. Chaar, M. J. Halliday, D. S. Moebus, B. K. Ray, and M. Y. Wong. Orthogonal Defect Classification—A Concept for In-Process Measurement. *IEEE Transactions on Software Engineering*, November 1992, pp. 943–956.
8. M. Butcher, H. Munro, and T. Kratschmer. Improving Software Testing via ODC: Three Case Studies. *IBM Systems Journal*, January 2002, pp. 31–44.
9. J. Juran, M. Gryna, M. Frank, Jr., and R. Bingham, Jr. *Quality Control Handbook*, 3rd ed. McGraw-Hill, New York, 1979.
10. T. McCabe and G. Schulmeyer. The Pareto Principle Applied to Software Quality Assurance. In *Handbook of Software Quality Assurance*, 2nd ed., G. Schulmeyer and J. McManus, Eds. Van Nostrand Reinhold, New York, 1992.
11. K. Ishikawa. *What Is Total Quality Control*. Prentice-Hall, Englewood Cliffs, NJ, 1985.
12. P. Crosby. *Quality Is Free*. New American Library, New York, 1979.
13. K. Hino. Analysis and Prevention of Software Errors as a Quality Circle Activity. *Engineers* (Japanese), January 1985, pp. 6–10.
14. H. Sugaya. Analysis of the Causes of Software Bugs. *Nikkei Computer* (Japanese), August 1985, pp. 167–176.
15. R. G. Mays, C. L. Jones, G. J. Holloway, and D. P. Studinski. Experiences with Defect Prevention. *IBM System Journal*, Vol. **29**, No. 1, 1990, pp. 4–32.
16. M. Pezzé and M. Young. *Software Testing and Analysis: Process, Principles, and Techniques*. Wiley, Hoboken, NJ, 2007.
17. W. S. Humphrey. *A Discipline for Software Engineering*. Addison-Wesley, Reading, MA, 1995.
18. D. N. Card. Learning from Our Mistakes with Defect Causal Analysis. *IEEE Software*, January-February 1998, pp. 56–63.
19. D. N. Card. Understanding Causal Systems. *Crosstalk, the Journal of Defense Software Engineering*, October 2004, pp. 15–18.
20. N. Crockford. *An Introduction to Risk Management*, 2nd ed. Woodhead-Faulkner, Cambridge, U.K. 1986.
21. E. B. Swanson. The Dimensions of Maintenance. In *Proceeding of the Second International Conference on Software Engineering*, October 1976, pp. 492–497.
22. C. Jones. *Applied Software Measurement*. McGraw-Hill, New York, 1991.
23. H. Zhu, P. A. V. Hall, and J. H. R. May. Software Unit Test Coverage and Adequacy. *ACM Computing Survays*, December 1997, pp. 366–427.
24. A. A. Frost and M. J. Campo. Advancing Defect Containment to Quantitative Defect Management. *Crosstalk, the Journal of Defense Software Engineering*, December 2007, pp. 24–28.
25. R. D. Craig and S. P. Jaskiel. *Systematic Software Testing*. Artech House, Norwood, MA, 2002.

26. R. B. Grady and D. L. Caswell. *Software Metrics: Establishing a Company-Wide Program*. Prentice-Hall, Upper Saddle River, NJ, 1998.

Exercises

1. Why is it important to assign both severity and priority levels to a defect?
2. Implement the schema and the defect model discussed in this chapter using a commercially available defect tracking tool (viz., ClearQuest of IBM).
3. Why do in-process metrics need to be tracked on daily or weekly basis during system testing?
4. What is the difference between causal analysis, statistical analysis, and Pareto analysis? The ODC method belongs to which category?
5. Modify the schema shown in Table 13.2, including values of the fields and the state transition diagram in Figure 13.1, to model ODC.
6. The projected number of defects submitted, resolved, and remaining open in the first four weeks of a test project are given in the upper half of Table 13.18. The actual numbers of submitted and resolved defects are shown in the lower half of the table. Calculate the actual number of open defects.
7. For your current system testing project, select a set of in-process metrics discussed in this chapter and track it during the system test execution phase.
8. Develop a set of beta release criteria for your current test project.
9. Write a system test report after the completion of your current test project.
10. In a test project, the number of defects found in different phases of testing are as follows: unit testing, 163 defects; integration testing, 186 defects; system testing, 271 defects. The number of defects found by sustaining test engineers by conducting fault seeding experimentation is 57. What is the value of DRE? Calculate the value of the system test DRE?

TABLE 13.18 ARD Metric for Test Project

		Submitted				Resolved				Open			
Week	Build	Total	P1+P2	P3	P4	Total	P1+P2	P3	P4	Total	P1+P2	P3	P4
							Projected						
										184	50	63	71
1	build10	75	24	36	15	118	38	60	20	141	36	39	66
2	build11	75	24	36	15	118	38	60	20	98	22	15	61
3	build12	75	24	36	15	95	38	37	20	78	8	14	56
4	build13	14	5	7	2	20	10	5	5	72	3	16	53
							Actual						
1	build10	60	26	16	18	105	35	40	30				
2	build11	77	28	34	15	89	37	37	15				
3	build12	62	20	32	10	78	24	42	12				
4	build13	24	15	15	10	72	20	25	27				

TABLE 13.19 Scale for PhAge

	Phase Discovered							
Phase Injected	Requirements	High-Level Design	Detailed Design	Coding	Unit Testing	Integration Testing	System Testing	Acceptance Testing
Requiremants	0	1	2	4	6	8	10	14
High-level design		0	1	2	4	6	8	12
Detailed design			0	1	2	4	6	10
Coding				0	1	2	4	8

11. Modify the schema shown in Table 13.2, including values of the fields to capture in the defect dynamic model.
12. Use the defect age (PhAge) given in Table 13.19 to recalculate the spoilage metric for the Boomerang test project.