FUTURE VISION BIE

One Stop for All Study Materials

& Lab Programs

Future Vision

By K B Hemanth Raj

Scan the QR Code to Visit the Web Page

Or

Visit : https://hemanthrajhemu.github.io

Gain Access to All Study Materials according to VTU,
CSE – Computer Science Engineering,
ISE – Information Science Engineering,
ECE - Electronics and Communication Engineering
& MORE...

Join Telegram to get Instant Updates: https://bit.ly/VTU_TELEGRAM

Contact: MAIL: futurevisionbie@gmail.com

INSTAGRAM: www.instagram.com/hemanthraj_hemu/

INSTAGRAM: www.instagram.com/futurevisionbie/

WHATSAPP SHARE: https://bit.ly/FVBIESHARE

Software Testing and Analysis: Process, Principles, and Techniques

Contents

List of Figures xi

List of Tables xv

I Fundamentals of Test and Analysis 1

1 Software Test and Analysis in a Nutshell 3
1.1 Engineering Processes and Verification . . . 3
1.2 Basic Questions . . . 5
1.3 When Do Verification and Validation Start and End? . . . 5
1.4 What Techniques Should Be Applied? . . . 7
1.5 How Can We Assess the Readiness of a Product? . . . 10
1.6 How Can We Ensure the Quality of Successive Releases? . . . 11
1.7 How Can the Development Process Be Improved? . . . 11

2 A Framework for Test and Analysis 15
2.1 Validation and Verification . . . 15
2.2 Degrees of Freedom . . . 18
2.3 Varieties of Software . . . 23

3 Basic Principles 29
3.1 Sensitivity . . . 29
3.2 Redundancy . . . 32
3.3 Restriction . . . 33
3.4 Partition . . . 35
3.5 Visibility . . . 36
3.6 Feedback . . . 36

4 Test and Analysis Activities Within a Software Process 39
4.1 The Quality Process . . . 39
4.2 Planning and Monitoring . . . 41
4.3 Quality Goals . . . 42
4.4 Dependability Properties . . . 43
4.5 Analysis . . . 46

4.6 Testing . . . 48
4.7 Improving the Process . . . 49
4.8 Organizational Factors . . . 50

II Basic Techniques 53

5 Finite Models 55
5.1 Overview . . . 55
5.2 Finite Abstractions of Behavior . . . 58
5.3 Control Flow Graphs . . . 59
5.4 Call Graphs . . . 63
5.5 Finite State Machines . . . 65

6 Dependence and Data Flow Models 77
6.1 Definition-Use Pairs . . . 77
6.2 Data Flow Analysis . . . 82
6.3 Classic Analyses: Live and Avail . . . 85
6.4 From Execution to Conservative Flow Analysis . . . 91
6.5 Data Flow Analysis with Arrays and Pointers . . . 94
6.6 Interprocedural Analysis . . . 96

7 Symbolic Execution and Proof of Properties 101
7.1 Symbolic State and Interpretation . . . 102
7.2 Summary Information . . . 104
7.3 Loops and Assertions . . . 105
7.4 Compositional Reasoning . . . 108
7.5 Reasoning about Data Structures and Classes . . . 109

8 Finite State Verification 113
8.1 Overview . . . 113
8.2 State Space Exploration . . . 116
8.3 The State Space Explosion Problem . . . 126
8.4 The Model Correspondence Problem . . . 129
8.5 Granularity of Modeling . . . 131
8.6 Intensional Models . . . 134
8.7 Model Refinement . . . 138
8.8 Data Model Verification with Relational Algebra . . . 140

III Problems and Methods 149

9 Test Case Selection and Adequacy 151
9.1 Overview . . . 151
9.2 Test Specifications and Cases . . . 152
9.3 Adequacy Criteria . . . 154
9.4 Comparing Criteria . . . 157

10 Functional Testing **161**
10.1 Overview . . . 161
10.2 Random versus Partition Testing Strategies . . . 162
10.3 A Systematic Approach . . . 167
10.4 Choosing a Suitable Approach . . . 174

11 Combinatorial Testing **179**
11.1 Overview . . . 180
11.2 Category-Partition Testing . . . 180
11.3 Pairwise Combination Testing . . . 188
11.4 Catalog-Based Testing . . . 194

12 Structural Testing **211**
12.1 Overview . . . 212
12.2 Statement Testing . . . 215
12.3 Branch Testing . . . 217
12.4 Condition Testing . . . 219
12.5 Path Testing . . . 222
12.6 Procedure Call Testing . . . 229
12.7 Comparing Structural Testing Criteria . . . 230
12.8 The Infeasibility Problem . . . 230

13 Data Flow Testing **235**
13.1 Overview . . . 236
13.2 Definition-Use Associations . . . 236
13.3 Data Flow Testing Criteria . . . 239
13.4 Data Flow Coverage with Complex Structures . . . 241
13.5 The Infeasibility Problem . . . 243

14 Model-Based Testing **245**
14.1 Overview . . . 245
14.2 Deriving Test Cases from Finite State Machines . . . 246
14.3 Testing Decision Structures . . . 251
14.4 Deriving Test Cases from Control and Data Flow Graphs . . . 257
14.5 Deriving Test Cases from Grammars . . . 257

15 Testing Object-Oriented Software **271**
15.1 Overview . . . 271
15.2 Issues in Testing Object-Oriented Software . . . 272
15.3 An Orthogonal Approach to Test . . . 280
15.4 Intraclass Testing . . . 282
15.5 Testing with State Machine Models . . . 282
15.6 Interclass Testing . . . 286
15.7 Structural Testing of Classes . . . 293
15.8 Oracles for Classes . . . 298
15.9 Polymorphism and Dynamic Binding . . . 301

15.10 Inheritance . . . 303
15.11 Genericity . . . 306
15.12 Exceptions . . . 308

16 Fault-Based Testing 313
16.1 Overview . . . 313
16.2 Assumptions in Fault-Based Testing . . . 314
16.3 Mutation Analysis . . . 315
16.4 Fault-Based Adequacy Criteria . . . 319
16.5 Variations on Mutation Analysis . . . 321

17 Test Execution 327
17.1 Overview . . . 327
17.2 From Test Case Specifications to Test Cases . . . 328
17.3 Scaffolding . . . 329
17.4 Generic versus Specific Scaffolding . . . 330
17.5 Test Oracles . . . 332
17.6 Self-Checks as Oracles . . . 334
17.7 Capture and Replay . . . 337

18 Inspection 341
18.1 Overview . . . 341
18.2 The Inspection Team . . . 343
18.3 The Inspection Process . . . 344
18.4 Checklists . . . 345
18.5 Pair Programming . . . 351

19 Program Analysis 355
19.1 Overview . . . 355
19.2 Symbolic Execution in Program Analysis . . . 356
19.3 Symbolic Testing . . . 358
19.4 Summarizing Execution Paths . . . 359
19.5 Memory Analysis . . . 360
19.6 Lockset Analysis . . . 363
19.7 Extracting Behavior Models from Execution . . . 365

IV Process 373

20 Planning and Monitoring the Process 375
20.1 Overview . . . 375
20.2 Quality and Process . . . 376
20.3 Test and Analysis Strategies . . . 377
20.4 Test and Analysis Plans . . . 382
20.5 Risk Planning . . . 386
20.6 Monitoring the Process . . . 389

20.7 Improving the Process . . . 394
20.8 The Quality Team . . . 399

21 Integration and Component-based Software Testing 405
21.1 Overview . . . 405
21.2 Integration Testing Strategies . . . 408
21.3 Testing Components and Assemblies . . . 413

22 System, Acceptance, and Regression Testing 417
22.1 Overview . . . 417
22.2 System Testing . . . 418
22.3 Acceptance Testing . . . 421
22.4 Usability . . . 423
22.5 Regression Testing . . . 427
22.6 Regression Test Selection Techniques . . . 428
22.7 Test Case Prioritization and Selective Execution . . . 434

23 Automating Analysis and Test 439
23.1 Overview . . . 439
23.2 Automation and Planning . . . 441
23.3 Process Management . . . 441
23.4 Static Metrics . . . 443
23.5 Test Case Generation and Execution . . . 445
23.6 Static Analysis and Proof . . . 445
23.7 Cognitive Aids . . . 448
23.8 Version Control . . . 449
23.9 Debugging . . . 449
23.10 Choosing and Integrating Tools . . . 451

24 Documenting Analysis and Test 455
24.1 Overview . . . 455
24.2 Organizing Documents . . . 456
24.3 Test Strategy Document . . . 458
24.4 Analysis and Test Plan . . . 458
24.5 Test Design Specification Documents . . . 460
24.6 Test and Analysis Reports . . . 462

Bibliography 467

Index 479

Chapter 3

Basic Principles

Mature engineering disciplines are characterized by basic principles. Principles provide a rationale for defining, selecting, and applying techniques and methods. They are valid beyond a single technique and over a time span in which techniques come and go, and can help engineers study, define, evaluate, and apply new techniques.

Analysis and testing (A&T) has been common practice since the earliest software projects. A&T activities were for a long time based on common sense and individual skills. It has emerged as a distinct discipline only in the last three decades.

This chapter advocates six principles that characterize various approaches and techniques for analysis and testing: sensitivity, redundancy, restriction, partition, visibility, and feedback. Some of these principles, such as partition, visibility, and feedback, are quite general in engineering. Others, notably sensitivity, redundancy, and restriction, are specific to A&T and contribute to characterizing A&T as a discipline.

3.1 Sensitivity

Human developers make errors, producing faults in software. Faults may lead to failures, but faulty software may not fail on every execution. The sensitivity principle states that it is better to fail every time than sometimes.

Consider the cost of detecting and repairing a software fault. If it is detected immediately (e.g., by an on-the-fly syntactic check in a design editor), then the cost of correction is very small, and in fact the line between fault prevention and fault detection is blurred. If a fault is detected in inspection or unit testing, the cost is still relatively small. If a fault survives initial detection efforts at the unit level, but triggers a failure detected in integration testing, the cost of correction is much greater. If the first failure is detected in system or acceptance testing, the cost is very high indeed, and the most costly faults are those detected by customers in the field.

A fault that triggers a failure on every execution is unlikely to survive past unit testing. A characteristic of faults that escape detection until much later is that they trigger failures only rarely, or in combination with circumstances that seem unrelated or are difficult to control. For example, a fault that results in a failure only for some unusual configurations of customer equipment may be difficult and expensive to detect.

A fault that results in a failure randomly but very rarely — for example, a race condition that only occasionally causes data corruption — may likewise escape detection until the software is in use by thousands of customers, and even then be difficult to diagnose and correct.

The small C program in Figure 3.1 has three faulty calls to string copy procedures. The call to strcpy, strncpy, and stringCopy all pass a source string "Muddled," which is too long to fit in the array middle. The vulnerability of strcpy is well known, and is the culprit in the by-now-standard buffer overflow attacks on many network services. Unfortunately, the fault may or may not cause an observable failure depending on the arrangement of memory (in this case, it depends on what appears in the position that would be middle[7], which will be overwritten with a newline character). The standard recommendation is to use strncpy in place of strcpy. While strncpy avoids overwriting other memory, it truncates the input without warning, and sometimes without properly null-terminating the output. The replacement function stringCopy, on the other hand, uses an assertion to ensure that, if the target string is too long, the program always fails in an observable manner.

The sensitivity principle says that we should try to make these faults easier to detect by making them cause failure more often. It can be applied in three main ways: at the design level, changing the way in which the program fails; at the analysis and testing level, choosing a technique more reliable with respect to the property of interest; and at the environment level, choosing a technique that reduces the impact of external factors on the results.

Replacing strcpy and strncpy with stringCopy in the program of Figure 3.1 is a simple example of application of the sensitivity principle in design. Run-time array bounds checking in many programming languages (including Java but not C or C++) is an example of the sensitivity principle applied at the language level. A variety of tools and replacements for the standard memory management library are available to enhance sensitivity to memory allocation and reference faults in C and C++.

The fail-fast property of Java iterators is another application of the sensitivity principle. A Java iterator provides a way of accessing each item in a collection data structure. Without the fail-fast property, modifying the collection while iterating over it could lead to unexpected and arbitrary results, and failure might occur rarely and be hard to detect and diagnose. A fail-fast iterator has the property that an immediate and observable failure (throwing ConcurrentModificationException) occurs when the illegal modification occurs. Although fail-fast behavior is not guaranteed if the update occurs in a different thread, a fail-fast iterator is far more sensitive than an iterator without the fail-fast property.

So far, we have discussed the sensitivity principle applied to design and code: always privilege design and code solutions that lead to consistent behavior, that is, such that fault occurrence does not depend on uncontrolled execution conditions that may mask faults, thus resulting in random failures. The sensitivity principle can also be applied to test and analysis techniques. In this case, we privilege techniques that cause faults to consistently manifest in failures.

Deadlock and race conditions in concurrent systems may depend on the relative speed of execution of the different threads or processes, and a race condition may lead

```
1  /**
2  * Worse than broken: Are you feeling lucky?
3  */
4
5  #include <assert.h>
6
7     char before[ ] = "=Before=";
8     char middle[ ] = "Middle";
9     char after[ ] = "=After=";
10
11 void  show() {
12    printf("%s\n%s\n%s\n", before, middle, after);
13 }
14
15 void  stringCopy(char *target, const char *source, int howBig);
16
17 int  main(int argc, char *argv) {
18    show();
19    strcpy(middle, "Muddled");     /* Fault, but may not fail */
20    show();
21    strncpy(middle, "Muddled", sizeof(middle)); /* Fault, may not fail */
22    show();
23    stringCopy(middle, "Muddled",sizeof(middle)); /* Guaranteed to fail */
24    show();
25 }
26
27 /* Sensitive version of strncpy; can be counted on to fail
28 * in an observable way EVERY time the source is too large
29 * for the target, unlike the standard strncpy or strcpy.
30 */
31 void  stringCopy(char *target, const char *source, int howBig) {
32    assert(strlen(source) < howBig);
33    strcpy(target, source);
34 }
```

Figure 3.1: Standard C functions strcpy *and* strncpy *may or may not fail when the source string is too long. The procedure* stringCopy *is* sensitive: *It is guaranteed to fail in an observable way if the source string is too long.*

to an observable failure only under rare conditions. Testing a concurrent system on a single configuration may fail to reveal deadlocks and race conditions. Repeating the tests with different configurations and system loads may help, but it is difficult to predict or control the circumstances under which failure occurs. We may observe that testing is not sensitive enough for revealing deadlocks and race conditions, and we may substitute other techniques that are more sensitive and less dependent on factors outside the developers' and testers' control. Model checking and reachability analysis techniques are limited in the scope of the faults they can detect, but they are very sensitive to this particular class of faults, having the advantage that they attain complete independence from any particular execution environment by systematically exploring all possible interleavings of processes.

Test adequacy criteria identify partitions of the input domain of the unit under test that must be sampled by test suites. For example, the statement coverage criterion requires each statement to be exercised at least once, that is, it groups inputs according to the statements they execute. Reliable criteria require that inputs belonging to the same class produce the same test results: They all fail or they all succeed. When this happens, we can infer the correctness of a program with respect to the a whole class of inputs from a single execution. Unfortunately, general reliable criteria do not exist[1].

Code inspection can reveal many subtle faults. However, inspection teams may produce completely different results depending on the cohesion of the team, the discipline of the inspectors, and their knowledge of the application domain and the design technique. The use of detailed checklists and a disciplined review process may reduce the influence of external factors, such as teamwork attitude, inspectors' discipline, and domain knowledge, thus increasing the predictability of the results of inspection. In this case, sensitivity is applied to reduce the influence of external factors.

Similarly, skilled test designers can derive excellent test suites, but the quality of the test suites depends on the mood of the designers. Systematic testing criteria may not do better than skilled test designers, but they can reduce the influence of external factors, such as the tester's mood.

3.2 Redundancy

Redundancy is the opposite of independence. If one part of a software artifact (program, design document, etc.) constrains the content of another, then they are not entirely independent, and it is possible to check them for consistency.

The concept and definition of redundancy are taken from information theory. In communication, redundancy can be introduced into messages in the form of error-detecting and error-correcting codes to guard against transmission errors. In software test and analysis, we wish to detect faults that could lead to differences between intended behavior and actual behavior, so the most valuable form of redundancy is in the form of an explicit, redundant statement of intent.

Where redundancy can be introduced or exploited with an automatic, algorithmic check for consistency, it has the advantage of being much cheaper and more thorough

[1] Existence of a general, reliable test coverage criterion would allow us to prove the equivalence of programs. Readers interested in this topic will find more information in Chapter 9.

than dynamic testing or manual inspection. Static type checking is a classic application of this principle: The type declaration is a statement of intent that is at least partly redundant with the use of a variable in the source code. The type declaration constrains other parts of the code, so a consistency check (type check) can be applied.

An important trend in the evolution of programming languages is introduction of additional ways to declare intent and automatically check for consistency. For example, Java enforces rules about explicitly declaring each exception that can be thrown by a method.

Checkable redundancy is not limited to program source code, nor is it something that can be introduced only by programming language designers. For example, software design tools typically provide ways to check consistency between different design views or artifacts. One can also intentionally introduce redundancy in other software artifacts, even those that are not entirely formal. For example, one might introduce rules quite analogous to type declarations for semistructured requirements specification documents, and thereby enable automatic checks for consistency and some limited kinds of completeness.

When redundancy is already present — as between a software specification document and source code — then the remaining challenge is to make sure the information is represented in a way that facilitates cheap, thorough consistency checks. Checks that can be implemented by automatic tools are usually preferable, but there is value even in organizing information to make inconsistency easier to spot in manual inspection.

Of course, one cannot always obtain cheap, thorough checks of source code and other documents. Sometimes redundancy is exploited instead with run-time checks. Defensive programming, explicit run-time checks for conditions that should always be true if the program is executing correctly, is another application of redundancy in programming.

3.3 Restriction

When there are no acceptably cheap and effective ways to check a property, sometimes one can change the problem by checking a different, more restrictive property or by limiting the check to a smaller, more restrictive class of programs.

Consider the problem of ensuring that each variable is initialized before it is used, on every execution. Simple as the property is, it is not possible for a compiler or analysis tool to precisely determine whether it holds. See the program in Figure 3.2 for an illustration. Can the variable k ever be uninitialized the first time i is added to it? If someCondition(0) always returns true, then k will be initialized to zero on the first time through the loop, before k is incremented, so perhaps there is no potential for a run-time error — but method someCondition could be arbitrarily complex and might even depend on some condition in the environment. Java's solution to this problem is to enforce a stricter, simpler condition: A program is not permitted to have any syntactic control paths on which an uninitialized reference could occur, regardless of whether those paths could actually be executed. The program in Figure 3.2 has such a path, so the Java compiler rejects it.

Java's rule for initialization before use is a program source code restriction that

```
1       /** A trivial method with a potentially uninitialized variable.
2         * Maybe someCondition(0) is always true, and therefore k is
3         * always initialized before use ... but it's impossible, in
4         * general, to know for sure. Java rejects the method.
5         */
6       static void questionable() {
7           int k;
8           for (int i=0; i < 10; ++i) {
9               if (someCondition(i)) {
10                  k = 0;
11              } else {
12                  k += i;
13              }
14          }
15          System.out.println(k);
16      }
17  }
```

Figure 3.2: Can the variable k *ever be uninitialized the first time* i *is added to it? The property is undecidable, so Java enforces a simpler, stricter property.*

enables precise, efficient checking of a simple but important property by the compiler. The choice of programming language(s) for a project may entail a number of such restrictions that impact test and analysis. Additional restrictions may be imposed in the form of programming standards (e.g., restricting the use of type casts or pointer arithmetic in C), or by tools in a development environment. Other forms of restriction can apply to architectural and detailed design.

Consider, for example, the problem of ensuring that a transaction consisting of a sequence of accesses to a complex data structure by one process appears to the outside world as if it had occurred atomically, rather than interleaved with transactions of other processes. This property is called *serializability*: The end result of a set of such transactions should appear as if they were applied in some serial order, even if they didn't.

One way to ensure serializability is to make the transactions really serial (e.g., by putting the whole sequence of operations in each transaction within a Java synchronized block), but that approach may incur unacceptable performance penalties. One would like to allow interleaving of transactions that don't interfere, while still ensuring the appearance of atomic access, and one can devise a variety of locking and versioning techniques to achieve this. Unfortunately, checking directly to determine whether the serializability property has been achieved is very expensive at run-time, and precisely checking whether it holds on all possible executions is impossible. Fortunately, the problem becomes much easier if we impose a particular locking or versioning scheme on the program at design time. Then the problem becomes one of proving, on the one hand, that the particular concurrency control protocol has the desired property, and then

checking that the program obeys the protocol. Database researchers have completed the first step, and some of the published and well-known concurrency control protocols are trivial to check at run-time and simple enough that (with some modest additional restrictions) they can be checked even by source code analysis.

From the above examples it should be clear that the restriction principle is useful mainly during design and specification; it can seldom be applied post hoc on a complete software product. In other words, restriction is mainly a principle to be applied in design for test. Often it can be applied not only to solve a single problem (like detecting potential access of uninitialized variables, or nonserializable execution of transactions) but also at a more general, architectural level to simplify a whole set of analysis problems.

Stateless component interfaces are an example of restriction applied at the architectural level. An interface is stateless if each service request (method call, remote procedure call, message send and reply) is independent of all others; that is, the service does not "remember" anything about previous requests. Stateless interfaces are far easier to test because the correctness of each service request and response can be checked independently, rather than considering all their possible sequences or interleavings. A famous example of simplifying component interfaces by making them stateless is the Hypertext Transport Protocol (HTTP) 1.0 of the World-Wide-Web, which made Web servers not only much simpler and more robust but also much easier to test.

3.4 Partition

Partition, often also known as "divide and conquer," is a general engineering principle. Dividing a complex problem into subproblems to be attacked and solved independently is probably the most common human problem-solving strategy. Software engineering in particular applies this principle in many different forms and at almost all development levels, from early requirements specifications to code and maintenance. Analysis and testing are no exception: the partition principle is widely used and exploited.

Partitioning can be applied both at process and technique levels. At the process level, we divide complex activities into sets of simple activities that can be attacked independently. For example, testing is usually divided into unit, integration, subsystem, and system testing. In this way, we can focus on different sources of faults at different steps, and at each step, we can take advantage of the results of the former steps. For instance, we can use units that have been tested as stubs for integration testing. Some static analysis techniques likewise follow the modular structure of the software system to divide an analysis problem into smaller steps.

Many static analysis techniques first construct a model of a system and then analyze the model. In this way they divide the overall analysis into two subtasks: first simplify the system to make the proof of the desired properties feasible and then prove the property with respect to the simplified model. The question "Does this program have the desired property?" is decomposed into two questions, "Does this model have the desired property?" and "Is this an accurate model of the program?"

Since it is not possible to execute the program with every conceivable input, systematic testing strategies must identify a finite number of classes of test cases to exe-

cute. Whether the classes are derived from specifications (functional testing) or from program structure (structural testing), the process of enumerating test obligations proceeds by dividing the sources of information into significant elements (clauses or special values identifiable in specifications, statements or paths in programs), and creating test cases that cover each such element or certain combinations of elements.

3.5 Visibility

Visibility means the ability to measure progress or status against goals. In software engineering, one encounters the visibility principle mainly in the form of process visibility, and then mainly in the form of schedule visibility: ability to judge the state of development against a project schedule. Quality process visibility also applies to measuring achieved (or predicted) quality against quality goals. The principle of visibility involves setting goals that can be assessed as well as devising methods to assess their realization.

Visibility is closely related to observability, the ability to extract useful information from a software artifact. The architectural design and build plan of a system determines what will be observable at each stage of development, which in turn largely determines the visibility of progress against goals at that stage.

A variety of simple techniques can be used to improve observability. For example, it is no accident that important Internet protocols like HTTP and SMTP (Simple Mail Transport Protocol, used by Internet mail servers) are based on the exchange of simple textual commands. The choice of simple, human-readable text rather than a more compact binary encoding has a small cost in performance and a large payoff in observability, including making construction of test drivers and oracles much simpler. Use of human-readable and human-editable files is likewise advisable wherever the performance cost is acceptable.

A variant of observability through direct use of simple text encodings is providing readers and writers to convert between other data structures and simple, human-readable and editable text. For example, when designing classes that implement a complex data structure, designing and implementing also a translation from a simple text format to the internal structure, and vice versa, will often pay back handsomely in both ad hoc and systematic testing. For similar reasons it is often useful to design and implement an equality check for objects, even when it is not necessary to the functionality of the software product.

3.6 Feedback

Feedback is another classic engineering principle that applies to analysis and testing. Feedback applies both to the process itself (process improvement) and to individual techniques (e.g., using test histories to prioritize regression testing).

Systematic inspection and walkthrough derive part of their success from feedback. Participants in inspection are guided by checklists, and checklists are revised and refined based on experience. New checklist items may be derived from root cause anal-

ysis, analyzing previously observed failures to identify the initial errors that lead to them.

Summary

Principles constitute the core of a discipline. They form the basis of methods, techniques, methodologies and tools. They permit understanding, comparing, evaluating and extending different approaches, and they constitute the lasting basis of knowledge of a discipline.

The six principles described in this chapter are

- Sensitivity: better to fail every time than sometimes,
- Redundancy: making intentions explicit,
- Restriction: making the problem easier,
- Partition: divide and conquer,
- Visibility: making information accessible, and
- Feedback: applying lessons from experience in process and techniques.

Principles are identified heuristically by searching for a common denominator of techniques that apply to various problems and exploit different methods, sometimes borrowing ideas from other disciplines, sometimes observing recurrent phenomena. Potential principles are validated by finding existing and new techniques that exploit the underlying ideas. Generality and usefulness of principles become evident only with time. The initial list of principles proposed in this chapter is certainly incomplete. Readers are invited to validate the proposed principles and identify additional principles.

Further Reading

Analysis and testing is a relatively new discipline. To our knowledge, the principles underlying analysis and testing have not been discussed in the literature previously. Some of the principles advocated in this chapter are shared with other software engineering disciplines and are discussed in many books. A good introduction to software engineering principles is the third chapter of Ghezzi, Jazayeri, and Mandrioli's book on software engineering [GJM02].

Exercises

3.1. Indicate which principles guided the following choices:

1. Use an externally readable format also for internal files, when possible.
2. Collect and analyze data about faults revealed and removed from the code.
3. Separate test and debugging activities; that is, separate the design and execution of test cases to reveal failures (test) from the localization and removal of the corresponding faults (debugging).
4. Distinguish test case design from execution.
5. Produce complete fault reports.
6. Use information from test case design to improve requirements and design specifications.
7. Provide interfaces for fully inspecting the internal state of a class.

3.2. A simple mechanism for augmenting fault tolerance consists of replicating computation and comparing the obtained results. Can we consider redundancy for fault tolerance an application of the redundancy principle?

3.3. A system safety specification describes prohibited behaviors (what the system must never do). Explain how specified safety properties can be viewed as an implementation of the redundancy principle.

3.4. Process visibility can be increased by extracting information about the progress of the process. Indicate some information that can be easily produced to increase process visibility.

Chapter 4

Test and Analysis Activities Within a Software Process

Dependability and other qualities of software are not ingredients that can be added in a final step before delivery. Rather, software quality results from a whole set of interdependent activities, among which analysis and testing are necessary but far from sufficient. And while one often hears of a testing "phase" in software development, as if testing were a distinct activity that occurred at a particular point in development, one should not confuse this flurry of test execution with the whole process of software test and analysis any more than one would confuse program compilation with programming.

Testing and analysis activities occur throughout the development and evolution of software systems, from early in requirements engineering through delivery and subsequent evolution. Quality depends on every part of the software process, not only on software analysis and testing; no amount of testing and analysis can make up for poor quality arising from other activities. On the other hand, an essential feature of software processes that produce high-quality products is that software test and analysis is thoroughly integrated and not an afterthought.

4.1 The Quality Process

One can identify particular activities and responsibilities in a software development process that are focused primarily on ensuring adequate dependability of the software product, much as one can identify other activities and responsibilities concerned primarily with project schedule or with product usability. It is convenient to group these quality assurance activities under the rubric "quality process," although we must also recognize that quality is intertwined with and inseparable from other facets of the overall process. Like other parts of an overall software process, the quality process provides a framework for selecting and arranging activities aimed at a particular goal, while also considering interactions and trade-offs with other important goals. All software development activities reflect constraints and trade-offs, and quality activities are no

exception. For example, high dependability is usually in tension with time to market, and in most cases it is better to achieve a reasonably high degree of dependability on a tight schedule than to achieve ultra-high dependability on a much longer schedule, although the opposite is true in some domains (e.g., certain medical devices).

The quality process should be structured for completeness, timeliness, and cost-effectiveness. Completeness means that appropriate activities are planned to detect each important class of faults. What the important classes of faults are depends on the application domain, the organization, and the technologies employed (e.g., memory leaks are an important class of faults for C++ programs, but seldom for Java programs). Timeliness means that faults are detected at a point of high leverage, which in practice almost always means that they are detected as early as possible. Cost-effectiveness means that, subject to the constraints of completeness and timeliness, one chooses activities depending on their cost as well as their effectiveness. Cost must be considered over the whole development cycle and product life, so the dominant factor is likely to be the cost of repeating an activity through many change cycles.

Activities that one would typically consider as being in the domain of quality assurance or quality improvement, that is, activities whose primary goal is to prevent or detect faults, intertwine and interact with other activities carried out by members of a software development team. For example, architectural design of a software system has an enormous impact on the test and analysis approaches that will be feasible and on their cost. A precise, relatively formal architectural model may form the basis for several static analyses of the model itself and of the consistency between the model and its implementation, while another architecture may be inadequate for static analysis and, if insufficiently precise, of little help even in forming an integration test plan.

The intertwining and mutual impact of quality activities on other development activities suggests that it would be foolish to put off quality activities until late in a project. The effects run not only from other development activities to quality activities but also in the other direction. For example, early test planning during requirements engineering typically clarifies and improves requirements specifications. Developing a test plan during architectural design may suggest structures and interfaces that not only facilitate testing earlier in development, but also make key interfaces simpler and more precisely defined.

There is also another reason for carrying out quality activities at the earliest opportunity and for preferring earlier to later activities when either could serve to detect the same fault: The single best predictor of the cost of repairing a software defect is the time between its introduction and its detection. A defect introduced in coding is far cheaper to repair during unit test than later during integration or system test, and most expensive if it is detected by a user of the fielded system. A defect introduced during requirements engineering (e.g., an ambiguous requirement) is relatively cheap to repair at that stage, but may be hugely expensive if it is only uncovered by a dispute about the results of a system acceptance test.

4.2 Planning and Monitoring

process visibility

Process visibility is a key factor in software process in general, and software quality processes in particular. A process is visible to the extent that one can answer the question, "How does our progress compare to our plan?" Typically, schedule visibility is a main emphasis in process design ("Are we on schedule? How far ahead or behind?"), but in software quality process an equal emphasis is needed on progress against quality goals. If one cannot gain confidence in the quality of the software system long before it reaches final testing, the quality process has not achieved adequate visibility.

A well-designed quality process balances several activities across the whole development process, selecting and arranging them to be as cost-effective as possible, and to improve early visibility. Visibility is particularly challenging and is one reason (among several) that quality activities are usually placed as early in a software process as possible. For example, one designs test cases at the earliest opportunity (not "just in time") and uses both automated and manual static analysis techniques on software artifacts that are produced before actual code.

Early visibility also motivates the use of "proxy" measures, that is, use of quantifiable attributes that are not identical to the properties that one really wishes to measure, but that have the advantage of being measurable earlier in development. For example, we know that the number of faults in design or code is not a true measure of reliability. Nonetheless, one may count faults uncovered in design inspections as an early indicator of potential quality problems, because the alternative of waiting to receive a more accurate estimate from reliability testing is unacceptable.

Quality goals can be achieved only through careful planning of activities that are matched to the identified objectives. Planning is integral to the quality process and is elaborated and revised through the whole project. It encompasses both an overall strategy for test and analysis, and more detailed test plans.

A&T strategy

The overall analysis and test strategy identifies company- or project-wide standards that must be satisfied: procedures for obtaining quality certificates required for certain classes of products, techniques and tools that must be used, and documents that must be produced. Some companies develop and certify procedures following international standards such as ISO 9000 or SEI Capability Maturity Model, which require detailed documentation and management of analysis and test activities and well-defined phases, documents, techniques, and tools. A&T strategies are described in detail in Chapter 20, and a sample strategy document for the Chipmunk Web presence is given in Chapter 24.

A&T plan

The initial build plan for Chipmunk Web-based purchasing functionality includes an analysis and test plan. A complete analysis and test plan is a comprehensive description of the quality process and includes several items: It indicates objectives and scope of the test and analysis activities; it describes documents and other items that must be available for performing the planned activities, integrating the quality process with the software development process; it identifies items to be tested, thus allowing for simple completeness checks and detailed planning; it distinguishes features to be tested from those not to be tested; it selects analysis and test activities that are considered essential for success of the quality process; and finally it identifies the staff involved in analysis and testing and their respective and mutual responsibilities.

The final analysis and test plan includes additional information that illustrates constraints, pass and fail criteria, schedule, deliverables, hardware and software requirements, risks, and contingencies. Constraints indicate deadlines and limits that may be derived from the hardware and software implementation of the system under analysis and the tools available for analysis and testing. Pass and fail criteria indicate when a test or analysis activity succeeds or fails, thus supporting monitoring of the quality process. The schedule describes the individual tasks to be performed and provides a feasible schedule. Deliverables specify which documents, scaffolding and test cases must be produced, and indicate the quality expected from such deliverables. Hardware, environment and tool requirements indicate the support needed to perform the scheduled activities. The risk and contingency plan identifies the possible problems and provides recovery actions to avoid major failures. The test plan is discussed in more detail in Chapter 20.

4.3 Quality Goals

Process visibility requires a clear specification of goals, and in the case of quality process visibility this includes a careful distinction among dependability qualities. A team that does not have a clear idea of the difference between reliability and robustness, for example, or of their relative importance in a project, has little chance of attaining either. Goals must be further refined into a clear and reasonable set of objectives. If an organization claims that nothing less than 100% reliability will suffice, it is not setting an ambitious objective. Rather, it is setting no objective at all, and choosing not to make reasoned trade-off decisions or to balance limited resources across various activities. It is, in effect, abrogating responsibility for effective quality planning, and leaving trade-offs among cost, schedule, and quality to an arbitrary, ad hoc decision based on deadline and budget alone.

The relative importance of qualities and their relation to other project objectives varies. Time-to-market may be the most important property for a mass market product, usability may be more prominent for a Web based application, and safety may be the overriding requirement for a life-critical system.

Product qualities are the goals of software quality engineering, and process qualities are means to achieve those goals. For example, development processes with a high degree of visibility are necessary for creation of highly dependable products. The process goals with which software quality engineering is directly concerned are often on the "cost" side of the ledger. For example, we might have to weigh stringent reliability objectives against their impact on time-to-market, or seek ways to improve time-to-market without adversely impacting robustness.

internal and external qualities

Software product qualities can be divided into those that are directly visible to a client and those that primarily affect the software development organization. Reliability, for example, is directly visible to the client. Maintainability primarily affects the development organization, although its consequences may indirectly affect the client as well, for example, by increasing the time between product releases. Properties that are directly visible to users of a software product, such as dependability, latency, us-

ability, and throughput, are called *external* properties. Properties that are not directly visible to end users, such as maintainability, reusability, and traceability, are called *internal* properties, even when their impact on the software development and evolution processes may indirectly affect users.

dependability

The external properties of software can ultimately be divided into dependability (does the software do what it is intended to do?) and usefulness. There is no precise way to distinguish these, but a rule of thumb is that when software is not dependable, we say it has a fault, or a defect, or (most often) a bug, resulting in an undesirable behavior or failure.

usefulness

It is quite possible to build systems that are very reliable, relatively free from hazards, and completely useless. They may be unbearably slow, or have terrible user interfaces and unfathomable documentation, or they may be missing several crucial features. How should these properties be considered in software quality? One answer is that they are not part of quality at all unless they have been explicitly specified, since quality is the presence of specified properties. However, a company whose products are rejected by its customers will take little comfort in knowing that, by some definitions, they were high-quality products.

We can do better by considering quality as fulfillment of required and desired properties, as distinguished from specified properties. For example, even if a client does not explicitly specify the required performance of a system, there is always *some* level of performance that is required to be useful.

One of the most critical tasks in software quality analysis is making desired properties explicit, since properties that remain unspecified (even informally) are very likely to surface unpleasantly when it is discovered that they are not met. In many cases these implicit requirements can not only be made explicit, but also made sufficiently precise that they can be made part of dependability or reliability. For example, while it is better to explicitly recognize usability as a requirement than to leave it implicit, it is better yet to augment[1] usability requirements with specific interface standards, so that a deviation from the standards is recognized as a defect.

4.4 Dependability Properties

correctness

The simplest of the dependability properties is correctness: A program or system is correct if it is consistent with its specification. By definition, a specification divides all possible system behaviors[2] into two classes, *successes* (or correct executions) and *failures*. All of the possible behaviors of a correct system are successes.

A program cannot be mostly correct or somewhat correct or 30% correct. It is absolutely correct on all possible behaviors, or else it is not correct. It is very easy to achieve correctness, since every program is correct with respect to some (very bad)

[1] Interface standards augment, rather than replace, usability requirements because conformance to the standards is not sufficient assurance that the requirement is met. This is the same relation that other specifications have to the user requirements they are intended to fulfill. In general, verifying conformance to specifications does not replace validating satisfaction of requirements.

[2] We are simplifying matters somewhat by considering only specifications of behaviors. A specification may also deal with other properties, such as the disk space required to install the application. A system may thus also be "incorrect" if it violates one of these static properties.

specification. Achieving correctness with respect to a useful specification, on the other hand, is seldom practical for nontrivial systems. Therefore, while correctness may be a noble goal, we are often interested in assessing some more achievable level of dependability.

reliability

Reliability is a statistical approximation to correctness, in the sense that 100% reliability is indistinguishable from correctness. Roughly speaking, reliability is a measure of the likelihood of correct function for some "unit" of behavior, which could be a single use or program execution or a period of time. Like correctness, reliability is relative to a specification (which determines whether a unit of behavior is counted as a success or failure). Unlike correctness, reliability is also relative to a particular usage profile. The same program can be more or less reliable depending on how it is used.

availability

Particular measures of reliability can be used for different units of execution and different ways of counting success and failure. *Availability* is an appropriate measure when a failure has some duration in time. For example, a failure of a network router may make it impossible to use some functions of a local area network until the service is restored; between initial failure and restoration we say the router is "down" or "unavailable." The availability of the router is the time in which the system is "up" (providing normal service) as a fraction of total time. Thus, a network router that averages 1 hour of down time in each 24-hour period would have an availability of $\frac{23}{24}$, or 95.8%.

MTBF

Mean time between failures (MTBF) is yet another measure of reliability, also using time as the unit of execution. The hypothetical network switch that typically fails once in a 24-hour period and takes about an hour to recover has a mean time between failures of 23 hours. Note that availability does not distinguish between two failures of 30 minutes each and one failure lasting an hour, while MTBF does.

The definitions of correctness and reliability have (at least) two major weaknesses. First, since the success or failure of an execution is relative to a specification, they are only as strong as the specification. Second, they make no distinction between a failure that is a minor annoyance and a failure that results in catastrophe. These are simplifying assumptions that we accept for the sake of precision, but in some circumstances — particularly, but not only, for critical systems — it is important to consider dependability properties that are less dependent on specification and that do distinguish among failures depending on severity.

safety

Software safety is an extension of the well-established field of system safety into software. Safety is concerned with preventing certain undesirable behaviors, called

hazards

hazards. It is quite explicitly not concerned with achieving any useful behavior apart from whatever functionality is needed to prevent hazards. Software safety is typically a concern in "critical" systems such as avionics and medical systems, but the basic principles apply to any system in which particularly undesirable behaviors can be distinguished from run-of-the-mill failure. For example, while it is annoying when a word processor crashes, it is much more annoying if it irrecoverably corrupts document files. The developers of a word processor might consider safety with respect to the hazard of file corruption separately from reliability with respect to the complete functional requirements for the word processor.

Just as correctness is meaningless without a specification of allowed behaviors,

safety is meaningless without a specification of hazards to be prevented, and in practice the first step of safety analysis is always finding and classifying hazards. Typically, hazards are associated with some system in which the software is embedded (e.g., the medical device), rather than the software alone. The distinguishing feature of safety is that it is concerned *only* with these hazards, and not with other aspects of correct functioning.

The concept of safety is perhaps easier to grasp with familiar physical systems. For example, lawn-mowers in the United States are equipped with an interlock device, sometimes called a "dead-man switch." If this switch is not actively held by the operator, the engine shuts off. The dead-man switch does not contribute in any way to cutting grass; its sole purpose is to prevent the operator from reaching into the mower blades while the engine runs.

One is tempted to say that safety is an aspect of correctness, because a good system specification would rule out hazards. However, safety is best considered as a quality distinct from correctness and reliability for two reasons. First, by focusing on a few hazards and ignoring other functionality, a separate safety specification can be much simpler than a complete system specification, and therefore easier to verify. To put it another way, while a good system specification *should* rule out hazards, we cannot be confident that either specifications or our attempts to verify systems are good enough to provide the degree of assurance we require for hazard avoidance. Second, even if the safety specification were redundant with regard to the full system specification, it is important because (by definition) we regard avoidance of hazards as more crucial than satisfying other parts of the system specification.

Correctness and reliability are contingent upon normal operating conditions. It is not reasonable to expect a word processing program to save changes normally when the file does not fit in storage, or to expect a database to continue to operate normally when the computer loses power, or to expect a Web site to provide completely satisfactory service to all visitors when the load is 100 times greater than the maximum for which it was designed. Software that fails under these conditions, which violate the premises of its design, may still be "correct" in the strict sense, yet the manner in which the software fails is important. It is acceptable that the word processor fails to write the new file that does not fit on disk, but unacceptable to also corrupt the previous version of the file in the attempt. It is acceptable for the database system to cease to function when the power is cut, but unacceptable for it to leave the database in a corrupt state. And it is usually preferable for the Web system to turn away some arriving users rather than becoming too slow for all, or crashing. Software that gracefully degrades or fails "softly" outside its normal operating parameters is *robust*.

robustness

Software safety is a kind of robustness, but robustness is a more general notion that concerns not only avoidance of hazards (e.g., data corruption) but also partial functionality under unusual situations. Robustness, like safety, begins with explicit consideration of unusual and undesirable situations, and should include augmenting software specifications with appropriate responses to undesirable events.

Figure 4.1 illustrates the relation among dependability properties.

Quality analysis should be part of the feasibility study. The sidebar on page 47

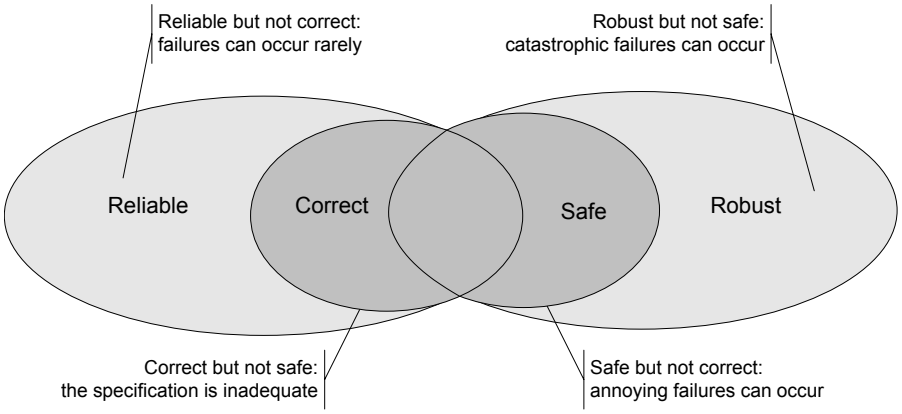


Figure 4.1: Relation among dependability properties

shows an excerpt of the feasibility study for the Chipmunk Web presence. The primary quality requirements are stated in terms of dependability, usability, and security. Performance, portability and interoperability are typically not primary concerns at this stage, but they may come into play when dealing with other qualities.

4.5 Analysis

Analysis techniques that do not involve actual execution of program source code play a prominent role in overall software quality processes. Manual inspection techniques and automated analyses can be applied at any development stage. They are particularly well suited at the early stages of specifications and design, where the lack of executability of many intermediate artifacts reduces the efficacy of testing.

Inspection, in particular, can be applied to essentially any document including requirements documents, architectural and more detailed design documents, test plans and test cases, and of course program source code. Inspection may also have secondary benefits, such as spreading good practices and instilling shared standards of quality. On the other hand, inspection takes a considerable amount of time and requires meetings, which can become a scheduling bottleneck. Moreover, re-inspecting a changed component can be as expensive as the initial inspection. Despite the versatility of inspection, therefore, it is used primarily where other techniques are either inapplicable or where other techniques do not provide sufficient coverage of common faults.

Automated static analyses are more limited in applicability (e.g., they can be applied to some formal representations of requirements models but not to natural language documents), but are selected when available because substituting machine cycles for human effort makes them particularly cost-effective. The cost advantage of automated static analyses is diminished by the substantial effort required to formalize and properly structure a model for analysis, but their application can be further mo-

Excerpt of Web Presence Feasibility Study

Purpose of this document

This document was prepared for the Chipmunk IT management team. It describes the results of a feasibility study undertaken to advise Chipmunk corporate management whether to embark on a substantial redevelopment effort to add online shopping functionality to the Chipmunk Computers' Web presence.

Goals

The primary goal of a Web presence redevelopment is to add online shopping facilities. Marketing estimates an increase of 15% over current direct sales within 24 months, and an additional 8% savings in direct sales support costs from shifting telephone price inquiries to online price inquiries. [. . .]

Architectural Requirements

The logical architecture will be divided into three distinct subsystems: human interface, business logic, and supporting infrastructure. Each major subsystem must be structured for phased development, with initial features delivered 6 months from inception, full features at 12 months, and a planned revision at 18 months from project inception. [. . .]

Quality Requirements

Dependability: With the introduction of direct sales and customer relationship management functions, dependability of Chipmunk's Web services becomes business-critical. A critical core of functionality will be identified, isolated from less critical functionality in design and implementation, and subjected to the highest level of scrutiny. We estimate that this will be approximately 20% of new development and revisions, and that the V&V costs for those portions will be approximately triple the cost of V&V for noncritical development.

Usability: The new Web presence will be, to a much greater extent than before, the public face of Chipmunk Computers. [. . .]

Security: Introduction of online direct ordering and billing raises a number of security issues. Some of these can be avoided initially by contracting with one of several service companies that provide secure credit card transaction services. Nonetheless, order tracking, customer relationship management, returns, and a number of other functions that cannot be effectively outsourced raise significant security and privacy issues. Identifying and isolating security concerns will add a significant but manageable cost to design validation. [. . .]

tivated by their ability to thoroughly check for particular classes of faults for which checking with other techniques is very difficult or expensive. For example, finite state verification techniques for concurrent systems requires construction and careful structuring of a formal design model, and addresses only a particular family of faults (faulty synchronization structure). Yet it is rapidly gaining acceptance in some application domains because that family of faults is difficult to detect in manual inspection and resists detection through dynamic testing.

Sometimes the best aspects of manual inspection and automated static analysis can be obtained by carefully decomposing properties to be checked. For example, suppose a desired property of requirements documents is that each special term in the application domain appear in a glossary of terms. This property is not directly amenable to an automated static analysis, since current tools cannot distinguish meaningful domain terms from other terms that have their ordinary meanings. The property can be checked with manual inspection, but the process is tedious, expensive, and error-prone. A hybrid approach can be applied if each domain term is marked in the text. Manually checking that domain terms are marked is much faster and therefore less expensive than manually looking each term up in the glossary, and marking the terms permits effective automation of cross-checking with the glossary.

4.6 Testing

Despite the attractiveness of automated static analyses when they are applicable, and despite the usefulness of manual inspections for a variety of documents including but not limited to program source code, dynamic testing remains a dominant technique. A closer look, though, shows that dynamic testing is really divided into several distinct activities that may occur at different points in a project.

Tests are executed when the corresponding code is available, but testing activities start earlier, as soon as the artifacts required for designing test case specifications are available. Thus, acceptance and system test suites should be generated before integration and unit test suites, even if executed in the opposite order.

Early test design has several advantages. Tests are specified independently from code and when the corresponding software specifications are fresh in the mind of analysts and developers, facilitating review of test design. Moreover, test cases may highlight inconsistencies and incompleteness in the corresponding software specifications. Early design of test cases also allows for early repair of software specifications, preventing specification faults from propagating to later stages in development. Finally, programmers may use test cases to illustrate and clarify the software specifications, especially for errors and unexpected conditions.

No engineer would build a complex structure from parts that have not themselves been subjected to quality control. Just as the “earlier is better” rule dictates using inspection to reveal flaws in requirements and design before they are propagated to program code, the same rule dictates module testing to uncover as many program faults as possible before they are incorporated in larger subsystems of the product. At Chipmunk, developers are expected to perform functional and structural module testing before a work assignment is considered complete and added to the project baseline. The

test driver and auxiliary files are part of the work product and are expected to make re-execution of test cases, including result checking, as simple and automatic as possible, since the same test cases will be used over and over again as the product evolves.

4.7 Improving the Process

While the assembly-line, mass production industrial model is inappropriate for software, which is at least partly custom-built, there is almost always some commonality among projects undertaken by an organization over time. Confronted by similar problems, developers tend to make the same kinds of errors over and over, and consequently the same kinds of software faults are often encountered project after project. The quality process, as well as the software development process as a whole, can be improved by gathering, analyzing, and acting on data regarding faults and failures.

The goal of quality process improvement is to find cost-effective countermeasures for classes of faults that are expensive because they occur frequently, or because the failures they cause are expensive, or because, once detected, they are expensive to repair. Countermeasures may be either prevention or detection and may involve either quality assurance activities (e.g., improved checklists for design inspections) or other aspects of software development (e.g., improved requirements specification methods).

The first part of a process improvement feedback loop, and often the most difficult to implement, is gathering sufficiently complete and accurate raw data about faults and failures. A main obstacle is that data gathered in one project goes mainly to benefit other projects in the future and may seem to have little direct benefit for the current project, much less to the persons asked to provide the raw data. It is therefore helpful to integrate data collection as well as possible with other, normal development activities, such as version and configuration control, project management, and bug tracking. It is also essential to minimize extra effort. For example, if revision logs in the revision control database can be associated with bug tracking records, then the time between checking out a module and checking it back in might be taken as a rough guide to cost of repair.

Raw data on faults and failures must be aggregated into categories and prioritized. Faults may be categorized along several dimensions, none of them perfect. Fortunately, a flawless categorization is not necessary; all that is needed is some categorization scheme that is sufficiently fine-grained and tends to aggregate faults with similar causes and possible remedies, and that can be associated with at least rough estimates of relative frequency and cost. A small number of categories — maybe just one or two — are chosen for further study.

The analysis step consists of tracing several instances of an observed fault or failure back to the human error from which it resulted, or even further to the factors that led to that human error. The analysis also involves the reasons the fault was not detected and eliminated earlier (e.g., how it slipped through various inspections and levels of testing). This process is known as "root cause analysis," but the ultimate aim is for the most cost-effective countermeasure, which is sometimes but not always the ultimate root cause. For example, the persistence of security vulnerabilities through buffer overflow errors in network applications may be attributed at least partly to widespread

root cause analysis

use of programming languages with unconstrained pointers and without array bounds checking, which may in turn be attributed to performance concerns and a requirement for interoperability with a large body of legacy code. The countermeasure could involve differences in programming methods (e.g., requiring use of certified "safe" libraries for buffer management), or improvements to quality assurance activities (e.g., additions to inspection checklists), or sometimes changes in management practices.

4.8 Organizational Factors

The quality process includes a wide variety of activities that require specific skills and attitudes and may be performed by quality specialists or by software developers. Planning the quality process involves not only resource management but also identification and allocation of responsibilities.

A poor allocation of responsibilities can lead to major problems in which pursuit of individual goals conflicts with overall project success. For example, splitting responsibilities of development and quality-control between a development and a quality team, and rewarding high productivity in terms of lines of code per person-month during development may produce undesired results. The development team, not rewarded to produce high-quality software, may attempt to maximize productivity to the detriment of quality. The resources initially planned for quality assurance may not suffice if the initial quality of code from the"very productive" development team is low. On the other hand, combining development and quality control responsibilities in one undifferentiated team, while avoiding the perverse incentive of divided responsibilities, can also have unintended effects: As deadlines near, resources may be shifted from quality assurance to coding, at the expense of product quality.

Conflicting considerations support both the separation of roles (e.g., recruiting quality specialists), and the mobility of people and roles (e.g, rotating engineers between development and testing tasks).

At Chipmunk, responsibility for delivery of the new Web presence is distributed among a development team and a quality assurance team. Both teams are further articulated into groups. The quality assurance team is divided into the analysis and testing group, responsible for the dependability of the system, and the usability testing group, responsible for usability. Responsibility for security issues is assigned to the infrastructure development group, which relies partly on external consultants for final tests based on external attack attempts.

Having distinct teams does not imply a simple division of all tasks between teams by category. At Chipmunk, for example, specifications, design, and code are inspected by mixed teams; scaffolding and oracles are designed by analysts and developers; integration, system, acceptance, and regression tests are assigned to the test and analysis team; unit tests are generated and executed by the developers; and coverage is checked by the testing team before starting integration and system testing. A specialist has been hired for analyzing faults and improving the process. The process improvement specialist works incrementally while developing the system and proposes improvements at each release.

Summary

Test and analysis activities are not a late phase of the development process, but rather a wide set of activities that pervade the whole process. Designing a quality process with a suitable blend of test and analysis activities for the specific application domain, development environment, and quality goals is a challenge that requires skill and experience.

A well-defined quality process must fulfill three main goals: improving the software product during and after development, assessing its quality before delivery, and improving the process within and across projects. These challenging goals can be achieved by increasing visibility, scheduling activities as early as practical, and monitoring results to adjust the process. Process visibility — that is, measuring and comparing progress to objectives — is a key property of the overall development process. Performing A&T activities early produces several benefits: It increases control over the process, it hastens fault identification and reduces the costs of fault removal, it provides data for incrementally tuning the development process, and it accelerates product delivery. Feedback is the key to improving the process by identifying and removing persistent errors and faults.

Further Reading

Qualities of software are discussed in many software engineering textbooks; the discussion in Chapter 2 of Ghezzi, Jazayeri, and Mandrioli [GJM02] is particularly useful. Process visibility is likewise described in software engineering textbooks, usually with an emphasis on schedule. Musa [Mus04] describes a quality process oriented particularly to establishing a quantifiable level of reliability based on models and testing before release. Chillarege et al. [CBC+92] present principles for gathering and analyzing fault data, with an emphasis on feedback within a single process but applicable also to quality process improvement.

Exercises

4.1. We have stated that 100% reliability is indistinguishable from correctness, but they are not quite identical. Under what circumstance might an incorrect program be 100% reliable? *Hint:* Recall that a program may be more or less reliable depending on how it is used, but a program is either correct or incorrect regardless of usage.

4.2. We might measure the reliability of a network router as the fraction of all packets that are correctly routed, or as the fraction of total service time in which packets are correctly routed. When might these two measures be different?

4.3. If I am downloading a very large file over a slow modem, do I care more about the availability of my internet service provider or its mean time between failures?

4.4. Can a system be correct and yet unsafe?

4.5. Under what circumstances can making a system more safe make it less reliable?

4.6. Software application domains can be characterized by the relative importance of schedule (calendar time), total cost, and dependability. For example, while all three are important for game software, schedule (shipping product in September to be available for holiday purchases) has particular weight, while dependability can be somewhat relaxed. Characterize a domain you are familiar with in these terms.

4.7. Consider responsiveness as a desirable property of an Internet chat program. The informal requirement is that messages typed by each member of a chat session appear instantaneously on the displays of other users. Refine this informal requirement into a concrete specification that can be verified. Is anything lost in the refinement?

4.8. Identify some correctness, robustness and safety properties of a word processor.

Chapter 20

Planning and Monitoring the Process

Any complex process requires planning and monitoring. The quality process requires coordination of many different activities over a period that spans a full development cycle and beyond. Planning is necessary to order, provision, and coordinate all the activities that support a quality goal, and monitoring of actual status against a plan is required to steer and adjust the process.

Required Background

- Chapter 4
  Introduction of basic concepts of quality process, goals, and activities provides useful background for understanding this chapter.

20.1 Overview

Planning involves scheduling activities, allocating resources, and devising observable, unambiguous milestones against which progress and performance can be monitored. Monitoring means answering the question, "How are we doing?"

Quality planning is one aspect of project planning, and quality processes must be closely coordinated with other development processes. Coordination among quality and development tasks may constrain ordering (e.g., unit tests are executed after creation of program units). It may shape tasks to facilitate coordination; for example, delivery may be broken into smaller increments to allow early testing. Some aspects of the project plan, such as feedback and design for testability, may belong equally to the quality plan and other aspects of the project plan.

Quality planning begins at the inception of a project and is developed with the overall project plan, instantiating and building on a quality strategy that spans several projects. Like the overall project plan, the quality plan is developed incrementally, beginning with the feasibility study and continuing through development and delivery.

Formulation of the plan involves risk analysis and contingency planning. Execution of the plan involves monitoring, corrective action, and planning for subsequent releases and projects.

Allocating responsibility among team members is a crucial and difficult part of planning. When one person plays multiple roles, explicitly identifying each responsibility is still essential for ensuring that none are neglected.

20.2 Quality and Process

A software plan involves many intertwined concerns, from schedule to cost to usability and dependability. Despite the intertwining, it is useful to distinguish individual concerns and objectives to lessen the likelihood that they will be neglected, to allocate responsibilities, and to make the overall planning process more manageable.

For example, a mature software project plan will include architectural design reviews, and the quality plan will allocate effort for reviewing testability aspects of the structure and build order. Clearly, design for testability is an aspect of software design and cannot be carried out by a separate testing team in isolation. It involves both test designers and other software designers in explicitly evaluating testability as one consideration in selecting among design alternatives. The objective of incorporating design for testability in the quality process is primarily to ensure that it is not overlooked and secondarily to plan activities that address it as effectively as possible.

An appropriate quality process follows a form similar to the overall software process in which it is embedded. In a strict (and unrealistic) waterfall software process, one would follow the "V model" (Figure 2.1 on page 16) in a sequential manner, beginning unit testing only as implementation commenced following completion of the detailed design phase, and finishing unit testing before integration testing commenced. In the XP "test first" method, unit testing is conflated with subsystem and system testing. A cycle of test design and test execution is wrapped around each small-grain incremental development step. The role that inspection and peer reviews would play in other processes is filled in XP largely by pair programming. A typical spiral process model lies somewhere between, with distinct planning, design, and implementation steps in several increments coupled with a similar unfolding of analysis and test activities. Some processes specifically designed around quality activities are briefly outlined in the sidebars on pages 378, 380, and 381.

A general principle, across all software processes, is that the cost of detecting and repairing a fault increases as a function of time between committing an error and detecting the resultant faults. Thus, whatever the intermediate work products in a software plan, an efficient quality plan will include a matched set of intermediate validation and verification activities that detect most faults within a short period of their introduction. Any step in a software process that is not paired with a validation or verification step is an opportunity for defects to fester, and any milestone in a project plan that does not include a quality check is an opportunity for a misleading assessment of progress.

The particular verification or validation step at each stage depends on the nature of the intermediate work product and on the anticipated defects. For example, anticipated defects in a requirements statement might include incompleteness, ambiguity,

inconsistency, and overambition relative to project goals and resources. A review step might address some of these, and automated analyses might help with completeness and consistency checking.

The evolving collection of work products can be viewed as a set of descriptions of different parts and aspects of the software system, at different levels of detail. Portions of the implementation have the useful property of being executable in a conventional sense, and are the traditional subject of testing, but every level of specification and design can be both the subject of verification activities and a source of information for verifying other artifacts. A typical intermediate artifact — say, a subsystem interface definition or a database schema — will be subject to the following steps:

Internal consistency check: Check the artifact for compliance with structuring rules that define "well-formed" artifacts of that type. An important point of leverage is defining the syntactic and semantic rules thoroughly and precisely enough that many common errors result in detectable violations. This is analogous to syntax and strong-typing rules in programming languages, which are not enough to guarantee program correctness but effectively guard against many simple errors.

External consistency check: Check the artifact for consistency with related artifacts. Often this means checking for conformance to a "prior" or "higher-level" specification, but consistency checking does not depend on sequential, top-down development — all that is required is that the related information from two or more artifacts be defined precisely enough to support detection of discrepancies. Consistency usually proceeds from broad, syntactic checks to more detailed and expensive semantic checks, and a variety of automated and manual verification techniques may be applied.

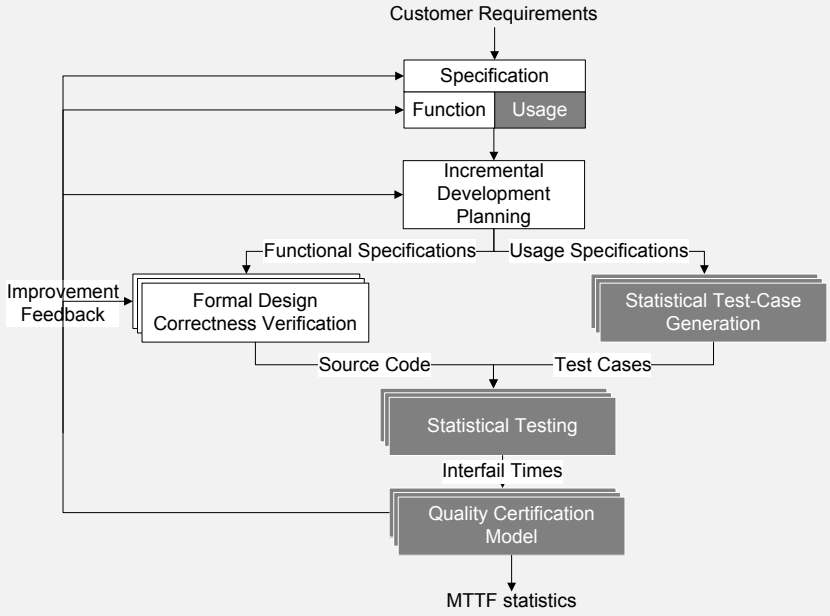
Generation of correctness conjectures: Correctness conjectures, which can be test outcomes or other objective criteria, lay the groundwork for external consistency checks of other work products, particularly those that are yet to be developed or revised. Generating correctness conjectures for other work products will frequently motivate refinement of the current product. For example, an interface definition may be elaborated and made more precise so that implementations can be effectively tested.

20.3 Test and Analysis Strategies

Lessons of past experience are an important asset of organizations that rely heavily on technical skills. A body of explicit knowledge, shared and refined by the group, is more valuable than islands of individual competence. Organizational knowledge in a shared and systematic form is more amenable to improvement and less vulnerable to organizational change, including the loss of key individuals. Capturing the lessons of experience in a consistent and repeatable form is essential for avoiding errors, maintaining consistency of the process, and increasing development efficiency.

Cleanroom

The Cleanroom process model, introduced by IBM in the late 1980s, pairs development with V&V activities and stresses analysis over testing in the early phases. Testing is left for system certification. The Cleanroom process involves two cooperating teams, the development and the quality teams, and five major activities: specification, planning, design and verification, quality certification, and feedback.



In the *specification* activity, the development team defines the required behavior of the system, while the quality team defines usage scenarios that are later used for deriving system test suites. The *planning* activity identifies incremental development and certification phases.

After planning, all activities are iterated to produce incremental releases of the system. Each system increment is fully deployed and certified before the following step. Design and code undergo formal inspection ("*Correctness verification*") before release. One of the key premises underpinning the Cleanroom process model is that rigorous design and formal inspection produce "nearly fault-free software."

Usage profiles generated during specification are applied in the *statistical testing* activity to gauge quality of each release. Another key assumption of the Cleanroom process model is that usage profiles are sufficiently accurate that statistical testing will provide an accurate measure of quality as perceived by users.[a] Reliability is measured in terms of mean time between failures (MTBF) and is constantly controlled after each release. Failures are reported to the development team for correction, and if reliability falls below an acceptable range, failure data is used for process improvement before the next incremental release.

[a] See Chapter 22 for more detail on statistical testing and usage profiling.

Software organizations can develop useful, organization-specific strategies because of similarities among projects carried out by a particular organization in a set of related application domains. Test and analysis strategies capture commonalities across projects and provide guidelines for maintaining consistency among quality plans.

A strategy is distinguished from a plan in that it is not specific to a single project. Rather, it provides guidance and a general framework for developing quality plans for several projects, satisfying organizational quality standards, promoting homogeneity across projects, and making both the creation and execution of individual project quality plans more efficient.

The quality strategy is an intellectual asset of an individual organization prescribing a set of solutions to problems specific to that organization. Among the factors that particularize the strategy are:

Structure and size: Large organizations typically have sharper distinctions between development and quality groups, even if testing personnel are assigned to development teams. In smaller organizations, it is more common for a single person to serve multiple roles. Where responsibility is distributed among more individuals, the quality strategy will require more elaborate attention to coordination and communication, and in general there will be much greater reliance on documents to carry the collective memory.

In a smaller organization, or an organization that has devolved responsibility to small, semi-autonomous teams, there is typically less emphasis on formal communication and documents but a greater emphasis on managing and balancing the multiple roles played by each team member.

Overall process: We have already noted the intertwining of quality process with other aspects of an overall software process, and this is of course reflected in the quality strategy. For example, if an organization follows the Cleanroom methodology, then inspections will be required but unit testing forbidden. An organization that adopts the XP methodology is likely to follow the "test first" and pair programming elements of that approach, and in fact would find a more document-heavy approach a difficult fit.

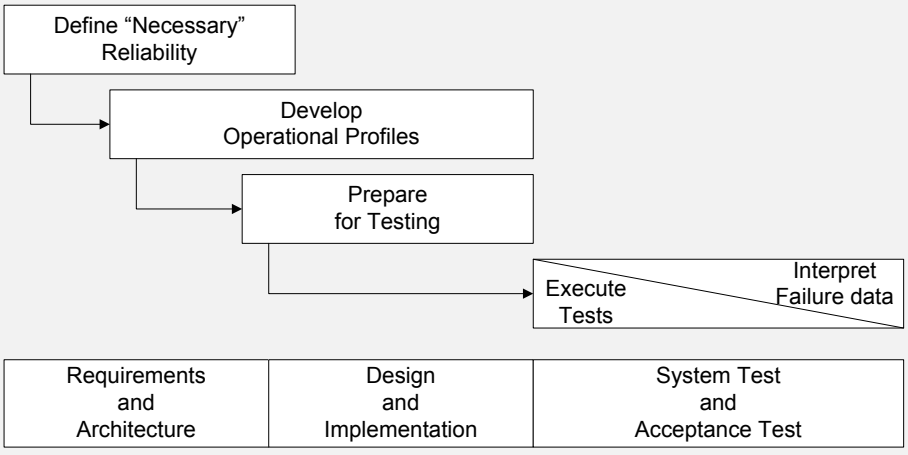
Notations, standard process steps, and even tools can be reflected in the quality strategy to the extent they are consistent from project to project. For example, if an organization consistently uses a particular combination of UML diagram notations to document subsystem interfaces, then the quality strategy might include derivation of test designs from those notations, as well as review and analysis steps tailored to detect the most common and important design flaws at that point. If a particular version and configuration control system is woven into process management, the quality strategy will likewise exploit it to support and enforce quality process steps.

Application domain: The domain may impose both particular quality objectives (e.g., privacy and security in medical records processing), and in some cases particular steps and documentation required to obtain certification from an external authority. For example, the RTCA/DO-178B standard for avionics software requires testing to the modified condition/decision coverage (MC/DC) criterion.

SRET

The software reliability engineered testing (SRET) approach, developed at AT&T in the early 1990s, assumes a spiral development process and augments each coil of the spiral with rigorous testing activities. SRET identifies two main types of testing: *development testing*, used to find and remove faults in software at least partially developed in-house, and *certification testing*, used to either accept or reject outsourced software.

The SRET approach includes seven main steps. Two initial, quick decision-making steps determine which systems require separate testing and which type of testing is needed for each system to be tested. The five core steps are executed in parallel with each coil of a spiral development process.



The five core steps of SRET are:

Define "Necessary" Reliability: Determine operational models, that is, distinct patterns of system usage that require separate testing, classify failures according to their severity, and engineer the reliability strategy with fault prevention, fault removal, and fault tolerance activities.

Develop Operational Profiles: Develop both overall profiles that span operational models and operational profiles within single operational models.

Prepare for Testing: Specify test cases and procedures.

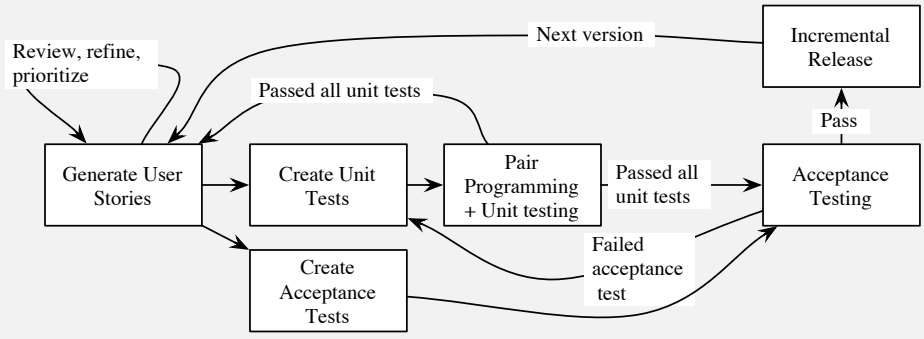
Execute Tests

Interpret Failure Data: Interpretation of failure data depends on the type of testing. In development testing, the goal is to track progress and compare present failure intensities with objectives. In certification testing, the goal is to determine if a software component or system should be accepted or rejected.

Extreme Programming (XP)

The extreme programming methodology (XP) emphasizes simplicity over generality, global vision and communication over structured organization, frequent changes over big releases, continuous testing and analysis over separation of roles and responsibilities, and continuous feedback over traditional planning.

Customer involvement in an XP project includes requirements analysis (development, refinement, and prioritization of *user stories*) and acceptance testing of very frequent iterative releases. Planning is based on prioritization of user stories, which are implemented in short iterations. Test cases corresponding to scenarios in user stories serve as partial specifications.



Test cases suitable for batch execution are part of the system code base and are implemented prior to the implementation of features they check ("test-first"). Developers work in pairs, incrementally developing and testing a module. Pair programming effectively conflates a review activity with coding. Each release is checked by running all the tests devised up to that point of development, thus essentially merging unit testing with integration and system testing. A failed acceptance test is viewed as an indication that additional unit tests are needed.

Although there are no standard templates for analysis and test strategies, we can identify a few elements that should be part of almost any good strategy. A strategy should specify common quality requirements that apply to all or most products, promoting conventions for unambiguously stating and measuring them, and reducing the likelihood that they will be overlooked in the quality plan for a particular project. A strategy should indicate a set of documents that is normally produced during the quality process, and their contents and relationships. It should indicate the activities that are prescribed by the overall process organization. Often a set of standard tools and practices will be prescribed, such as the interplay of a version and configuration control tool with review and testing procedures. In addition, a strategy includes guidelines for project staffing and assignment of roles and responsibilities. An excerpt of a sample strategy document is presented in Chapter 24.

20.4 Test and Analysis Plans

An analysis and test plan details the steps to be taken in a particular project. A plan should answer the following questions:

- *What quality activities will be carried out?*
- *What are the dependencies among the quality activities and between quality and development activities?*
- *What resources are needed and how will they be allocated?*
- *How will both the process and the evolving product be monitored to maintain an adequate assessment of quality and early warning of quality and schedule problems?*

Each of these issues is addressed to some extent in the quality strategy, but must be elaborated and particularized. This is typically the responsibility of a quality manager, who should participate in the initial feasibility study to identify quality goals and estimate the contribution of test and analysis tasks on project cost and schedule.

To produce a quality plan that adequately addresses the questions above, the quality manager must identify the items and features to be verified, the resources and activities that are required, the approaches that should be followed, and criteria for evaluating the results.

Items and features to be verified circumscribe the target of the quality plan. While there is an obvious correspondence between items to be developed or modified and those to undergo testing, they may differ somewhat in detail. For example, overall evaluation of the user interface may be the purview of a separate human factors group. The items to be verified, moreover, include many intermediate artifacts such as requirements specifications and design documents, in addition to portions of the delivered system. Approaches to be taken in verification and validation may vary among items. For example, the plan may prescribe inspection and testing for all items and additional static analyses for multi-threaded subsystems.

Quality goals must be expressed in terms of properties satisfied by the product and must be further elaborated with metrics that can be monitored during the course of the project. For example, if known failure scenarios are classified as critical, severe, moderate, and minor, then we might decide in advance that a product version may enter end-user acceptance testing only when it has undergone system testing with no outstanding critical or severe failures.

Defining quality objectives and process organization in detail requires information that is not all available in the early stages of development. Test items depend on design decisions; detailed approaches to evaluation can be defined only after examining requirements and design specifications; tasks and schedule can be completed only after the design; new risks and contingencies may be introduced by decisions taken during development. On the other hand, an early plan is necessary for estimating and controlling cost and schedule. The quality manager must start with an initial plan based on incomplete and tentative information, and incrementally refine the plan as more and better information becomes available during the project.

After capturing goals as well as possible, the next step in construction of a quality plan is to produce an overall rough list of tasks. The quality strategy and past experience provide a basis for customizing the list to the current project and for scaling tasks appropriately. For example, experience (preferably in the form of collected and analyzed data from past projects, rather than personal memory) might suggest a ratio of 3:5 for person-months of effort devoted to integration test relative to coding effort. Historical data may also provide scaling factors for the application domain, interfaces with externally developed software, and experience of the quality staff. To the extent possible, the quality manager must break large tasks into component subtasks to obtain better estimates, but it is inevitable that some task breakdown must await further elaboration of the overall project design and schedule.

The manager can start noting dependencies among the quality activities and between them and other activities in the overall project, and exploring arrangements of tasks over time. The main objective at this point is to schedule quality activities so that assessment data are provided continuously throughout the project, without unnecessary delay of other development activities. For example, the quality manager may note that the design and implementation of different subsystems are scheduled in different phases, and may plan subsystem testing accordingly.

Where there is a choice between scheduling a quality activity earlier or later, the earliest point possible is always preferable. However, the demand on resources (staff time, primarily) must be leveled over time, and often one must carefully schedule the availability of particular critical resources, such as an individual test designer with expertise in a particular technology. Maintaining a consistent level of effort limits the number of activities that can be carried on concurrently, and resource constraints together with the objective of minimizing project delays tends to force particular orderings on tasks.

If one has a choice between completing two tasks in four months, or completing the first task in two months and then the second in another two months, the schedule that brings one task to completion earlier is generally advantageous from the perspective of process visibility, as well as reduced coordination overhead. However, many activities demand a fraction of a person's attention over a longer period and cannot be compressed. For example, participation in design and code inspection requires a substantial investment of effort, but typically falls short of a full-time assignment. Since delayed inspections can be a bottleneck in progress of a project, they should have a high priority when they can be carried out, and are best interleaved with tasks that can be more flexibly scheduled.

While the project plan shows the expected schedule of tasks, the arrangement and ordering of tasks are also driven by risk. The quality plan, like the overall project plan, should include an explicit risk plan that lists major risks and contingencies, as discussed in the next section.

A key tactic for controlling the impact of risk in the project schedule is to minimize the likelihood that unexpected delay in one task propagates through the whole schedule and delays project completion. One first identifies the *critical paths* through the project schedule. Critical paths are chains of activities that must be completed in sequence and that have maximum overall duration. Tasks on the critical path have a high priority for early scheduling, and likewise the tasks on which they depend (which may not

critical paths

themselves be on the critical path) should be scheduled early enough to provide some schedule slack and prevent delay in the inception of the critical tasks.

A *critical dependence* occurs when a task on a critical path is scheduled immediately after some other task on the critical path, particularly if the length of the critical path is close to the length of the project. Critical dependence may occur with tasks outside the quality plan part of the overall project plan.

critical dependence

The primary tactic available for reducing the schedule risk of a critical dependence is to decompose a task on the critical path, factoring out subtasks that can be performed earlier. For example, an acceptance test phase late in a project is likely to have a critical dependence on development and system integration. One cannot entirely remove this dependence, but its potential to delay project completion is reduced by factoring test design from test execution.

Figure 20.1 shows alternative schedules for a simple project that starts at the beginning of January and must be completed by the end of May. In the top schedule, indicated as *CRITICAL SCHEDULE*, the tasks *Analysis and design*, *Code and Integration*, *Design and execute subsystem tests*, and *Design and execute system tests* form a critical path that spans the duration of the entire project. A delay in any of the activities will result in late delivery. In this schedule, only the *Produce user documentation* task does not belong to the critical path, and thus only delays of this task can be tolerated.

In the middle schedule, marked as *UNLIMITED RESOURCES*, the test design and execution activities are separated into distinct tasks. Test design tasks are scheduled early, right after *analysis and design*, and only test execution is scheduled after *Code and integration*. In this way the tasks *Design subsystem tests* and *Design system tests* are removed from the critical path, which now spans 16 weeks with a tolerance of 5 weeks with respect to the expected termination of the project. This schedule assumes enough resources for running *Code and integration*, *Production of user documentation*, *Design of subsystem tests*, and *Design of system tests*.

The *LIMITED RESOURCES* schedule at the bottom of Figure 20.1 rearranges tasks to meet resource constraints. In this case we assume that test design and execution, and production of user documentation share the same resources and thus cannot be executed in parallel. We can see that, despite the limited parallelism, decomposing testing activities and scheduling test design earlier results in a critical path of 17 weeks, 4 weeks earlier than the expected termination of the project. Notice that in the example, the critical path is formed by the tasks *Analysis and design*, *Design subsystem tests*, *Design system tests*, *Produce user documentation*, *Execute subsystem tests*, and *Execute system tests*. In fact, the limited availability of resources results in dependencies among *Design subsystem tests*, *Design system tests* and *Produce user documentation* that last longer than the parallel task *Code and integration*.

The completed plan must include frequent milestones for assessing progress. A rule of thumb is that, for projects of a year or more, milestones for assessing progress should occur at least every three months. For shorter projects, a reasonable maximum interval for assessment is one quarter of project duration.

Figure 20.2 shows a possible schedule for the initial analysis and test plan for the business logic of the Chipmunk Web presence in the form of a GANTT diagram. In the initial plan, the manager has allocated time and effort to inspections of all major artifacts, as well as test design as early as practical and ongoing test execution dur-

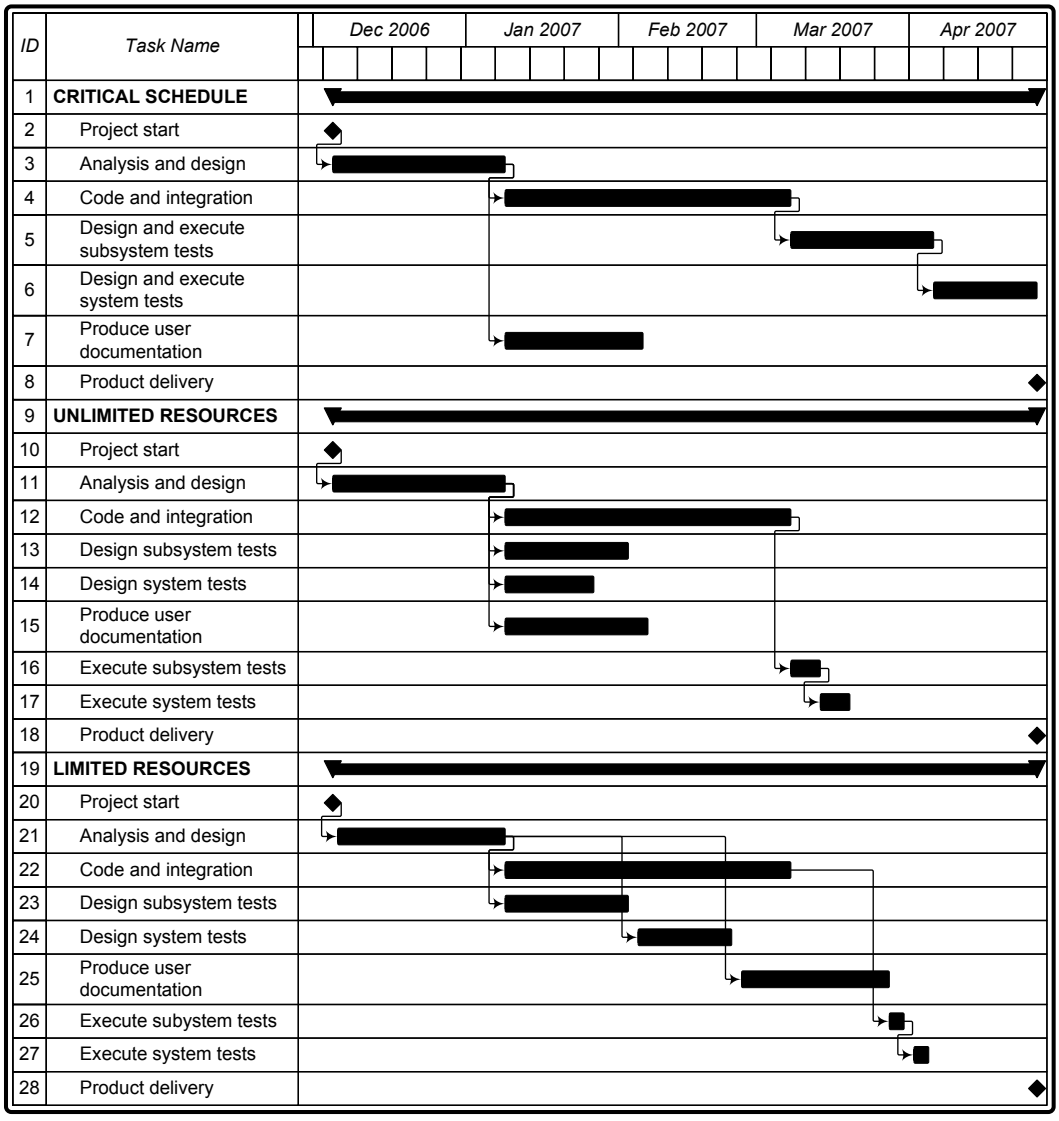


Figure 20.1: Three possible simple schedules with different risks and resource allocation. The bars indicate the duration of the tasks. Diamonds indicate milestones, and arrows between bars indicate precedence between tasks.

ing development. Division of the project into major parts is reflected in the plan, but further elaboration of tasks associated with units and smaller subsystems must await corresponding elaboration of the architectural design. Thus, for example, inspection of the shopping facilities code and the unit test suites is shown as a single aggregate task. Even this initial plan does reflect the usual Chipmunk development strategy of regular "synch and stabilize" periods punctuating development, and the initial quality plan reflects the Chipmunk strategy of assigning responsibility for producing unit test suites to developers, with review by a member of the quality team.

The GANTT diagram shows four main groups of analysis and test activities: *design inspection*, *code inspection*, *test design*, and *test execution*. The distribution of activities over time is constrained by resources and dependence among activities. For example, *system test execution* starts after completion of *system test design* and cannot finish before system integration (the *sync and stablize* elements of *development framework*) is complete. Inspection activities are constrained by specification and design activities. Test design activities are constrained by limited resources. Late scheduling of the design of integration tests for the administrative business logic subsystem is necessary to avoid overlap with design of tests for the shopping functionality subsystem.

The GANTT diagram does not highlight intermediate milestones, but we can easily identify two in April and July, thus dividing the development into three main phases. The first phase (January to April) corresponds to requirements analysis and architectural design activities and terminates with the architectural design baseline. In this phase, the quality team focuses on design inspection and on the design of acceptance and system tests. The second phase (May to July) corresponds to subsystem design and to the implementation of the first complete version of the system. It terminates with the first stabilization of the administrative business logic subsystem. In this phase, the quality team completes the design inspection and the design of test cases. In the final stage, the development team produces the final version, while the quality team focuses on code inspection and test execution.

Absence of test design activities in the last phase results from careful identification of activities that allowed early planning of critical tasks.

20.5 Risk Planning

Risk is an inevitable part of every project, and so risk planning must be a part of every plan. Risks cannot be eliminated, but they can be assessed, controlled, and monitored.

The risk plan component of the quality plan is concerned primarily with personnel risks, technology risks, and schedule risk. Personnel risk is any contingency that may make a qualified staff member unavailable when needed. For example, the reassignment of a key test designer cannot always be avoided, but the possible consequences can be analyzed in advance and minimized by careful organization of the work. Technology risks in the quality plan include risks of technology used specifically by the quality team and risks of quality problems involving other technology used in the product or project. For example, changes in the target platform or in the testing environment, due to new releases of the operating system or to the adoption of a new testing tool suite, may not be schedulable in advance, but may be taken into account in the

Figure 20.2: Initial schedule for quality activities in development of the business logic subsystem of the Chipmunk Web presence, presented as a GANTT diagram.

organization of the testing environment. Schedule risk arises primarily from optimistic assumptions in the quality plan. For example, underestimating scaffolding design and maintenance is a common mistake that cannot always be avoided, but consequences can be mitigated (e.g., by allowing for a reasonable slack time that can absorb possible delays). Many risks and the tactics for controlling them are generic to project management (e.g., cross-training to reduce the impact of losing a key staff member). Here we focus on risks that are specific to quality planning or for which risk control measures play a special role in the quality plan.

The duration of integration, system, and acceptance test execution depends to a large extent on the quality of software under test. Software that is sloppily constructed or that undergoes inadequate analysis and test before commitment to the code base will slow testing progress. Even if responsibility for diagnosing test failures lies with developers and not with the testing group, a test execution session that results in many failures and generates many failure reports is inherently more time consuming than executing a suite of tests with few or no failures. This schedule vulnerability is yet another reason to emphasize earlier activities, in particular those that provide early indications of quality problems. Inspection of design and code (with quality team participation) can help control this risk, and also serves to communicate quality standards and best practices among the team.

If unit testing is the responsibility of developers, test suites are part of the unit deliverable and should undergo inspection for correctness, thoroughness, and automation. While functional and structural coverage criteria are no panacea for measuring test thoroughness, it is reasonable to require that deviations from basic coverage criteria be justified on a case-by-case basis. A substantial deviation from the structural coverage observed in similar products may be due to many causes, including inadequate testing, incomplete specifications, unusual design, or implementation decisions. The modules that present unusually low structural coverage should be inspected to identify the cause.

The cost of analysis and test is multiplied when some requirements demand a very high level of assurance. For example, if a system that has previously been used in biological research is modified or redeveloped for clinical use, one should anticipate that all development costs, and particularly costs of analysis and test, will be an order of magnitude higher. In addition to the risk of underestimating the cost and schedule impact of stringent quality requirements, the risk of failing to achieve the required dependability increases. One important tactic for controlling this risk is isolating critical properties as far as possible in small, simple components. Of course these aspects of system specification and architectural design are not entirely within control of the quality team; it is crucial that at least the quality manager, and possibly other members of the quality team, participate in specification and design activities to assess and communicate the impact of design alternatives on cost and schedule.

Architectural design is also the primary point of leverage to control cost and risks of testing systems with complex external interfaces. For example, the hardware platform on which an embedded system must be tested may be a scarce resource, in demand for debugging as well as testing. Preparing and executing a test case on that platform may be time-consuming, magnifying the risk that system and operational testing may go over schedule and delay software delivery. This risk may be reduced by careful consideration of design-for-testability in architectural design. A testable design isolates and

minimizes platform dependencies, reducing the portion of testing that requires access to the platform. It will typically provide additional interfaces to enhance controllability and observability in testing. A considerable investment in test scaffolding, from self-diagnosis to platform simulators, may also be warranted.

Risks related both to critical requirements and limitations on testability can be partially addressed in system specifications and programming standards. For example, it is notoriously difficult to detect race conditions by testing multi-threaded software. However, one may impose a design and programming discipline that prevents race conditions, such as a simple monitor discipline with resource ordering. Detecting violations of that discipline, statically and dynamically, is much simpler than detecting actual data races. This tactic may be reflected in several places in the project plan, from settling on the programming discipline in architectural design to checking for proper use of the discipline in code and design inspections, to implementation or purchase of tools to automate compliance checking.

The sidebars on page 390 and 391 summarize a set of risks both generic to process management and specific to quality control that a quality manager must consider when defining a quality plan.

20.6 Monitoring the Process

The quality manager monitors progress of quality activities, including results as well as schedule, to identify deviations from the quality plan as early as possible and take corrective action. Effective monitoring, naturally, depends on a plan that is realistic, well organized, and sufficiently detailed with clear, unambiguous milestones and criteria. We say a process is *visible* to the extent that it can be effectively monitored.

Successful completion of a planned activity must be distinguished from mere termination, as otherwise it is too tempting to meet an impending deadline by omitting some planned work. Skipping planned verification activities or addressing them superficially can seem to accelerate a late project, but the boost is only apparent; the real effect is to postpone detection of more faults to later stages in development, where their detection and removal will be far more threatening to project success.

For example, suppose a developer is expected to deliver unit test cases as part of a work unit. If project deadlines are slipping, the developer is tempted to scrimp on designing unit tests and writing supporting code, perhaps dashing off a few superficial test cases so that the unit can be committed to the code base. The rushed development and inadequate unit testing are nearly guaranteed to leave bugs that surface later, perhaps in integration or system testing, where they will have a far greater impact on project schedule. Worst of all, they might be first detected in operational use, reducing the real and perceived quality of the delivered product. In monitoring progress, therefore, it is essential to include appropriate metrics of the thoroughness or completeness of the activity.

Monitoring produces a surfeit of detail about individual activities. Managers need to make decisions based on an overall understanding of project status, so raw monitoring information must be aggregated in ways that provide an overall picture.

Risk Management in the Quality Plan: Risks Generic to Process Management

The quality plan must identify potential risks and define appropriate control tactics. Some risks and control tactics are generic to process management, while others are specific to the quality process. Here we provide a brief overview of some risks generic to process management. Risks specific to the quality process are summarized in the sidebar on page 391.

Personnel Risks	**Example Control Tactics**
A staff member is lost (becomes ill, changes employer, etc.) or is underqualified for task (the project plan assumed a level of skill or familiarity that the assigned member did not have).	Cross train to avoid overdependence on individuals; encourage and schedule continuous education; provide open communication with opportunities for staff self-assessment and identification of skills gaps early in the project; provide competitive compensation and promotion policies and a rewarding work environment to retain staff; include training time in the project schedule.
Technology Risks	**Example Control Tactics**
Many faults are introduced interfacing to an unfamiliar commercial off-the-shelf (COTS) component.	Anticipate and schedule extra time for testing unfamiliar interfaces; invest training time for COTS components and for training with new tools; monitor, document, and publicize common errors and correct idioms; introduce new tools in lower-risk pilot projects or prototyping exercises.
Test and analysis automation tools do not meet expectations.	Introduce new tools in lower-risk pilot projects or prototyping exercises; anticipate and schedule time for training with new tools.
COTS components do not meet quality expectations.	Include COTS component qualification testing early in project plan; introduce new COTS components in lower-risk pilot projects or prototyping exercises.
Schedule Risks	**Example Control Tactics**
Inadequate unit testing leads to unanticipated expense and delays in integration testing.	Track and reward quality unit testing as evidenced by low-fault densities in integration.
Difficulty of scheduling meetings makes inspection a bottleneck in development.	Set aside times in a weekly schedule in which inspections take precedence over other meetings and other work; try distributed and asynchronous inspection techniques, with a lower frequency of face-to-face inspection meetings.

Risk Management in the Quality Plan: Risks Specific to Quality Management

Here we provide a brief overview of some risks specific to the quality process. Risks generic to process management are summarized in the sidebar at page 390.

Development Risks Poor quality software delivered to testing group or inadequate unit test and analysis before committing to the code base.	**Example Control Tactics** Provide early warning and feedback; schedule inspection of design, code and test suites; connect development and inspection to the reward system; increase training through inspection; require coverage or other criteria at unit test level.
Executions Risks Execution costs higher than planned; scarce resources available for testing (testing requires expensive or complex machines or systems not easily available.)	**Example Control Tactics** Minimize parts that require full system to be executed; inspect architecture to assess and improve testability; increase intermediate feedback; invest in scaffolding.
Requirements Risks High assurance critical requirements.	**Example Control Tactics** Compare planned testing effort with former projects with similar criticality level to avoid underestimating testing effort; balance test and analysis; isolate critical parts, concerns and properties.

One key aggregate measure is the number of faults that have been revealed and removed, which can be compared to data obtained from similar past projects. Fault detection and removal can be tracked against time and will typically follow a characteristic distribution similar to that shown in Figure 20.3. The number of faults detected per time unit tends to grow across several system builds, then to decrease at a much lower rate (usually half the growth rate) until it stabilizes.

An unexpected pattern in fault detection may be a symptom of problems. If detected faults stop growing earlier than expected, one might hope it indicates exceptionally high quality, but it would be wise to consider the alternative hypothesis that fault detection efforts are ineffective. A growth rate that remains high through more than half the planned system builds is a warning that quality goals may be met late or not at all, and may indicate weaknesses in fault removal or lack of discipline in development (e.g., a rush to add features before delivery, with a consequent deemphasis on quality control).

A second indicator of problems in the quality process is faults that remain open longer than expected. Quality problems are confirmed when the number of open faults does not stabilize at a level acceptable to stakeholders.

The accuracy with which we can predict fault data and diagnose deviations from expectation depends on the stability of the software development and quality processes, and on availability of data from similar projects. Differences between organizations and across application domains are wide, so by far the most valuable data is from

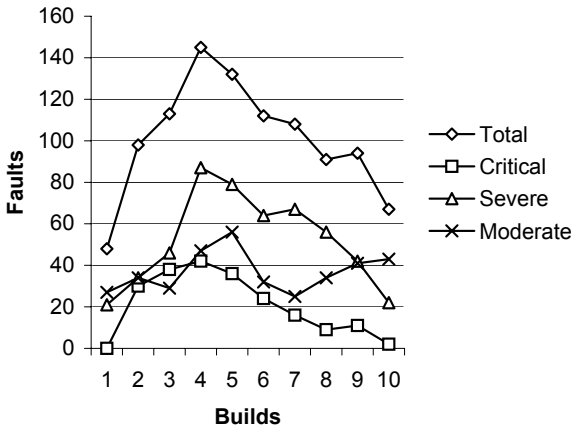


Figure 20.3: A typical distribution of faults for system builds through time.

similar projects in one's own organization.

The faultiness data in Figure 20.3 are aggregated by severity levels. This helps in better understanding the process. Growth in the number of *moderate* faults late in the development process may be a symptom of good use of limited resources concentrated in removing *critical* and *severe* faults, not spent solving *moderate* problems.

Accurate classification schemata can improve monitoring and may be used in very large projects, where the amount of detailed information cannot be summarized in overall data. The orthogonal defect classification (ODC) approach has two main steps: (1) fault classification and (2) fault analysis.

orthogonal defect classification (ODC)

ODC fault classification is done in two phases: when faults are detected and when they are fixed. At detection time, we record the *activity* executed when the fault is revealed, the *trigger* that exposed the fault, and the perceived or actual *impact* of the fault on the customer. A possible taxonomy for activities and triggers is illustrated in the sidebar at page 395. Notice that triggers depend on the activity. The sidebar at page 396 illustrates a possible taxonomy of customer impacts.

At fix time, we record *target*, *type*, *source*, and *age* of the software. The *target* indicates the entity that has been fixed to remove the fault, and can be *requirements*, *design*, *code*, *build/package*, or *documentation/development*. The *type* indicates the type of the fault. Taxonomies depend on the target. The sidebar at page 396 illustrates a taxonomy of types of faults removed from design or code. Fault types may be augmented with an indication of the nature of the fault, which can be: *missing*, that is, the fault is to due to an omission, as in a missing statement; *incorrect*, as in the use of a wrong parameter; or *extraneous*, that is, due to something not relevant or pertinent to the document or code, as in a section of the design document that is not pertinent to the current product and should be removed. The *source* of the fault indicates the origin of the faulty modules: *in-house*, *library*, *ported from other platforms*, or *outsourced* code.

The *age* indicates the age of the faulty element — whether the fault was found in *new*, *old (base)*, *rewritten*, or *re-fixed* code.

The detailed information on faults allows for many analyses that can provide information on the development and the quality process. As in the case of analysis of simple faultiness data, the interpretation depends on the process and the product, and should be based on past experience. The taxonomy of faults, as well as the analysis of faultiness data, should be refined while applying the method.

When we first apply the ODC method, we can perform some preliminary analysis using only part of the collected information:

Distribution of fault types versus activities: Different quality activities target different classes of faults. For example, algorithmic (that is, local) faults are targeted primarily by unit testing, and we expect a high proportion of faults detected by unit testing to be in this class. If the proportion of algorithmic faults found during unit testing is unusually small, or a larger than normal proportion of algorithmic faults are found during integration testing, then one may reasonably suspect that unit tests have not been well designed. If the mix of faults found during integration testing contains an unusually high proportion of algorithmic faults, it is also possible that integration testing has not focused strongly enough on interface faults.

Distribution of triggers over time during field test: Faults corresponding to simple usage should arise early during field test, while faults corresponding to complex usage should arise late. In both cases, the rate of disclosure of new faults should asymptotically decrease. Unexpected distributions of triggers over time may indicate poor system or acceptance test. If triggers that correspond to simple usage reveal many faults late in acceptance testing, we may have chosen a sample that is not representative of the user population. If faults continue growing during acceptance test, system testing may have failed, and we may decide to resume it before continuing with acceptance testing.

Age distribution over target code: Most faults should be located in new and rewritten code, while few faults should be found in base or re-fixed code, since base and re-fixed code has already been tested and corrected. Moreover, the proportion of faults in new and rewritten code with respect to base and re-fixed code should gradually increase. Different patterns may indicate holes in the fault tracking and removal process or may be a symptom of inadequate test and analysis that failed in revealing faults early (in previous tests of base or re-fixed code). For example, an increase of faults located in base code after porting to a new platform may indicate inadequate tests for portability.

Distribution of fault classes over time: The proportion of missing code faults should gradually decrease, while the percentage of extraneous faults may slowly increase, because missing functionality should be revealed with use and repaired, while extraneous code or documentation may be produced by updates. An increasing number of missing faults may be a symptom of instability of the product, while a sudden sharp increase in extraneous faults may indicate maintenance problems.

20.7 Improving the Process

Many classes of faults that occur frequently are rooted in process and development flaws. For example, a shallow architectural design that does not take into account resource allocation can lead to resource allocation faults. Lack of experience with the development environment, which leads to misunderstandings between analysts and programmers on rare and exceptional cases, can result in faults in exception handling. A performance assessment system that rewards faster coding without regard to quality is likely to promote low quality code.

The occurrence of many such faults can be reduced by modifying the process and environment. For example, resource allocation faults resulting from shallow architectural design can be reduced by introducing specific inspection tasks. Faults attributable to inexperience with the development environment can be reduced with focused training sessions. Persistently poor programming practices may require modification of the reward system.

Often, focused changes in the process can lead to product improvement and significant cost reduction. Unfortunately, identifying the weak aspects of a process can be extremely difficult, and often the results of process analysis surprise even expert managers. The analysis of the fault history can help software engineers build a feedback mechanism to track relevant faults to their root causes, thus providing vital information for improving the process. In some cases, information can be fed back directly into the current product development, but more often it helps software engineers improve the development of future products. For example, if analysis of faults reveals frequent occurrence of severe memory management faults in C programs, we might revise inspection checklists and introduce dynamic analysis tools, but it may be too late to change early design decisions or select a different programming language in the project underway. More fundamental changes may be made in future projects.

Δ root cause analysis

Root cause analysis (RCA) is a technique for identifying and eliminating process faults. RCA was first developed in the nuclear power industry and later extended to software analysis. It consists of four main steps to select significant classes of faults and track them back to their original causes: *What*, *When*, *Why*, and *How*.

What are the faults? The goal of this first step is to identify a class of important faults. Faults are categorized by severity and kind. The severity of faults characterizes the impact of the fault on the product. Although different methodologies use slightly different scales and terms, all of them identify a few standard levels, described in Table 20.1.

The RCA approach to categorizing faults, in contrast to ODC, does not use a predefined set of categories. The objective of RCA is not to compare different classes of faults over time, or to analyze and eliminate all possible faults, but rather to identify the few most important classes of faults and remove their causes. Successful application of RCA progressively eliminates the causes of the currently most important faults, which lose importance over time, so applying a static predefined classification would be useless. Moreover, the precision with which we identify faults depends on the specific project and process and varies over time.

ODC Classification of Triggers Listed by Activity

Design Review and Code Inspection

Design Conformance A discrepancy between the reviewed artifact and a prior-stage artifact that serves as its specification.

Logic/Flow An algorithmic or logic flaw.

Backward Compatibility A difference between the current and earlier versions of an artifact that could be perceived by the customer as a failure.

Internal Document An internal inconsistency in the artifact (e.g., inconsistency between code and comments).

Lateral Compatibility An incompatibility between the artifact and some other system or module with which it should interoperate.

Concurrency A fault in interaction of concurrent processes or threads.

Language Dependency A violation of language-specific rules, standards, or best practices.

Side Effects A potential undesired interaction between the reviewed artifact and some other part of the system.

Rare Situation An inappropriate response to a situation that is not anticipated in the artifact. (Error handling as specified in a prior artifact *design conformance*, not *rare situation*.)

Structural (White-Box) Test

Simple Path The fault is detected by a test case derived to cover a single program element.

Complex Path The fault is detected by a test case derived to cover a combination of program elements.

Functional (Black-Box) Test

Coverage The fault is detected by a test case derived for testing a single procedure (e.g., C function or Java method), without considering combination of values for possible parameters.

Variation The fault is detected by a test case derived to exercise a particular combination of parameters for a single procedure.

Sequencing The fault is detected by a test case derived for testing a sequence of procedure calls.

Interaction The fault is detected by a test case derived for testing procedure interactions.

System Test

Workload/Stress The fault is detected during workload or stress testing.

Recovery/Exception The fault is detected while testing exceptions and recovery procedures.

Startup/Restart The fault is detected while testing initialization conditions during start up or after possibly faulty shutdowns.

Hardware Configuration The fault is detected while testing specific hardware configurations.

Software Configuration The fault is detected while testing specific software configurations.

Blocked Test Failure occurred in setting up the test scenario.

ODC Classification of Customer Impact

Installability Ability of the customer to place the software into actual use. (Usability of the installed software is not included.)

Integrity/Security Protection of programs and data from either accidental or malicious destruction or alteration, and from unauthorized disclosure.

Performance The perceived and actual impact of the software on the time required for the customer and customer end users to complete their tasks.

Maintenance The ability to correct, adapt, or enhance the software system quickly and at minimal cost.

Serviceability Timely detection and diagnosis of failures, with minimal customer impact.

Migration Ease of upgrading to a new system release with minimal disruption to existing customer data and operations.

Documentation Degree to which provided documents (in all forms, including electronic) completely and correctly describe the structure and intended uses of the software.

Usability The degree to which the software and accompanying documents can be understood and effectively employed by the end user.

Standards The degree to which the software complies with applicable standards.

Reliability The ability of the software to perform its intended function without unplanned interruption or failure.

Accessibility The degree to which persons with disabilities can obtain the full benefit of the software system.

Capability The degree to which the software performs its intended functions consistently with documented system requirements.

Requirements The degree to which the system, in complying with document requirements, actually meets customer expectations

ODC Classification of Defect Types for Targets *Design* and *Code*

Assignment/Initialization A variable was not assigned the correct initial value or was not assigned any initial value.

Checking Procedure parameters or variables were not properly validated before use.

Algorithm/Method A correctness or efficiency problem that can be fixed by reimplementing a single procedure or local data structure, without a design change.

Function/Class/Object A change to the documented design is required to conform to product requirements or interface specifications.

Timing/Synchronization The implementation omits necessary synchronization of shared resources, or violates the prescribed synchronization protocol.

Interface/Object-Oriented Messages Module interfaces are incompatible; this can include syntactically compatible interfaces that differ in semantic interpretation of communicated data.

Relationship Potentially problematic interactions among procedures, possibly involving different assumptions but not involving interface incompatibility.

Level	Description	Example
Critical	The product is unusable.	The fault causes the program to crash.
Severe	Some product features cannot be used, and there is no workaround.	The fault inhibits importing files saved with a previous version of the program, and there is no way to convert files saved in the old format to the new one.
Moderate	Some product features require workarounds to use, and reduce efficiency, reliability, or convenience and usability.	The fault inhibits exporting in Postscript format. Postscript can be produced using the printing facility, but the process is not obvious or documented (loss of usability) and requires extra steps (loss of efficiency).
Cosmetic	Minor inconvenience.	The fault limits the choice of colors for customizing the graphical interface, violating the specification but causing only minor inconvenience.

Table 20.1: Standard severity levels for root cause analysis (RCA).

A good RCA classification should follow the uneven distribution of faults across categories. If, for example, the current process and the programming style and environment result in many interface faults, we may adopt a finer classification for interface faults and a coarse-grain classification of other kinds of faults. We may alter the classification scheme in future projects as a result of having identified and removed the causes of many interface faults.

Classification of faults should be sufficiently precise to allow identifying one or two most significant classes of faults considering severity, frequency, and cost of repair. It is important to keep in mind that severity and repair cost are not directly related. We may have cosmetic faults that are very expensive to repair, and critical faults that can be easily repaired. When selecting the target class of faults, we need to consider all the factors. We might, for example, decide to focus on a class of moderately severe faults that occur very frequently and are very expensive to remove, investing fewer resources in preventing a more severe class of faults that occur rarely and are easily repaired.

When did faults occur, and when were they found? It is typical of mature software processes to collect fault data sufficient to determine when each fault was detected (e.g., in integration test or in a design inspection). In addition, for the class of faults identified in the first step, we attempt to determine when those faults were introduced (e.g., was a particular fault introduced in coding, or did it result from an error in architectural design?).

Why did faults occur? In this core RCA step, we attempt to trace representative faults back to causes, with the objective of identifying a "root" cause associated with many faults in the class. Analysis proceeds iteratively by attempting to explain the

The *80/20* or *Pareto* Rule

Fault classification in root cause analysis is justified by the so-called *80/20* or *Pareto* rule. The Pareto rule is named for the Italian economist Vilfredo Pareto, who in the early nineteenth century proposed a mathematical power law formula to describe the unequal distribution of wealth in his country, observing that 20% of the people owned 80% of the wealth.

Pareto observed that in many populations, a few (20%) are vital and many (80%) are trivial. In fault analysis, the Pareto rule postulates that 20% of the code is responsible for 80% of the faults. Although proportions may vary, the rule captures two important facts:

1. Faults tend to accumulate in a few modules, so identifying potentially faulty modules can improve the cost effectiveness of fault detection.

2. Some classes of faults predominate, so removing the causes of a predominant class of faults can have a major impact on the quality of the process and of the resulting product.

The predominance of a few classes of faults justifies focusing on one class at a time.

error that led to the fault, then the cause of that error, the cause of that cause, and so on. The rule of thumb "ask why six times" does not provide a precise stopping rule for the analysis, but suggests that several steps may be needed to find a cause in common among a large fraction of the fault class under consideration.

Tracing the causes of faults requires experience, judgment, and knowledge of the development process. We illustrate with a simple example. Imagine that the first RCA step identified *memory leaks* as the most significant class of faults, combining a moderate frequency of occurrence with severe impact and high cost to diagnose and repair. The group carrying out RCA will try to identify the cause of memory leaks and may conclude that many of them result from *forgetting to release memory in exception handlers*. The RCA group may trace this problem in exception handling to lack of information: *Programmers can't easily determine what needs to be cleaned up in exception handlers*. The RCA group will ask *why* once more and may go back to a design error: *The resource management scheme assumes normal flow of control* and thus does not provide enough information to guide implementation of exception handlers. Finally, the RCA group may identify the root problem in an early design problem: *Exceptional conditions were an afterthought dealt with late in design*.

Each step requires information about the class of faults and about the development process that can be acquired through inspection of the documentation and interviews with developers and testers, but the key to success is curious probing through several levels of cause and effect.

How could faults be prevented? The final step of RCA is improving the process by removing root causes or making early detection likely. The measures taken may have

a minor impact on the development process (e.g., adding consideration of exceptional conditions to a design inspection checklist), or may involve a substantial modification of the process (e.g., making explicit consideration of exceptional conditions a part of all requirements analysis and design steps). As in tracing causes, prescribing preventative or detection measures requires judgment, keeping in mind that the goal is not perfection but cost-effective improvement.

ODC and RCA are two examples of feedback and improvement, which are an important dimension of most good software processes. Explicit process improvement steps are, for example, featured in both SRET (sidebar on page 380) and Cleanroom (sidebar on page 378).

20.8 The Quality Team

The quality plan must assign roles and responsibilities to people. As with other aspects of planning, assignment of responsibility occurs at a strategic level and a tactical level. The tactical level, represented directly in the project plan, assigns responsibility to individuals in accordance with the general strategy. It involves balancing level of effort across time and carefully managing personal interactions. The strategic level of organization is represented not only in the quality strategy document, but in the structure of the organization itself.

The strategy for assigning responsibility may be partly driven by external requirements. For example, independent quality teams may be required by certification agencies or by a client organization. Additional objectives include ensuring sufficient accountability that quality tasks are not easily overlooked; encouraging objective judgment of quality and preventing it from being subverted by schedule pressure; fostering shared commitment to quality among all team members; and developing and communicating shared knowledge and values regarding quality.

Measures taken to attain some objectives (e.g., autonomy to ensure objective assessment) are in tension with others (e.g., cooperation to meet overall project objectives). It is therefore not surprising to find that different organizations structure roles and responsibilities in a wide variety of different ways. The same individuals can play the roles of developer and tester, or most testing responsibility can be assigned to members of a distinct group, and some may even be assigned to a distinct organization on a contractual basis. Oversight and accountability for approving the work product of a task are sometimes distinguished from responsibility for actually performing a task, so the team organization is somewhat intertwined with the task breakdown.

Each of the possible organizations of quality roles makes some objectives easier to achieve and some more challenging. Conflict of one kind or another is inevitable, and therefore in organizing the team it is important to recognize the conflicts and take measures to control adverse consequences. If an individual plays two roles in potential conflict (e.g., a developer responsible for delivering a unit on schedule is also responsible for integration testing that could reveal faults that delay delivery), there must be countermeasures to control the risks inherent in that conflict. If roles are assigned to different individuals, then the corresponding risk is conflict between the individuals

(e.g., if a developer and a tester do not adequately share motivation to deliver a quality product on schedule).

An independent and autonomous testing team lies at one end of the spectrum of possible team organizations. One can make that team organizationally independent so that, for example, a project manager with schedule pressures can neither bypass quality activities or standards, nor reallocate people from testing to development, nor postpone quality activities until too late in the project. Separating quality roles from development roles minimizes the risk of conflict between roles played by an individual, and thus makes most sense for roles in which independence is paramount, such as final system and acceptance testing. An independent team devoted to quality activities also has an advantage in building specific expertise, such as test design. The primary risk arising from separation is in conflict between goals of the independent quality team and the developers.

When quality tasks are distributed among groups or organizations, the plan should include specific checks to ensure successful completion of quality activities. For example, when module testing is performed by developers and integration and system testing is performed by an independent quality team, the quality team should check the completeness of module tests performed by developers, for example, by requiring satisfaction of coverage criteria or inspecting module test suites. If testing is performed by an independent organization under contract, the contract should carefully describe the testing process and its results and documentation, and the client organization should verify satisfactory completion of the contracted tasks.

Existence of a testing team must not be perceived as relieving developers from responsibility for quality, nor is it healthy for the testing team to be completely oblivious to other pressures, including schedule pressure. The testing team and development team, if separate, must at least share the goal of shipping a high-quality product on schedule.

Independent quality teams require a mature development process to minimize communication and coordination overhead. Test designers must be able to work on sufficiently precise specifications and must be able to execute tests in a controllable test environment. Versions and configurations must be well defined, and failures and faults must be suitably tracked and monitored across versions.

It may be logistically impossible to maintain an independent quality group, especially in small projects and organizations, where flexibility in assignments is essential for resource management. Aside from the logistical issues, division of responsibility creates additional work in communication and coordination. Finally, quality activities often demand deep knowledge of the project, particularly at detailed levels (e.g., unit and early integration test). An outsider will have less insight into how and what to test, and may be unable to effectively carry out the crucial earlier activities, such as establishing acceptance criteria and reviewing architectural design for testability. For all these reasons, even organizations that rely on an independent verification and validation (IV&V) group for final product qualification allocate other responsibilities to developers and to quality professionals working more closely with the development team.

At the polar opposite from a completely independent quality team is full integration of quality activities with development, as in some “agile” processes including XP.

Communication and coordination overhead is minimized this way, and developers take full responsibility for the quality of their work product. Moreover, technology and application expertise for quality tasks will match the expertise available for development tasks, although the developer may have less specific expertise in skills such as test design.

The more development and quality roles are combined and intermixed, the more important it is to build into the plan checks and balances to be certain that quality activities and objective assessment are not easily tossed aside as deadlines loom. For example, XP practices like "test first" together with pair programming (sidebar on page 381) guard against some of the inherent risks of mixing roles.

Separate roles do not necessarily imply segregation of quality activities to distinct individuals. It is possible to assign both development and quality responsibility to developers, but assign two individuals distinct responsibilities for each development work product. Peer review is an example of mixing roles while maintaining independence on an item-by-item basis. It is also possible for developers and testers to participate together in some activities.

Many variations and hybrid models of organization can be designed. Some organizations have obtained a good balance of benefits by rotating responsibilities. For example, a developer may move into a role primarily responsible for quality in one project and move back into a regular development role in the next. In organizations large enough to have a distinct quality or testing group, an appropriate balance between independence and integration typically varies across levels of project organization. At some levels, an appropriate balance can be struck by giving responsibility for an activity (e.g., unit testing) to developers who know the code best, but with a separate oversight responsibility shared by members of the quality team. For example, unit tests may be designed and implemented by developers, but reviewed by a member of the quality team for effective automation (particularly, suitability for automated regression test execution as the product evolves) as well as thoroughness. The balance tips further toward independence at higher levels of granularity, such as in system and acceptance testing, where at least some tests should be designed independently by members of the quality team.

Outsourcing test and analysis activities is sometimes motivated by the perception that testing is less technically demanding than development and can be carried out by lower-paid and lower-skilled individuals. This confuses test execution, which should in fact be straightforward, with analysis and test design, which are as demanding as design and programming tasks in development. Of course, less skilled individuals *can* design and carry out tests, just as less skilled individuals *can* design and write programs, but in both cases the results are unlikely to be satisfactory.

Outsourcing can be a reasonable approach when its objectives are not merely minimizing cost, but maximizing independence. For example, an independent judgment of quality may be particularly valuable for final system and acceptance testing, and may be essential for measuring a product against an independent quality standard (e.g., qualifying a product for medical or avionic use). Just as an organization with mixed roles requires special attention to avoid the conflicts between roles played by an individual, radical separation of responsibility requires special attention to control conflicts

between the quality assessment team and the development team.

The plan must clearly define milestones and delivery for outsourced activities, as well as checks on the quality of delivery in both directions: Test organizations usually perform quick checks to verify the consistency of the software to be tested with respect to some minimal "testability" requirements; clients usually check the completeness and consistency of test results. For example, test organizations may ask for the results of inspections on the delivered artifact before they start testing, and may include some quick tests to verify the installability and testability of the artifact. Clients may check that tests satisfy specified functional and structural coverage criteria, and may inspect the test documentation to check its quality. Although the contract should detail the relation between the development and the testing groups, ultimately, outsourcing relies on mutual trust between organizations.

Open Research Issues

Orthogonal defect classification (introduced in the 1990s) and root cause analysis (introduced in the 1980s) remain key techniques for deriving useful guidance from experience. Considering widespread agreement on the importance of continuous process improvement, we should expect innovation and adaptation of these key techniques for current conditions. An example is the renewed interest in fault-proneness models, exploiting the rich historical data available in version control systems and bug tracking databases.

Globally distributed software teams and teams that span multiple companies and organizations pose many interesting challenges for software development in general and test and analysis in particular. We expect that both technical and management innovations will adapt to these important trends, with increasing interplay between research in software test and analysis and research in computer-supported collaborative work (CSCW).

Further Reading

IEEE publishes a standard for software quality assurance plans [Ins02], which serves as a good starting point. The plan outline in this chapter is based loosely on the IEEE standard. Jaaksi [Jaa03] provides a useful discussion of decision making based on distribution of fault discovery and resolution over the course of a project, drawn from experience at Nokia. Chaar et al. [CHBC93] describe the orthogonal defect classification technique, and Bhandari et al. [BHC+94] provide practical details useful in implementing it. Leszak et al. [LPS02] describe a retrospective process with root cause analysis, process compliance analysis, and software complexity analysis. Denaro and Pezzè [DP02] describe fault-proneness models for allocating effort in a test plan. DeMarco and Lister [DL99] is a popular guide to the human dimensions of managing software teams.

Exercises

20.1. Testing compatibility with a variety of device drivers is a significant cost and schedule factor in some projects. For example, a well-known developer of desktop publishing software maintains a test laboratory containing dozens of current and outdated models of Macintosh computer, running several operating system versions.

Put yourself in the place of the quality manager for a new version of this desktop publishing software, and consider in particular the printing subsystem of the software package. Your goal is to minimize the schedule impact of testing the software against a large number of printers, and in particular to reduce the risk that serious problems in the printing subsystem surface late in the project, or that testing on the actual hardware delays product release.

How can the software architectural design be organized to serve your goals of reducing cost and risk? Do you expect your needs in this regard will be aligned with those of the development manager, or in conflict? What other measures might you take in project planning, and in particular in the project schedule, to minimize risks of problems arising when the software is tested in an operational environment? Be as specific as possible, and avoid simply restating the general strategies presented in this chapter.

20.2. Chipmunk Computers has signed an agreement with a software house for software development under contract. Project leaders are encouraged to take advantage of this agreement to outsource development of some modules and thereby reduce project cost. Your project manager asks you to analyze the risks that may result from this choice and propose approaches to reduce the impact of the identified risks. What would you suggest?

20.3. Suppose a project applied orthogonal defect classification and analyzed correlation between fault types and fault triggers, as well as between fault types and impact. What useful information could be derived from cross-correlating those classifications, beyond the information available from each classification alone?

20.4. ODC attributes have been adapted and extended in several ways, one of which is including *fault qualifier*, which distinguishes whether the fault is due to missing, incorrect, or extraneous code. What attributes might *fault qualifier* be correlated with, and what useful information might thereby be obtained?

Chapter 24

Documenting Analysis and Test

Mature software processes include documentation standards for all the activities of the software process, including test and analysis activities. Documentation can be inspected to verify progress against schedule and quality goals and to identify problems, supporting process visibility, monitoring, and replicability.

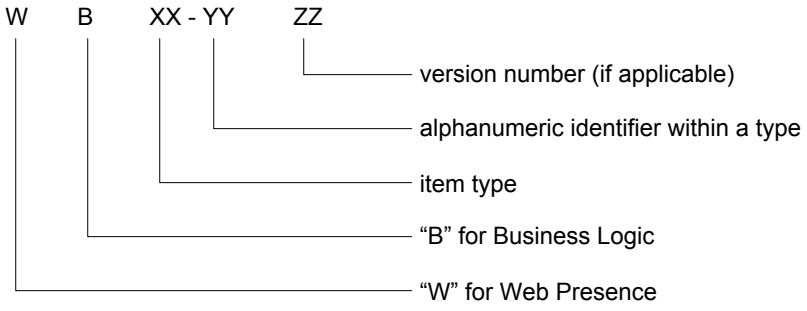
Required Background

- Chapter 20
  This chapter describes test and analysis strategy and plans, which are intertwined with documentation. Plans and strategy documents are part of quality documentation, and quality documents are used in process monitoring.

24.1 Overview

Documentation is an important element of the software development process, including the quality process. Complete and well-structured documents increase the reusability of test suites within and across projects. Documents are essential for maintaining a body of knowledge that can be reused across projects. Consistent documents provide a basis for monitoring and assessing the process, both internally and for external authorities where certification is desired. Finally, documentation includes summarizing and presenting data that forms the basis for process improvement. Test and analysis documentation includes summary documents designed primarily for human comprehension and details accessible to the human reviewer but designed primarily for automated analysis.

Documents are divided into three main categories: planning, specification, and reporting. *Planning documents* describe the organization of the quality process and include strategies and plans for the division or the company, and plans for individual projects. *Specification documents* describe test suites and test cases. A complete set of



analysis and test documentation

WB05-YYZZ	analysis and test strategy
WB06-YYZZ	analysis and test plan
WB07-YYZZ	test design specifications
WB08-YYZZ	test case specification
WB09-YYZZ	checklists
WB10-YYZZ	analysis and test logs
WB11-YYZZ	analysis and test summary reports
WB12-YYZZ	other analysis and test documents

Figure 24.1: Sample document naming conventions, compliant with IEEE standards.

analysis and test specification documents include test design specifications, test case specification, checklists, and analysis procedure specifications. *Reporting documents* include details and summary of analysis and test results.

24.2 Organizing Documents

In a small project with a sufficiently small set of documents, the arrangement of other project artifacts (e.g., requirements and design documents) together with standard content (e.g., mapping of subsystem test suites to the build schedule) provides sufficient organization to navigate through the collection of test and analysis documentation. In larger projects, it is common practice to produce and regularly update a global guide for navigating among individual documents.

Mature processes require all documents to contain metadata that facilitate their management. Documents must include some basic information about its *context* in order to make the document self-contained, *approval* indicating the persons responsible for the document and *document history*, as illustrated in the template on page 457.

Naming conventions help in quickly identifying documents. A typical standard for document names would include keywords indicating the general scope of the document, its nature, the specific document, and its version, as in Figure 24.1.

Chipmunk Document Template

Document Title

Approvals

issued by	*name*	*signature*	*date*
approved by	*name*	*signature*	*date*
distribution status	*(internal use only, restricted, ...)*		
distribution list	*(people to whom the document must be sent)*		

History

version	description

Table of Contents

List of sections.

Summary

Summarize the contents of the document. The summary should clearly explain the relevance of the document to its possible uses.

Goals of the document

Describe the purpose of this document: Who should read it, and why?

Required documents and references

Provide a reference to other documents and artifacts needed for understanding and exploiting this document. Provide a rationale for the provided references.

Glossary

Provide a glossary of terms required to understand this document.

Section 1

. . .

Section N

. . .

24.3 Test Strategy Document

Analysis and test strategies (Chapter 20) describe quality guidelines for sets of projects, usually for an entire company or organization. Strategies, and therefore strategy documents, vary widely among organizations, but we can identify a few key elements that should be included in almost any well-designed strategy document. These are illustrated in the document excerpt on page 459.

overall quality

Strategy documents indicate common quality requirements across products. Requirements may depend on business conditions. For example, a company that produces safety-critical software may need to satisfy minimum dependability requirements defined by a certification authority, while a department that designs software embedded in hardware products may need to ensure portability across product lines. Some requirements on dependability and usability may be necessary to maintain brand image and market position. For example, a company might decide to require conformance to W3C-WAI accessibility standards (see Chapter 22) uniformly across the product line.

The strategy document sets out requirements on other quality documents, typically including an analysis and test plan, test design specifications, test case specifications, test logs, and test summary reports. Basic document requirements, such as naming and versioning, follow standards for other project documentation, but quality documents may have additional, specialized requirements. For example, testing logs for avionics software may be required to contain references to the version of the simulator used for executing the test before installing the software on board the aircraft.

documentation quality

24.4 Analysis and Test Plan

While the format of an analysis and test strategy vary from company to company, the structure of an analysis and test plan is more standardized. A typical structure of a test and analysis plan includes information about items to be verified, features to be tested, the testing approach, pass and fail criteria, test deliverables, tasks, responsibilities and resources, and environment constraints. Basic elements are described in the sidebar on page 461.

The overall quality plan usually comprises several individual plans of limited scope. Each test and analysis plan should indicate the items to be verified through analysis or testing. They may include specifications or documents to be inspected, code to be analyzed or tested, and interface specifications to undergo consistency analysis. They may refer to the whole system or part of it — like a subsystem or a set of units. Where the project plan includes planned development increments, the analysis and test plan indicates the applicable versions of items to be verified.

items to be verified

For each item, the plan should indicate any special hardware or external software required for testing. For example, the plan might indicate that one suite of subsystem tests for a security package can be executed with a software simulation of a smart card reader, while another suite requires access to the physical device. Finally, for each item, the plan should reference related documentation, such as requirements and design specifications, and user, installation, and operations guides.

A test and analysis plan may not address all aspects of software quality and testing

An Excerpt of the Chipmunk Analysis and Test Strategy

Document CP05-14.03: Analysis and Test Strategy

. . .

Applicable Standards and Procedures

Artifact	*Applicable Standards and Guidelines*
Web application	Accessibility: W3C-WAI . . .
Reusable component (internally developed)	Inspection procedure: [WB12-03.12]
External component	Qualification procedure: [WB12-22.04]

. . .

Documentation Standards

Project documents must be archived according to the standard Chipmunk archive procedure [WB02-01.02]. Standard required documents include

Document	*Content & Organization Standard*
Quality plan	[WB06-01.03]
Test design specifications	[WB07-01.01] (per test suite)
Test case specifications	[WB08-01.07] (per test suite)
Test logs	[WB10-02.13]
Test summary reports	[WB11-01.11]
Inspection reports	[WB12-09.01]

. . .

Analysis and Test Activities

. . .

Tools

The following tools are approved and should be used in all development projects. Exceptions require configuration committee approval and must be documented in the project plan.

Fault logging	Chipmunk BgT [WB10-23.01]
. . .	

. . .

Staff and Roles

A development work unit consists of unit source code, including unit test cases, stubs, and harnesses, and unit test documentation. A unit may be committed to the project baseline when the source code, test cases, and test results have passed peer review.

. . .

References

[WB02-01.02] *Archive Procedure*

[WB06-01.03] *Quality Plan Guidelines*

[WB07-01.01] *Test Design Specifications Guidelines*

[WB08-01.07] *Test Case Specifications Guidelines*

[WB11-01.11] *Summary Reports Template*

[WB10-02.13] *Test Log Template*

[WB11-09.01] *Inspection Report Template*

[WB12-03.12] *Standard Inspection Procedures*

[WB12-22.04] *Quality Procedures for Software Developed by Third Parties*

[WB12-23.01] *BgT Installation Manual and User Guide*

. . .

features to be analyzed or tested

activities. It should indicate the features to be verified and those that are excluded from consideration (usually because responsibility for them is placed elsewhere). For example, if the item to be verified includes a graphical user interface, the test and analysis plan might state that it deals only with functional properties and not with usability, which is to be verified separately by a usability and human interface design team.

Explicit indication of features *not* to be tested, as well as those included in an analysis and test plan, is important for assessing completeness of the overall set of analysis and test activities. Assumption that a feature not considered in the current plan is covered at another point is a major cause of missing verification in large projects.

suspend and resume criteria

The quality plan must clearly indicate criteria for deciding the success or failure of each planned activity, as well as the conditions for suspending and resuming analysis and test.

test deliverables

Plans define items and documents that must be produced during verification. Test deliverables are particularly important for regression testing, certification, and process improvement. We will see the details of analysis and test documentation in the next section.

tasks and schedule

The core of an analysis and test plan is a detailed schedule of tasks. The schedule is usually illustrated with GANTT and PERT diagrams showing the relation among tasks as well as their relation to other project milestones.[1] The schedule includes the allocation of limited resources (particularly staff) and indicates responsibility for results.

resources and responsibilities

A quality plan document should also include an explicit risk plan with contingencies. As far as possible, contingencies should include unambiguous triggers (e.g., a date on which a contingency is activated if a particular task has not be completed) as well as recovery procedures.

environmental needs

Finally, the test and analysis plan should indicate scaffolding, oracles, and any other software or hardware support required for test and analysis activities.

24.5 Test Design Specification Documents

Design documentation for test suites and test cases serve essentially the same purpose as other software design documentation, guiding further development and preparing for maintenance. Test suite design must include all the information needed for initial selection of test cases and maintenance of the test suite over time, including rationale and anticipated evolution. Specification of individual test cases includes purpose, usage, and anticipated changes.

Test design specification documents describe complete test suites (i.e., sets of test cases that focus on particular aspects, elements, or phases of a software project). They may be divided into unit, integration, system, and acceptance test suites, if we organize them by the granularity of the tests, or functional, structural, and performance test suites, if the primary organization is based on test objectives. A large project may include many test design specifications for test suites of different kinds and granularity,

[1] Project scheduling is discussed in more detail in Chapter 20.

A Standard Organization of an Analysis and Test Plan

Analysis and test items:
The items to be tested or analyzed. The description of each item indicates version and installation procedures that may be required.

Features to be tested:
The features considered in the plan.

Features not to be tested:
Features not considered in the current plan.

Approach:
The overall analysis and test approach, sufficiently detailed to permit identification of the major test and analysis tasks and estimation of time and resources.

Pass/Fail criteria:
Rules that determine the status of an artifact subjected to analysis and test.

Suspension and resumption criteria:
Conditions to trigger suspension of test and analysis activities (e.g., an excessive failure rate) and conditions for restarting or resuming an activity.

Risks and contingencies:
Risks foreseen when designing the plan and a contingency plan for each of the identified risks.

Deliverables:
A list all A&T artifacts and documents that must be produced.

Task and schedule:
A complete description of analysis and test tasks, relations among them, and relations between A&T and development tasks, with resource allocation and constraints. A task schedule usually includes GANTT and PERT diagrams.

Staff and responsibilities:
Staff required for performing analysis and test activities, the required skills, and the allocation of responsibilities among groups and individuals. Allocation of resources to tasks is described in the schedule.

Environmental needs:
Hardware and software required to perform analysis or testing activities.

and for different versions or configurations of the system and its components. Each specification should be uniquely identified and related to corresponding project documents, as illustrated in the sidebar on page 463.

Test design specifications identify the features they are intended to verify and the approach used to select test cases. Features to be tested should be cross-referenced to relevant parts of a software specification or design document. The test case selection approach will typically be one of the test selection techniques described in Chapters 10 through 16 with documentation on how the technique has been applied.

A test design specification also includes description of the testing procedure and pass/fail criteria. The procedure indicates steps required to set up the testing environment and perform the tests, and includes references to scaffolding and oracles. Pass/fail criteria distinguish success from failure of a test suite as a whole. In the simplest case a test suite execution may be determined to have failed if any individual test case execution fails, but in system and acceptance testing it is common to set a tolerance level that may depend on the number and severity of failures.

A test design specification logically includes a list of test cases. Test case specifications may be physically included in the test design specification document, or the logical inclusion may be implemented by some form of automated navigation. For example, a navigational index can be constructed from references in test case specifications.

Individual test case specifications elaborate the test design for each individual test case, defining test inputs, required environmental conditions and procedures for test execution, as well as expected outputs or behavior. The environmental conditions may include hardware and software as well as any other requirements. For example, while most tests should be executed automatically without human interaction, intervention of personnel with certain special skills (e.g., a device operator) may be an environmental requirement for some.

A test case specification indicates the item to be tested, such as a particular module or product feature. It includes a reference to the corresponding test design document and describes any dependence on execution of other test cases. Like any standard document, a test case specification is labeled with a unique identifier. A sample test case specification is provided on page 464.

24.6 Test and Analysis Reports

Reports of test and analysis results serve both developers and test designers. They identify open faults for developers and aid in scheduling fixes and revisions. They help test designers assess and refine their approach, for example, noting when some class of faults is escaping early test and analysis and showing up only in subsystem and system testing (see Section 20.6, page 389).

A prioritized list of open faults is the core of an effective fault handling and repair procedure. Failure reports must be consolidated and categorized so that repair effort can be managed systematically, rather than jumping erratically from problem to problem and wasting time on duplicate reports. They must be prioritized so that effort is not

Functional Test Design Specification of check_configuration

Test Suite Identifier
WB07-15.01

Features to Be Tested
Functional test for check_configuration, module specification WB02-15.32.[a]

Approach
Combinatorial functional test of feature parameters, enumerated by category-partition method over parameter table on page 3 of this document.[b]

Procedure
Designed for conditional inclusion in nightly test run. Build target T02_15_32_11 includes JUnit harness and oracles, with test reports directed to standard test log. Test environment includes table MDB_15_32_03 for loading initial test database state.

Test cases[c]

WB07-15.01.C01	malformed model number
WB07-15.01.C02	model number not in DB
...	...
WB07-15.01.C09[d]	valid model number with all legal required slots and some legal optional slots
...	...
WB07-15.01.C19	empty model DB
WB07-15.01.C23	model DB with a single element
WB07-15.01.C24	empty component DB
WB07-15.01.C29	component DB with a single element

Pass/Fail Criterion
Successful completion requires correct execution of all test cases with no violations in test log.

[a] An excerpt of specification WB02-15.32 is presented in Figure 11.1, page 182.
[b] Reproduced in Table 11.1, page 187.
[c] The detailed list of test cases is produced automatically from the test case file, which in turn is generated from the specification of categories and partitions. The test suite is implicitly referenced by individual test case numbers (e.g., WB07-15.01.C09 is a test case in test suite WB07-15.01).
[d] See sample test case specification, page 464.

Test Case Specification for check_configuration

Test Case Identifier
WB07-15.01.C09[a]

Test items
Module check_configuration of the Chipmunk Web presence system, business logic subsystem.

Input specification
Test Case Specification:

Model No.	valid
No. of required slots for selected model (#SMRS)	many
No. of optional slots for selected model (#SMOS)	many
Correspondence of selection with model slots	complete
No. of required components with selection $\neq$ empty	$=$ No. of required slots
No. of optional components with select $\neq$ empty	$<$ No. of optional slots
Required component selection	all valid
Optional component selection	all valid
No. of models in DB	many
No. of components in DB	many

Test case:

Model number	Chipmunk C20
#SMRS	5
Screen	13”
Processor	Chipmunk II plus
Hard disk	30 GB
RAM	512 MB
OS	RodentOS 3.2 Personal Edition
#SMOS	4
External storage device	DVD player

Output Specification
return value valid

Environment Needs
Execute with ChipmunkDBM v3.4 database initialized from table MDB_15_32_03.

Special Procedural Requirements
none

Intercase Dependencies
none

[a]The prefix WB07-15.01 implicitly references a test suite to which this test case directly belongs. That test suite may itself be a component of higher level test suites, so logically the test case also belongs to any of those test suites. Furthermore, some additional test suites may be composed of selections from other test suites.

squandered on faults of relatively minor importance while critical faults are neglected or even forgotten.

Other reports should be crafted to suit the particular needs of an organization and project, including process improvement as described in Chapter 23. Summary reports serve primarily to track progress and status. They may be as simple as confirmation that the nightly build-and-test cycle ran successfully with no new failures, or they may provide somewhat more information to guide attention to potential trouble spots. Detailed test logs are designed for selective reading, and include summary tables that typically include the test suites executed, the number of failures, and a breakdown of failures into those repeated from prior test execution, new failures, and test cases that previously failed but now execute correctly.

In some domains, such as medicine or avionics, the content and form of test logs may be prescribed by a certifying authority. For example, some certifications require test execution logs signed by both the person who performed the test and a quality inspector, who ascertains conformance of the test execution with test specifications.

Open Research Issues

Many available tools generate documentation from test execution records and the tables used to generate test specifications, minimizing the extra effort of producing documents in a useful form. Test design derived automatically or semiautomatically from design models is growing in importance, as is close linking of program documentation with source code, ranging from simple comment extraction and indexing like Javadoc to sophisticated hypermedia systems. In the future we should see these trends converge, and expect to see test documentation fit in an overall framework for managing and navigating information on a software product and project.

Further Reading

The guidelines in this chapter are based partly on IEEE Standard 829-1998 [Ins98]. Summary reports must convey information efficiently, managing both overview and access to details. Tufte's books on information design are useful sources of principles and examples. The second [Tuf90] and fourth [Tuf06] volumes in the series are particularly relevant. Experimental hypermedia software documentation systems [ATWJ00] hint at possible future systems that incorporate test documentation with other views of an evolving software product.

Exercises

24.1. Agile software development methods (XP, Scrum, etc.) typically minimize documentation written during software development. Referring to the sidebar on page 381, identify standard analysis and test documents that could be generated automatically or semiautomatically or replaced with functionally equivalent, automatically generated documentation during an XP project.

24.2. Test documents may become very large and unwieldy. Sometimes a more compact specification of several test cases together is more useful than individual specifications of each test case. Referring to the test case specification on page 464, design a tabular form to compactly document a suite of similar test case specifications.

24.3. Design a checklist for inspecting test design specification documents.

24.4. The Chipmunk Web presence project is starting up, and it has been decided that all project artifacts, including requirements documents, documentation in English, Italian, French, and German, source code, test plans, and test suites, will be managed in one or more CVS repositories.[2] The project team is divided between Milan, Italy, and Eugene, Oregon. What are the main design choices and issues you will consider in designing the organization of the version control repositories?

[2] If you are more familiar with another version control system, such as Subversion or Perforce, you may substitute it for CVS.