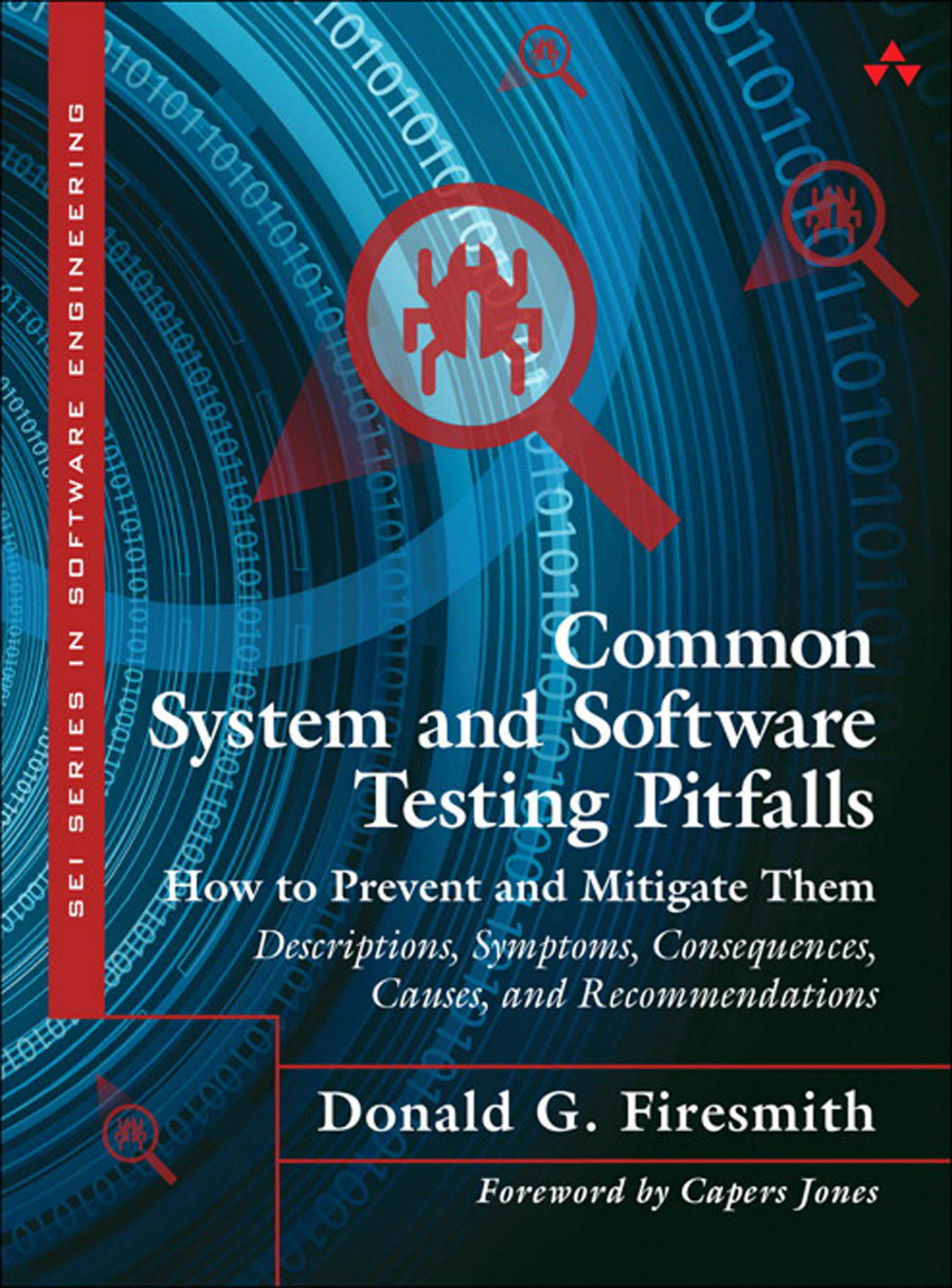

SEI SERIES IN SOFTWARE ENGINEERING
Common System and Software Testing Pitfalls
How to Prevent and Mitigate Them
Descriptions, Symptoms, Consequences, Causes, and Recommendations
Donald G. Firesmith
Foreword by Capers Jones

CONTENTS

CHAPTER 1

OVERVIEW

1.1 What Is Testing?

Testing is the activity of executing a system, subsystem, or component under specific preconditions (for example, pretest mode, states, stored data, and external conditions) with specific inputs so that its actual behavior (outputs and postconditions) can be compared with its required or expected behavior.

Testing differs from other verification and validation methods (for example, analysis, demonstration, and inspection) in that it is a dynamic, as opposed to a static, analysis method that involves the actual execution of the thing being tested.

Testing has the following goals:

- Primary goal:
  - Enable the system under test (SUT) to be improved by:
    - "Breaking" it (that is, by causing faults and failures)
    - Exposing its defects so that they can be fixed
- Secondary goals:
  - Provide adequate confidence based on sufficient objective evidence regarding the SUT's:
    - Quality

      A system's quality is not just its lack of defects or its correctness (in terms of meeting its requirements). A system must also have the necessary levels of relevant quality characteristics and attributes; for example, availability, capacity, extensibility, maintainability, performance, portability, reliability, robustness, safety, security, and usability.
    - Fitness for purpose
    - Readiness for shipping, deployment, or being placed into operation

1.2 Testing and the V Models

Figure 1.1 illustrates a common way of modeling system engineering: the traditional V Model of system engineering activities.[1] On the left side of the V are the analysis activities that decompose the users' problem into small, manageable pieces. Similarly, the right side of the V shows the synthesis activities that aggregate (and test) these pieces into the system that solves the users' problem.

While useful, the traditional V model does not really represent system engineering from the tester's viewpoint. The next three figures show three increasingly detailed V models that better capture the testing-specific aspects of system engineering.

Figure 1.2 illustrates a V model oriented around work products rather than activities. Specifically, these are the major executable work products because testing involves the execution of work products. In this case, the left side of the V illustrates the analysis of ever more detailed executable models, whereas the right side of the V illustrates the corresponding incremental and iterative synthesis of the actual system. This V model shows the executable things that are tested rather than the general system engineering activities that generate them.

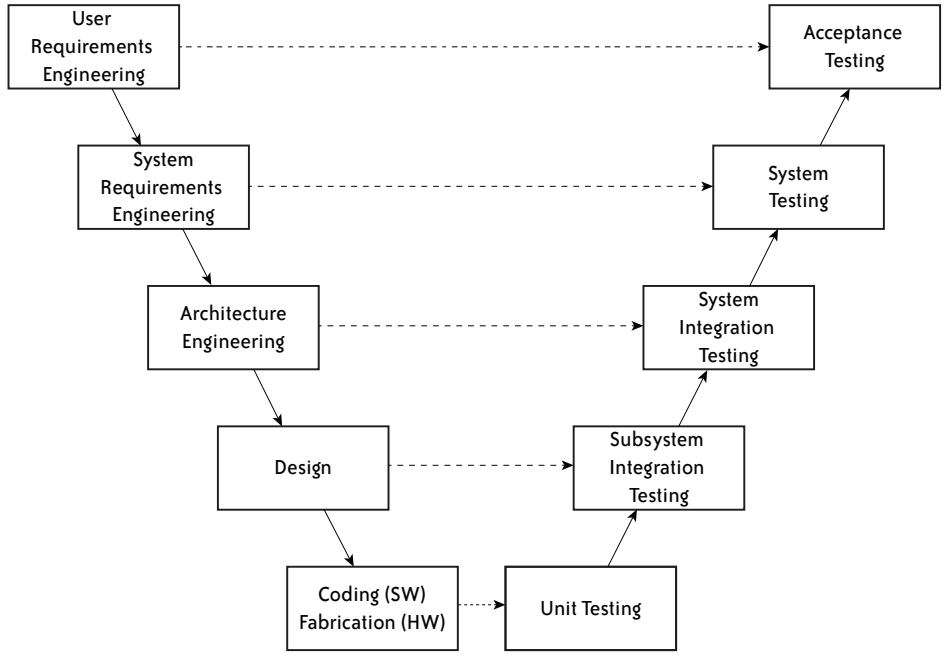


FIGURE 1.1 Traditional Single V model of system engineering activities

1. V stands for both validation and verification.

Figure 1.3 illustrates the Double-V model, which adds the corresponding tests to the Single V Model [Feiler 2012]. The key ideas to take away from this model are:

- Every executable work product should be tested. Testing need not, and in fact should not, be restricted to the implemented system and its parts. It is also important to test any executable requirements, architecture, and design. In this way, associated defects are found and fixed before they can migrate to the actual system and its parts. This typically involves testing executable requirements, architecture, or design models of the system under test (SUT) that are implemented in modeling languages (typically state-based and sufficiently formal) such as SpecTRM-RL, Architecture Analysis and Design Language (AADL), and Program Design Language (PDL); simulations of the SUT; or executable prototypes of the SUT.
- Tests should be created and performed as the corresponding work products are created. The short arrows with two arrowheads are used to show that (1) the executable work products can be developed first and used to drive the

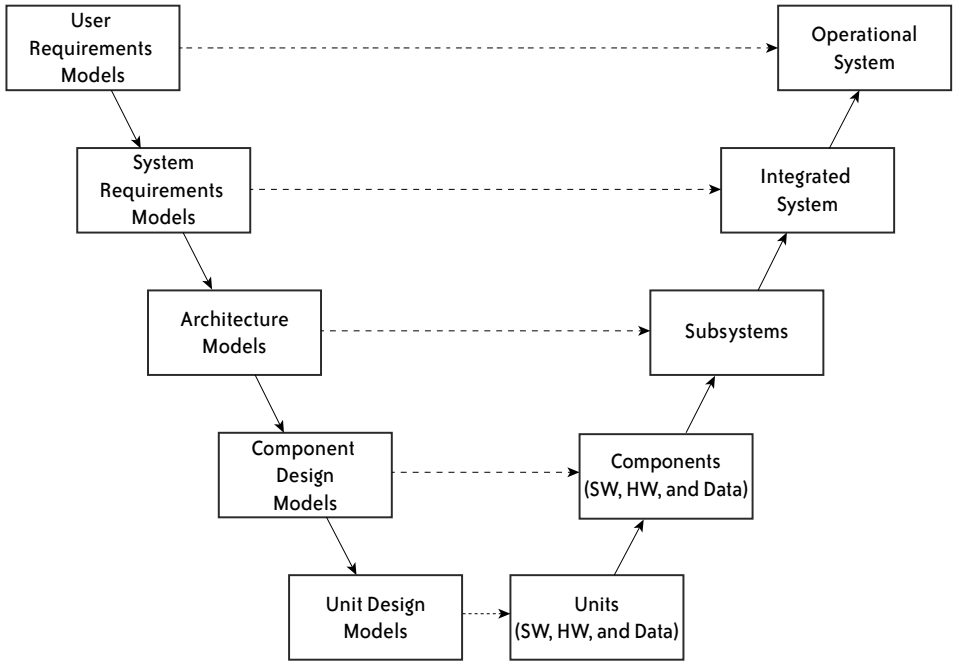


FIGURE 1.2 The Single V model of testable work products

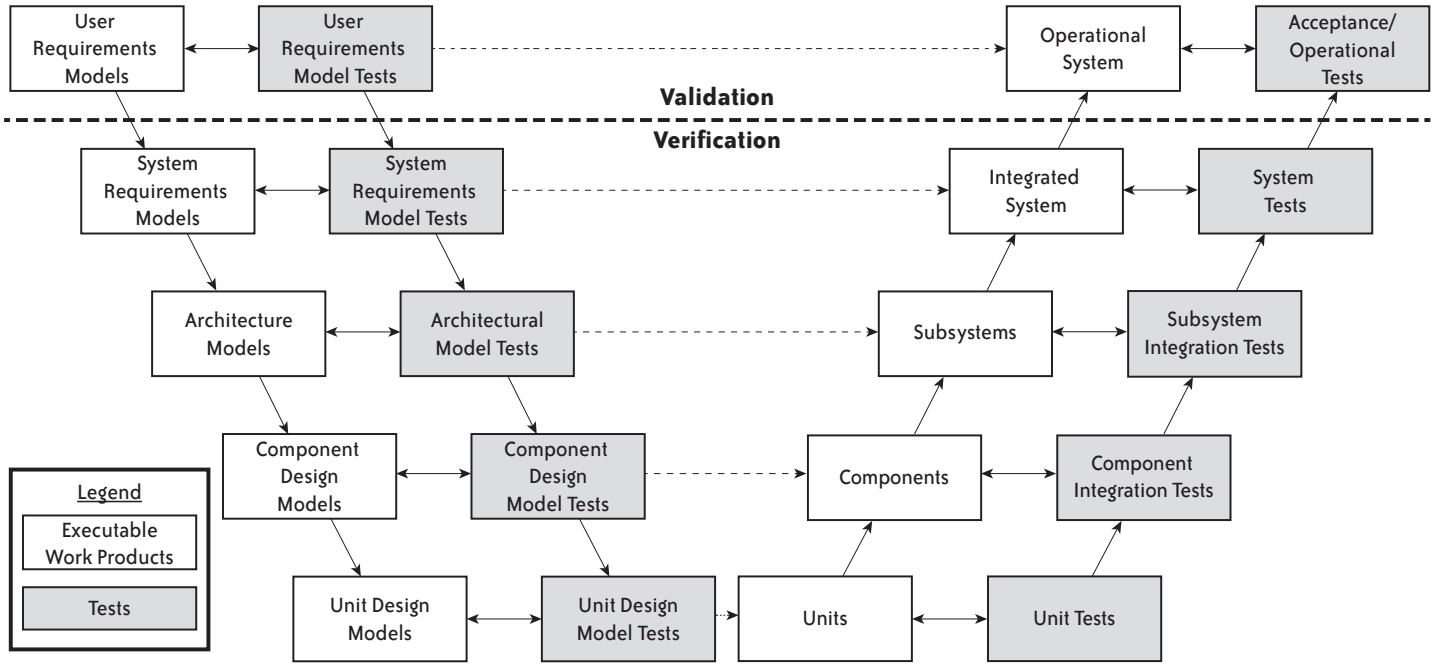


FIGURE 1.3 The Double V model of testable work products and corresponding tests

creation of the tests or (2) Test Driven Development (TDD) can be used, in which case the tests are developed before the work product they test.

- The top row of the model uses testing to *validate* that the system meets the needs of its stakeholders (that is, that the correct system is built). Conversely, the bottom four rows of the model use testing to *verify* that the system is built correctly (that is, architecture conforms to requirements, design conforms to architecture, implementation conforms to design, and so on).
- Finally, in practice, the two sides of the bottom row typically are combined so that the unit design models are incorporated into the units and so that the programming language is used as a program design language (PDL). Similarly, the unit design model tests are incorporated into the unit tests so that the same unit tests verify both the unit design and its implementation.

Figure 1.4 documents the Triple-V model, in which additional verification activities have been added to verify that the testing activities were performed properly. This provides evidence that testing is sufficiently complete and that it will not produce numerous false-positive and false-negative results.

Although the V models appear to show a sequential waterfall development cycle, they also can be used to illustrate an evolutionary (that is, incremental, iterative, and concurrent) development cycle that incorporates many small, potentially overlapping V models. However, when applying a V model to the agile development of a large, complex system, there are some potential complications that require more than a simple collection of small V models, such as:

- The architecturally significant requirements and the associated architecture need to be firmed up as rapidly as is practical because all subsequent increments depend on the architecture, which is difficult and expensive to modify once the initial increment(s) have been based on it.
- Multiple, cross-functional agile teams will be working on different components and subsystems simultaneously, so their increments must be coordinated across teams to produce consistent, testable components and subsystems that can be integrated and released.

Finally, it is interesting to note that these V models are applicable not just to the system under development but also to the development of the system's test environments or test beds and its test laboratories or facilities.

1.3 What Is a Defect?

A system *defect* (informally known as a bug) is a flaw or weakness in the system or one of its components that could cause it to behave in an unintended, unwanted manner or to exhibit an unintended, unwanted property. Defects are related to, but are different from:

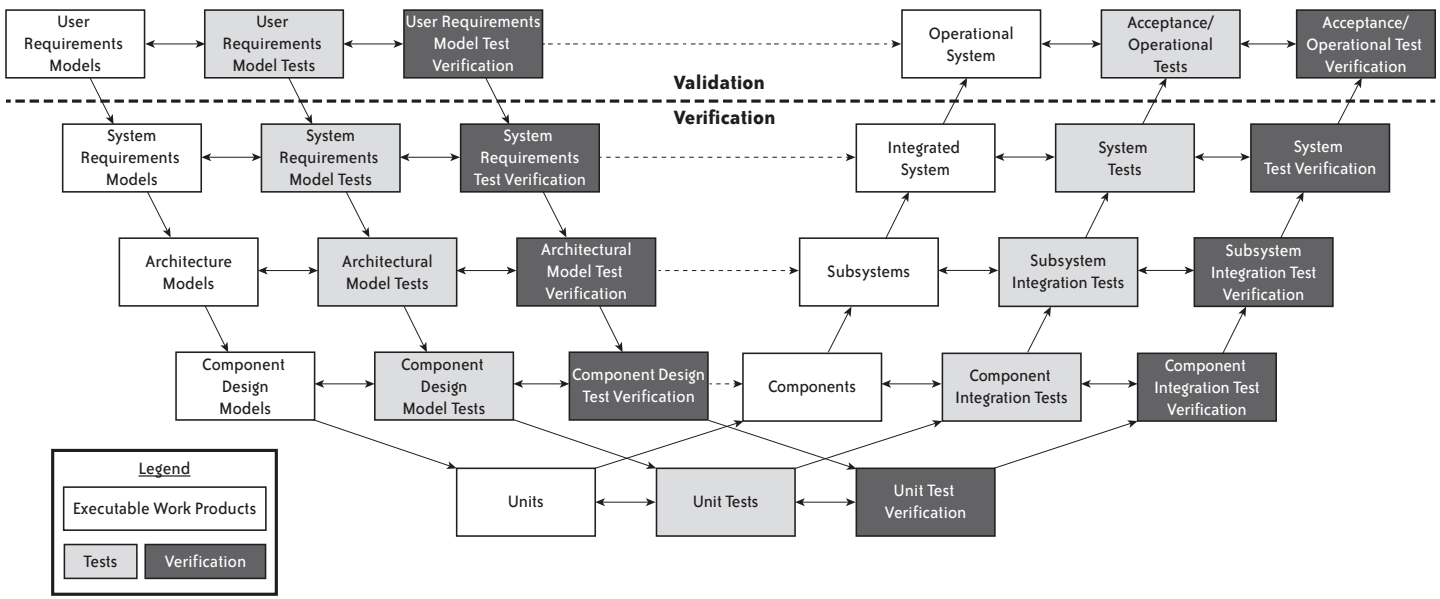


FIGURE 1.4 The Triple V model of work products, tests, and test verification

- **Errors** Human mistakes that cause the defect (for example, making a programming mistake or inputting incorrect data)
- **Faults** Incorrect conditions that are system-internal and not directly visible from outside the system's boundary (for example, the system stores incorrect data or is in an incorrect mode or state)
- **Failures** Events or conditions in which the system visibly behaves incorrectly or has incorrect properties (that is, one or more of its behaviors or properties are different from what its stakeholders can reasonably expect)

Common examples of defects include the following flaws or weaknesses:

- Defects can cause the SUT to violate specified (or unspecified) requirements, including:
  - Functional requirements
  - Data requirements
  - Interface requirements
  - Quality requirements
  - Architecture, design, implementation, and configuration constraints
- Defects can also result when the SUT conforms to incorrect or unnecessary requirements.
- Defects can cause the SUT to:
  - Fail to behave as it should
  - Be missing characteristics that it should have
  - Behave as it should not behave
  - Have characteristics that it should not have
- Defects can cause the SUT to be inconsistent with its architecture or design.
- Defects can result from incorrect or inappropriate architecture, design, implementation, or configuration decisions.
- Defects can violate design guidelines or coding standards.
- Defects can be safety or security vulnerabilities (for example, using inherently unsafe language features or failure to verify input data).

1.4 Why Is Testing Critical?

A National Institute of Standards & Technology (NIST) report [NIST 2002] states that inadequate testing methods and tools cost the US economy between $22.2 billion and $59.5 billion annually, with roughly half of these costs borne by software developers, in the form of extra testing, and half by software users, in the form of failure avoidance and mitigation efforts. The

same study notes that between 25% and 90% of software development budgets are often spent on testing.

Testing is currently the most important of the standard verification and validation methods used during system development and maintenance. This is not because testing is necessarily the most effective and efficient way to verify that the system behaves as it should; it is not. (See Table 1.1, below.) Rather, it is because far more effort, funding, and time are expended on testing than on all other types of verification put together.

According to Capers Jones, most forms of testing find only about 35% of the code defects [Jones 2013b]. Similarly, on average, individual programmers find less than half the defects in their own software.

For example, Capers Jones analyzed data regarding defect identification effectiveness from projects that were completed in early 2013 and produced the results summarized in Table 1.1 [Jones 2013a]. Thus, the use of requirements inspections identified 87% of requirements defects and 25.6% of all defects in the software and its documentation. Similarly, static analysis of the code identified 87% of the code defects and 33.2% of all defects. Finally, a project that used all of these static verification methods identified 95% of all defects.

As Table 1.2 shows, static verification methods are cumulatively more effective at identifying defects except, surprisingly, documentation defects.

TABLE 1.1 Average Percentage of Defects Found as a Function of Static Verification Method and Defect Type

Verification Method	*Defect Type (Location)*					*Total Effectiveness*
	Requirements	*Architecture*	*Design*	*Code*	*Documentation*	
Requirements Inspection	87%	5%	10%	5%	8.5%	25.6%
Architecture Inspection	10%	85%	10%	2.5%	12%	14.9%
Design Inspection	14%	10%	87%	7%	16%	37.3%
Code Inspection	15%	12.5%	20%	85%	10%	70.1%
Static Analysis	2%	2%	7%	87%	3%	33.2%
IV&V	12%	10%	23%	7%	18%	16.5%
SQA Review	17%	10%	17%	12%	12.4%	28.1%
Total	95.2%	92.7%	96.1%	99.1%	58.8%	95.0%

Source: Jones 2013a

TABLE 1.2 Cumulative Effectiveness at Finding Defects by Static Verification Methods, Testing, and Both

Verification Method	Defect Type (Location)					Total Effectiveness
	Requirements	Architecture	Design	Code	Documentation	
Static	95.2%	92.7%	96.1%	99.1%	58.8%	95.0%
Testing	72.3%	74.0%	87.6%	93.4%	95.5%	85.7%
Total	98.11%	98.68%	99.52%	99.94%	98.13%	99.27%

Source: Jones 2013a

1.5 The Limitations of Testing

In spite of its critical nature, testing has a number of pitfalls that make it far less effective and efficient than it should be. Testing is relatively ineffective in the sense that a significant number of residual defects remain in a completed system when it is placed into operation. Testing is also relatively inefficient when you consider the large amount of effort, funding, and time that is currently spent to find defects.

According to Capers Jones, most types of testing find only about 35% of the software defects [Jones 2013]. This is consistent with the following, more detailed analysis of defect detection rates as a function of test type and test capabilities, as shown in Table 1.3 [McConnell 2004].

As Table 1.4 shows, no single type of testing is very effective at uncovering defects, regardless of defect type. Even when all of these testing methods are used on an average project, they still only identify four out of five of the code defects.

TABLE 1.3 Defect Detection Rate

	Defect Detection Rates		
Test Type	Lowest	Mode	Highest
Unit Test	15%	30%	50%
Component Test	20%	30%	35%
Integration Test	25%	35%	40%
System Test	25%	40%	55%
Regression Test	15%	25%	30%
Low-volume Beta Test	25%	35%	40%
High-volume Beta Test	60%	75%	85%

Source: McConnell 2004

TABLE 1.4 Defect Detection Rate

Static Verification	*Project Defect Detection Rate*		
	Worst	*Average*	*Best*
Desk Checking	23%	25%	27%
Static Analysis	0%	55%	55%
Inspection	0%	0%	93%
Static Subtotal	19%	64%	98%

Testing	*Project Defect Detection Rate*		
	Worst	*Average*	*Best*
Unit Test	28%	30%	32%
Function Test	31%	33%	35%
Regression Test	10%	12%	14%
Component Test	28%	30%	32%
Performance Test	6%	10%	14%
System Test	32%	34%	36%
Acceptance Test	13%	15%	17%
Testing Subtotal	72%	81%	87%

Cumulative Total	81.1%	95.6%	99.96%

Source: Jones 2013b

1.6 What Is a Testing Pitfall?

A testing pitfall is any decision, mindset, action, or failure to act that unnecessarily and, potentially unexpectedly, causes testing to be less effective, less efficient, or more frustrating to perform. Basically, a testing pitfall is a commonly occurring way to screw up testing, and projects fall into pitfalls when testers, managers, requirements engineers, and other testing stakeholders make testing-related mistakes that can have unintended negative consequences.

In a sense, the description of a testing pitfall constitutes a testing antipattern. However, the term pitfall was specifically chosen to evoke the image of a hidden or not easily identified trap for the unwary or uninitiated. As with any trap, it is better to avoid a testing pitfall than it is to have to dig one's self and one's project out of it after having fallen in.

1.7 Categorizing Pitfalls

Many testing pitfalls can occur during the development or maintenance of software-reliant systems and software applications. While no project is likely to be so poorly managed and executed as to experience the majority of these pitfalls, most projects will suffer several of them. Similarly, although these testing pitfalls do not guarantee failure, they definitely pose serious risks that need to be managed.

This book documents 92 pitfalls that have been observed to commonly occur during testing. These pitfalls are categorized as follows:

- General Testing Pitfalls
  - Test Planning and Scheduling Pitfalls
  - Stakeholder Involvement and Commitment Pitfalls
  - Management-Related Testing Pitfalls
  - Staffing Pitfalls
  - Test Process Pitfalls
  - Test Tools and Environments Pitfalls
  - Test Communication Pitfalls
  - Requirements-Related Testing Pitfalls
- Test-Type-Specific Pitfalls
  - Unit Testing Pitfalls
  - Integration Testing Pitfalls
  - Specialty Engineering Testing Pitfalls
  - System Testing Pitfalls
  - System of Systems (SoS) Testing Pitfalls
  - Regression Testing Pitfalls

Although each of these testing pitfalls has been observed on multiple projects, it is entirely possible that you might have testing pitfalls that are not addressed by this document. Please notify me of any new testing pitfalls you stumble across or any additional recommended changes to the current pitfalls so that I can incorporate them into future editions of this book.

1.8 Pitfall Specifications

Chapter 2 gives high-level descriptions of the different pitfalls, while Chapter 3 documents each testing pitfall with the following detailed information:

- **Title** A short, descriptive name of the pitfall
- **Description** A brief definition of the pitfall
- **Potential Applicability** The context in which the pitfall may be applicable
- **Characteristic Symptoms (or, How You Will Know)** Symptoms that indicate the possible existence of the pitfall
- **Potential Negative Consequences (Why You Should Care)** Potential negative consequences to expect if the pitfall is not avoided or mitigated[3]
- **Potential Causes** Potential root and proximate causes of the pitfall[4]
- **Recommendations (What You Should Do)** Recommended actions (prepare, enable, perform, and verify) to take to avoid or mitigate the pitfall[5]
- **Related Pitfalls** A list of other related testing pitfalls

A few words on word choice and grammar are probably appropriate before you start reading about the individual pitfalls:

- **Potential Applicability** You may fall into these pitfalls on your project, but then again you may not. Some pitfalls will be more probable and therefore more relevant than others. Of course, if you have already fallen into a given pitfall, it ceases to be potentially applicable and is now absolutely applicable. Because potential applicability currently exists, it is described in the present tense.
- **Characteristic Symptoms** You may have observed these symptoms in the past, and you may well be observing them now. They may even be waiting for you in the future. To save me from having to write all three tenses and, more importantly, to save you from having to read them all, I have listed all symptoms in present tense.
- **Potential Negative Consequences** Once again, you may have suffered these consequences in the past, or they may be happening now. These consequences might still be in the future and avoidable (or subject to mitigation) if you follow the appropriate recommendations now. These consequences are also listed in the present tense.

Note that sometimes the first symptom(s) of a pitfall are the negative consequence(s) you are suffering from because you fell into it. Therefore, it is not always obvious whether something should be listed under symptoms, consequences, or both. To avoid listing the same negative event or situation twice for the same pitfall, I have endeavored to include it only once under the most obvious heading.

- **Potential Causes** Finally, the causes may also lie in your past, your present, or your future. However, they seem to sound best when written in the past tense, for they must by their very nature precede the pitfall's symptoms and consequences.

CHAPTER 2

BRIEF OVERVIEWS OF THE TESTING PITFALLS

This chapter provides a high-level descriptive overview of the testing pitfalls, including the symptoms by which you can recognize them.

2.1 General Testing Pitfalls

These general testing pitfalls are not primarily specific to any single type of testing.

2.1.1 Test Planning and Scheduling Pitfalls

The following pitfalls are related to test planning and scheduling:

1. *No Separate Test Planning Documentation (GEN-TPS-1)*
There is no separate testing-specific planning documentation, only incomplete, high-level overviews of testing in the general planning documents.

2. *Incomplete Test Planning (GEN-TPS-2)*
Test planning and its associated documentation are not sufficiently complete for the current point in the system development cycle.

3. *Test Plans Ignored (GEN-TPS-3)*
The test planning documentation is ignored (that is, it becomes "shelfware") once it is developed and delivered. It is neither used nor maintained.

4. *Test-Case Documents as Test Plans (GEN-TPS-4)*
Test-case documents that document specific test cases are mislabeled as test plans.

5. *Inadequate Test Schedule (GEN-TPS-5)*
The testing schedule is inadequate to complete proper testing.

6. *Testing at the End (GEN-TPS-6)*
All testing is performed late in the development cycle; there is little or no testing of executable models or unit or integration testing planned or performed during the early and middle stages of the development cycle.

2.1.2 Stakeholder Involvement and Commitment Pitfalls

The following pitfalls are related to stakeholder involvement in and commitment to testing:

7. *Wrong Testing Mindset (GEN-SIC-1)*
Some testers and testing stakeholders have an incorrect testing mindset, such as (1) the purpose of testing is to demonstrate that the system works properly rather than to determine where and how it fails, (2) it is the responsibility of testers to verify or "prove" that the system works, (3) the system is assumed to work, and so there is no reason to show that it doesn't work, and (4) testing is viewed as a cost center (that is, an expense) rather than as an investment (or something that can minimize future expenses).

8. *Unrealistic Testing Expectations (GEN-SIC-2)*
Testing stakeholders (especially customer representatives and managers) have unrealistic testing expectations, such as (1) testing detects all (or even the majority of) defects, (2) testing *proves* that there are no remaining defects and that the system therefore works as intended, (3) testing can be, for all practical purposes, exhaustive, (4) testing can be relied on for *all* verification, even though some requirements are better verified via analysis, demonstration, or inspection, and (5) testing (if it is automated) will guarantee the quality of the tests and reduce the testing effort.

9. *Lack of Stakeholder Commitment to Testing (GEN-SIC-3)*
Stakeholder commitment to the testing effort is inadequate; sufficient resources (for example, people, time in the schedule, tools, or funding) are not allocated the testing effort.

2.1.3 Management-Related Testing Pitfalls

The following testing pitfalls are related to management failures:

10. *Inadequate Test Resources (GEN-MGMT-1)*
Management allocates an inadequate amount of resources to testing, including (1) test time in the schedule with inadequate schedule reserves, (2) adequately trained and experienced testers and reviewers, (3) funding, and (4) test tools, test environments (for example, integration test beds and repositories of test data), and test facilities.

11. *Inappropriate External Pressures (GEN-MGMT-2)*
Managers and others in positions of authority subject testers to inappropriate external pressures.

12. *Inadequate Test-Related Risk Management (GEN-MGMT-3)*
There are too few test-related risks identified in the project's official risk repository, and those that are identified have inappropriately low probabilities, low harm severities, and low priorities.

13. *Inadequate Test Metrics (GEN-MGMT-4)*
Too few test-related metrics are being produced, analyzed, reported, and used in decision making.

14. *Inconvenient Test Results Ignored (GEN-MGMT-5)*
Management ignores or treats lightly inconvenient negative test results (especially those with negative ramifications for the schedule, budget, or system quality).

15. *Test Lessons Learned Ignored (GEN-MGMT-6)*
Lessons learned from testing on previous projects are ignored and not placed into practice on the current project.

2.1.4 Staffing Pitfalls

These pitfalls stem from personnel issues in one way or another:

16. *Lack of Independence (GEN-STF-1)*
The test organization or project test team lack adequate technical, managerial, and financial independence to enable them to withstand inappropriate pressure from the development (administrative and technical) management to cut corners.

17. *Unclear Testing Responsibilities (GEN-STF-2)*
The testing responsibilities are unclear and do not adequately address which organizations, teams, and people are going to be responsible for and perform the different types of testing.

18. *Inadequate Testing Expertise (GEN-STF-3)*
Some testers and testing stakeholders have inadequate testing-related understanding, expertise, experience, or training.

19. *Developers Responsible for All Testing (GEN-STF-4)*
There is no separate full-time tester role. Instead, every member of each development team is responsible for testing what he or she designed and implemented.

20. *Testers Responsible for All Testing (GEN-STF-5)*
Testers are responsible for all of the testing during system development. Developers are not even performing unit testing (of either their own software or that of their peers).

2.1.5 Test-Process Pitfalls

These pitfalls are related to the testing process rather than the people performing the testing:

21. *Testing and Engineering Processes Not Integrated (GEN-PRO-1)*
The testing process is not adequately integrated into the overall system engineering process, but is rather treated as a separate specialty engineering activity with only limited interfaces with the primary engineering activities.

22. *One-Size-Fits-All Testing (GEN-PRO-2)*
All testing is performed the same way, to the same level of rigor, regardless of its criticality.

23. *Inadequate Test Prioritization (GEN-PRO-3)*
Testing is not adequately prioritized (for example, all types of testing have the same priority).

24. *Functionality Testing Overemphasized (GEN-PRO-4)*
There is an overemphasis on testing functionality as opposed to testing quality, data, and interface requirements and testing architectural, design, and implementation constraints.

25. *Black-Box System Testing Overemphasized (GEN-PRO-5)*
There is an overemphasis on black-box system testing for requirements conformance, and there is very little white-box unit and integration testing for the architecture, design, and implementation verification.

26. *Black-Box System Testing Underemphasized (GEN-PRO-6)*
There is an overemphasis on white-box unit and integration testing, and very little time is spent on black-box system testing to verify conformance to the requirements.

27. *Too Immature for Testing (GEN-PRO-7)*
Products are delivered for testing when they are immature and not ready to be tested.

28. *Inadequate Evaluations of Test Assets (GEN-PRO-8)*
The quality of the test assets is not adequately evaluated prior to using them.

29. *Inadequate Maintenance of Test Assets (GEN-PRO-9)*
Test assets are not properly maintained (that is, adequately updated and iterated) as defects are found and the system or software under test (SUT) is changed.

30. *Testing as a Phase (GEN-PRO-10)*
Testing is treated as a phase that takes place late in a sequential (also known as waterfall) development cycle instead of as an ongoing activity that takes place continuously in an iterative, incremental, and concurrent (an evolutionary, or agile) development cycle.[6]

31. *Testers Not Involved Early (GEN-PRO-11)*
Testers are not involved at the beginning of the project, but rather only once an implementation exists to test.

32. *Incomplete Testing (GEN-PRO-12)*
The testers inappropriately fail to test certain testable behaviors, characteristics, or components of the system.

33. *No Operational Testing (GEN-PRO-13)*
Representative users are not performing any operational testing of the "completed" system under actual operational conditions.

34. *Inadequate Test Data (GEN-PRO-14)*
The test data (including individual test data and sets of test data) is incomplete or invalid.

35. *Test-Type Confusion (GEN-PRO-15)*
Test cases from one type of testing are redundantly repeated as part of another type of testing, even though the testing types have quite different purposes and scopes.

2.1.6 Test Tools and Environments Pitfalls

These pitfalls are related to the tools and environments used to perform testing:

36. *Over-Reliance on Manual Testing (GEN-TTE-1)*
Testers place too much reliance on manual testing such that the majority of testing is performed manually, without adequate support of test tools or test scripts.

37. *Over-Reliance on Testing Tools (GEN-TTE-2)*
Testers and other testing stakeholders place too much reliance on commercial off-the-shelf (COTS) and homegrown testing tools.

38. *Too Many Target Platforms (GEN-TTE-3)*
The test team and testers are not adequately prepared for testing applications that will execute on numerous target platforms (for example, hardware, operating system, and middleware).

39. *Target Platform Difficult to Access (GEN-TTE-4)*
The testers are not prepared to perform adequate testing when the target platform is not designed to enable access for testing.

40. *Inadequate Test Environments (GEN-TTE-5)*
There are insufficient test tools, test environments or test beds, and test laboratories or facilities, so adequate testing cannot be performed within the schedule and personnel limitations.

41. *Poor Fidelity of Test Environments (GEN-TTE-6)*
The testers build and use test environments or test beds that have poor fidelity to the operational environment of the system or software under test (SUT), and this causes inconclusive or incorrect test results (false-positive and false-negative test results).

42. *Inadequate Test Environment Quality (GEN-TTE-7)*
The quality of one or more test environments is inadequate due to an excessive number of defects.

43. *Test Assets Not Delivered (GEN-TTE-8)*
Developers deliver the system or software to the sustainers without the associated test assets. For example, delivering test assets (such as test plans, test reports, test cases, test oracles, test drivers or scripts, test stubs, and test environments) is neither required nor planned.

44. *Inadequate Test Configuration Management (GEN-TTE-9)*
Testing work products (for example, test cases, test scripts, test data, test tools, and test environments) are not under configuration management (CM).

45. *Developers Ignore Testability (GEN-TTE-10)*
It is unnecessarily difficult to develop automated tests because the developers do not consider testing when designing and implementing their system or software.

2.1.7 Test Communication Pitfalls

These pitfalls are related to poor communication with regard to testing:

46. *Inadequate Architecture or Design Documentation (GEN-COM-1)*[2]
Architects and designers produce insufficient architecture or design documentation (for example, models and documents) to support white-box (structural) unit and integration testing.

47. *Inadequate Defect Reports (GEN-COM-2)*
Testers and others create defect reports (also known as bug and trouble reports) that are incomplete, contain incorrect information, or are difficult to read.

2. Inadequate Requirements Documentation is covered in the next section, which concentrates on requirements.

48. *Inadequate Test Documentation (GEN-COM-3)*[3]
Testers create test documentation that is incomplete or contains incorrect information.

49. *Source Documents Not Maintained (GEN-COM-4)*
Developers do not properly maintain the requirements specifications, architecture documents, design documents, and associated models that are needed as inputs to the development of tests.

50. *Inadequate Communication Concerning Testing (GEN-COM-5)*
There is inadequate verbal and written communication concerning the testing among testers and other testing stakeholders.

2.1.8 Requirements-Related Testing Pitfalls

These pitfalls are related to the negative impact of poor requirements on testing:

51. *Ambiguous Requirements (GEN-REQ-1)*
Testers misinterpret a great many ambiguous requirements and therefore base their testing on incorrect interpretations of these requirements.

52. *Obsolete Requirements (GEN-REQ-2)*
Testers waste effort and time testing whether the system or software under test (SUT) correctly implements a great many obsolete requirements.

53. *Missing Requirements (GEN-REQ-3)*
Testers overlook many undocumented requirements and therefore do not plan for, develop, or run the associated overlooked test cases.

54. *Incomplete Requirements (GEN-REQ-4)*
Testers fail to detect that many requirements are incomplete; therefore, they develop and run correspondingly incomplete or incorrect test cases.

55. *Incorrect Requirements (GEN-REQ-5)*
Testers fail to detect that many requirements are incorrect, and therefore develop and run correspondingly incorrect test cases that produce false-positive and false-negative test results.

56. *Requirements Churn (GEN-REQ-6)*
Testers waste an excessive amount of time and effort developing and running test cases based on many requirements that are not sufficiently stable and that therefore change one or more times prior to delivery.

57. *Improperly Derived Requirements (GEN-REQ-7)*
Testers base their testing on improperly derived requirements, resulting in missing test cases, test cases at the wrong level of abstraction, or incorrect test

3. Incomplete test plans are addressed in Incomplete Test Planning (GEN-TPS-2). This pitfall is more general in that it addresses all testing documents, not just test plans.

cases based on cross cutting requirements that are allocated without modification to multiple architectural components.

58. *Verification Methods Not Properly Specified (GEN-REQ-8)*
Testers (or other developers) fail to properly specify the verification method(s) for each requirement, thereby causing requirements to be verified using unnecessarily inefficient or ineffective verification method(s).

59. *Lack of Requirements Trace (GEN-REQ-9)*
The testers do not trace the requirements to individual tests or test cases, thereby making it unnecessarily difficult to determine whether the tests are inadequate or excessive.

2.2 Test-Type-Specific Pitfalls

The following pitfalls are primarily restricted to a single type of testing:

2.2.1 Unit Testing Pitfalls

These pitfalls are related primarily to testing individual units:

60. *Testing Does Not Drive Design and Implementation (TTS-UNT-1)*
Software developers and testers do not develop their tests first and then use these tests to drive development of the associated architecture, design, and implementation.

61. *Conflict of Interest (TTS-UNT-2)*
Nothing is done to address the following conflict of interest that exists when developers test their own work products: Essentially, they are being asked to demonstrate that their software is defective.

2.2.2 Integration Testing Pitfalls

The following pitfalls are related primarily to integration testing:

62. *Integration Decreases Testability Ignored (TTS-INT-1)*
Testers fail to take into account that integration encapsulates the individual parts of the whole and the interactions between them, thereby making the internal parts of the integrated whole less observable and less controllable and, therefore, less testable.

63. *Inadequate Self-Monitoring (TTS-INT-2)*
Testers are unprepared to address the difficulty of testing encapsulated components due to a lack of system- or software-internal self-tests.

64. *Unavailable Components (TTS-INT-3)*
Integration testing must be postponed due to the unavailability of (1) system hardware or software components or (2) test environment components.

65. *System Testing as Integration Testing (TTS-INT-4)*
Testers are actually performing system-level tests of system functionality when they are supposed to be performing integration testing of component interfaces and interactions.

2.2.3 Specialty Engineering Testing Pitfalls

The following pitfalls are highly similar in nature, although they vary significantly in detail. This section could have been much larger because there are many different quality characteristics and associated attributes, each with its own associated potential symptoms, consequences, and causes.

66. *Inadequate Capacity Testing (TTS-SPC-1)*
Testers perform little or no capacity testing (or the capacity testing they do perform is superficial) to determine the degree to which the system or software degrades gracefully as capacity limits are approached, reached, and exceeded.

67. *Inadequate Concurrency Testing (TTS-SPC-2)*
Testers perform little or no concurrency testing (or the concurrency testing they do perform is superficial) to explicitly uncover the defects that cause the common types of concurrency faults and failures: deadlock, livelock, starvation, priority inversion, race conditions, inconsistent views of shared memory, and unintentional infinite loops.

68. *Inadequate Internationalization Testing (TTS-SPC-3)*
Testers perform little or no internationalization testing—or the internationalization testing they do perform is superficial—to determine the degree to which the system is configurable to perform appropriately in multiple countries.

69. *Inadequate Interoperability Testing (TTS-SPC-4)*
Testers perform little or no interoperability testing (or the interoperability testing they do perform is superficial) to determine the degree to which the system successfully interfaces and collaborates with other systems.

70. *Inadequate Performance Testing (TTS-SPC-5)*
Testers perform little or no performance testing (or the testing they do perform is only superficial) to determine the degree to which the system has adequate levels of the performance quality attributes: event schedulability, jitter, latency, response time, and throughput.

71. *Inadequate Reliability Testing (TTS-SPC-6)*
Testers perform little or no long-duration reliability testing (also known as stability testing)—or the reliability testing they do perform is superficial (for example, it is not done under operational profiles and is not based on the results of any reliability models)—to determine the degree to which the system continues to function over time without failure.

72. *Inadequate Robustness Testing (TTS-SPC-7)*
Testers perform little or no robustness testing, or the robustness testing they do perform is superficial (for example, it is not based on the results of any robustness models), to determine the degree to which the system exhibits adequate error, fault, failure, and environmental tolerance.

73. *Inadequate Safety Testing (TTS-SPC-8)*
Testers perform little or no safety testing, or the safety testing they do perform is superficial (for example, it is not based on the results of a safety or hazard analysis), to determine the degree to which the system is safe from causing or suffering accidental harm.

74. *Inadequate Security Testing (TTS-SPC-9)*
Testers perform little or no security testing—or the security testing they do perform is superficial (for example, it is not based on the results of a security or threat analysis)—to determine the degree to which the system is secure from causing or suffering malicious harm.

75. *Inadequate Usability Testing (TTS-SPC-10)*
Testers or usability engineers perform little or no usability testing—or the usability testing they do perform is superficial—to determine the degree to which the system's human-machine interfaces meet the system's requirements for usability, manpower, personnel, training, human factors engineering (HFE), and habitability.

2.2.4 System Testing Pitfalls

The following pitfalls are related primarily to the testing of completely integrated systems:

76. *Test Hooks Remain (TTS-SYS-1)*
Testers fail to remove temporary test hooks after completing testing, so they remain in the delivered or fielded system.

77. *Lack of Test Hooks (TTS-SYS-2)*
Testers fail to take into account how a lack of test hooks makes it more difficult to test parts of the system hidden via information hiding.

78. *Inadequate End-to-End Testing (TTS-SYS-3)*
Testers perform inadequate system-level functional testing of a system's end-to-end support for its missions.

2.2.5 System of Systems (SoS) Testing Pitfalls

The following pitfalls are related to testing systems of systems:

79. *Inadequate SoS Planning (TTS-SoS-1)*
Testers and SoS architects perform an inadequate amount of SoS test planning and fail to appropriately document their plans in SoS-level test planning documentation.

80. *Unclear SoS Testing Responsibilities (TTS-SoS-2)*
Managers or testers fail to clearly define and document the responsibilities for performing end-to-end SoS testing.

81. *Inadequate Resources for SoS Testing (TTS-SoS-3)*
Management fails to provide adequate resources for system of systems (SoS) testing.

82. *SoS Testing Not Properly Scheduled (TTS-SoS-4)*
System of systems testing is not properly scheduled and coordinated with the individual systems' testing and delivery schedules.

83. *Inadequate SoS Requirements (TTS-SoS-5)*
Many SoS-level requirements are missing, are of poor quality, or are never officially approved or funded.

84. *Inadequate Support from Individual System Projects (TTS-SoS-6)*
Test support from individual system development or maintenance projects is inadequate to perform system of systems testing.

85. *Inadequate Defect Tracking Across Projects (TTS-SoS-7)*
Defect tracking across individual system development or maintenance projects is inadequate to support system of systems testing.

86. *Finger-Pointing (TTS-SoS-8)*
Different system development or maintenance projects assign the responsibility for finding and fixing SoS-level defects to other projects.

2.2.6 Regression Testing Pitfalls

The following pitfalls are related primarily to regression testing:

87. *Inadequate Regression Test Automation (TTS-REG-1)*
Testers and developers have automated an insufficient number of tests to enable adequate regression testing.

88. *Regression Testing Not Performed (TTS-REG-2)*
Testers and maintainers perform insufficient regression testing to determine if new defects have been accidentally introduced when changes are made to the system.

89. *Inadequate Scope of Regression Testing (TTS-REG-3)*
The scope of regression testing is insufficiently broad.

90. *Only Low-Level Regression Tests (TTS-REG-4)*
Only low-level (for example, unit-level and possibly integration) regression tests are rerun, so there is no system, acceptance, or operational regression testing and no SoS regression testing.

91. *Test Resources Not Delivered for Maintenance (TTS-REG-5)*
The test resources produced by the development organization are not made available to the maintenance organization to support testing new capabilities and regression testing changes.

92. *Only Functional Regression Testing (TTS-REG-6)*
Testers and maintainers only perform regression testing to determine if changes introduce functionality-related defects.

Testing at the End (GEN-TPS-6)

Description All testing is performed late in the development cycle.

Potential Applicability This pitfall is potentially applicable anytime that:

- A well-planned and well-documented nontrivial testing program is justified.
- The project has a nontrivial schedule that is sufficiently long to enable testing to be postponed.

Characteristic Symptoms

- Testing is scheduled to be performed late in the development cycle on the Project Master Schedule.
  - There is no testing of executable requirements, architecture, and design models, possibly because no such models were developed.
  - This is essentially testing only on the right side of the Single V Model (pages 2–3) when a sequential waterfall V model is used.
  - This is the opposite of Test Driven Development (TDD).
- Testers are only involved after requirements engineering, architecture engineering, design, and implementation, when all of the defects have already been created. Thus, testing is not used to help prevent any defects or to uncover the defects as they are produced.
- Little or no unit or integration testing is planned or is performed during the early and middle stages of the development cycle.
- There is insufficient time to perform testing during the current incremental, iterative build such that some or all of the testing of the current build is postponed until the following build.

Potential Negative Consequences

- There is insufficient time left in the schedule to correct any major defects found. [12]
- It is difficult to achieve and demonstrate the required degree of test coverage.
- Because so much of the system has been integrated before the beginning of testing, it is very difficult to find and localize defects that remain hidden within the internals of the system.

- Postponing testing from one build to another creates an ever-growing bow wave of testing that can never be performed unless the schedule is radically changed to postpone all work except for testing until testing gets caught up (often a necessary but not politically acceptable action).

Potential Causes

- The project used a strictly interpreted, traditional, sequential waterfall development cycle, whereby testing the requirements, architecture, and design does *not* occur (that is, a strict Single V model was used instead of a Double V model or Triple V model [see pages 5–6]).
- Management was not able to staff the testing team early during the development cycle.
- Management was primarily interested in system testing and did not recognize the need for lower-level (for example, unit and integration) testing.
- There was insufficient time allocated in an incremental, iterative build, both to develop an iterative increment and to adequately test it.

Recommendations

- Prepare:
  - Plan and schedule testing to be performed iteratively, incrementally, and in a parallel manner (that is, use an evolutionary development cycle), starting early during development.
  - Provide training in incremental, iterative testing.
  - Incorporate iterative and incremental testing into the project's system engineering process.
- Enable:
  - Provide adequate testing resources (staffing, tools, budget, and schedule) early during development.
- Perform:
  - Perform testing in an iterative, incremental, and parallel manner starting early during the development cycle.
  - Testers and developers collaborate and work closely together so that new or updated components can be unit and integration tested as soon as is practical.
- Verify:
  - In an ongoing manner (or at the very least, during major project milestones), determine whether testing is being performed iteratively, incrementally, and in parallel with design, implementation, and integration.
  - Use testing metrics to determine the status and ongoing progress of testing.

Related Pitfalls Testing and Engineering Processes Not Integrated (GEN-PRO-1), Testing as a Phase (GEN-PRO-10), Testers Not Involved Early (GEN-PRO-11)

- Determine (for example, via conversation or questioning) whether testing goes beyond "demonstrate that the system works" (sunny-day path testing) to also include "demonstrate that the system does not work" (rainy-day path testing)
- Determine whether the testers exhibit the correct testing mindset.

Related Pitfalls Inappropriate External Pressures (GEN-MGMT-2), Inadequate Communication Concerning Testing (GEN-COM-5)

Unrealistic Testing Expectations (GEN-SIC-2)

Description Testing stakeholders (especially customer representatives and managers) have various unrealistic expectations with regard to testing.

Potential Applicability This pitfall is always potentially applicable.

Characteristic Symptoms

- Testing stakeholders (for example, managers and customer representatives) and some testers falsely believe that:
  - Testing detects all (or even the majority of) defects.[19]
  - Testing *proves* that there are no remaining defects and that the system therefore works as intended.
  - Testing can be, for all practical purposes, exhaustive. (This is false because, for example, testing cannot test all input values under all conditions.)
  - Automated testing can be exhaustive. (It is impractical to automate certain types of tests and exhaustive testing is almost always impossible, regardless of whether the tests are manual or automated.)
  - Automated testing improves or even guarantees the quality of the tests. (It is quite possible to automate poor tests.)
  - Automated testing will always decrease costs. (It may cost more, depending on how rapidly the system under test—especially its user interface—is iterated and how often regression testing must occur.)
  - There is no need to provide additional resources to develop, verify, and maintain the automated test cases.
  - Testing can be relied on for *all* verification, even though some requirements are better verified via analysis, demonstration, certification, and inspection.

Potential Negative Consequences

- Testing stakeholders (for example, customer representatives, managers, developers or testers) have a false sense of security (that is, unjustified and incorrect confidence) that the system will function properly when delivered and deployed.
- Non-testing forms of verification (for example, analysis, demonstration, inspection, and simulation) are not given adequate emphasis, thereby unnecessarily increasing cost and schedule.
- When the system inevitably fails, the testers are more likely to get "blamed" for causing the failure of the unrealistic expectation. Although this may happen even without the unrealistic expectation, the existence of the expectation increases the likelihood and severity of the blame.

Potential Causes

- Managers and other testing stakeholders did not understand that:
  - A passed test could result from a weak or incorrect test rather than from a lack of defects.
  - There are always defects to be revealed. A truly successful or useful test is one that uncovers one or more defects, whereas a passed test proves only that the system worked in that single, specific instance.[20]
  - Test automation requires specialized expertise and needs to be budgeted for the effort required to develop, verify, and maintain the automated tests. Testing stakeholders may get a false sense of security that there are no defects when the system passes all automated tests; these tests could be incomplete, contain incorrect data, or have defects in their scripting.
- Testing stakeholders and testers were not exposed to research results that document the relatively large percentage of residual defects that typically remain after testing.
- Testers and testing stakeholders were not trained in verification approaches (for example, analysis, demonstration, and inspection) other than testing and its relative pros and cons.
- Project testing metrics did not include estimates of residual defects.

Recommendations

- Prepare:
  - Collect information on the limitations of testing.

- Collect information on when and how to augment testing with other types of verification.

- **Enable:**
  - Provide basic training in verification methods, including their associated strengths and limitations.
- **Perform:**
  - Explain the limits of testing to managers, customer representatives, testers, and other testing stakeholders:
    - Testing will not detect all (or even a majority of the) defects.
    - *No* testing is truly exhaustive.
    - Testing cannot *prove* (or demonstrate) that the system works under all combinations of preconditions and trigger events.
    - A passed test could result from a weak test rather than from a lack of defects.
    - A truly successful test is one that finds one or more defects.
  - Do not rely on testing for the verification of all requirements, but rather also incorporate other verification approaches, especially when verifying the architecturally significant quality requirements.
  - Collect, analyze, and report testing metrics that estimate the number of defects remaining after testing.
- **Verify:**
  - Determine whether testing stakeholders understand the limitations of testing.
  - Determine whether testing is the only type of verification being used.
  - Determine whether the number of defects remaining is being estimated and reported.

Related Pitfalls Inappropriate External Pressures (GEN-MGMT-2), Inadequate Communication Concerning Testing (GEN-COM-5), Regression Testing Not Performed (TTS-REG-2)

Lack of Stakeholder Commitment to Testing (GEN-SIC-3)

Description Stakeholder commitment to the testing effort is inadequate.

Potential Applicability This pitfall is always potentially applicable.

Characteristic Symptoms

- Stakeholders ignore the testers and their test results. For example, stakeholders stop:

Poor Fidelity of Test Environments (GEN-TTE-6)

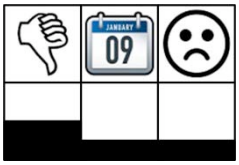
Description The testers build and use test environments or test beds that have poor fidelity to the operational environment of the system or software under test (SUT), and this causes inconclusive or incorrect test results (false-positive and false-negative test results).

Potential Applicability This pitfall is potentially applicable anytime that test environments are being used, especially if they were developed in-house.

Characteristic Symptoms

- Parts of certain test environments poorly emulate or simulate parts of the operational environment:
  - Test tools play the role of system-external actors (for example, human operators or external systems), whereby the behavior of the test tool does not exactly match that of the system-external actors.
  - The test tools communicate with the SUT via a different network than the one that actors will eventually use to communicate with the SUT, such as a public (or private) network substituting for a private (or public) network, or an unclassified network (such as the Internet or Non-classified Internet Protocol Router Network [NIPRNet]) substituting for a classified network (such as the Secure Internet Protocol Router Network [SIPRNet]), whereby the test network's behavior (for example, bandwidth or reliability) does not exactly match that of the actual network.
  - Test drivers or test stubs substitute for the actual clients or servers of the SUT.
  - The test environment contains prototype subsystems, hardware, or software that substitute for actual subsystems, hardware, or software.
  - The test environment contains test software that imperfectly simulates actual software or test hardware that imperfectly emulates actual hardware.
  - The test data (or database) is an imperfect replica of the actual data (or database)[49]:
    - The size of the data or database differs.
    - The type or vendor of the database differs (for example, relational versus object or Oracle versus IBM versus Microsoft)
    - The format of the data differs.
- Cloud-based testing is being used.

Potential Negative Consequences



- The low fidelity of a testing environment causes too many tests to yield inconclusive or incorrect results:
  - Testing produces false-positive test results (that is, the system passes the test even though it will not work in the actual operational environment).
  - Testing produces false-negative test results (that is, the system fails the test even though it will work properly in the actual operational environment).[50]
- It is more difficult to localize and fix defects.
- Test cases need to be repeated when the fidelity problems are solved by:
  - Using a different test environment that better conforms to the operational system and its environment (for example, by replacing software simulation by hardware or by replacing prototype hardware with actual operational hardware)
  - Fixing defects in the test environment

Potential Causes

- **Forms of poor fidelity**. Testing was performed using test environments consisting of components or versions of components that are different from the operational environment(s):
  - **Different software platform**. The software test platform significantly differed from the one(s) that were used to execute the delivered software:
    - Compiler or programming language class library
    - Operating system(s) such as:
      - Android, Apple iOS, Windows (for example, an application that must run on all popular mobile devices), LINUX, UNIX
      - Non-real-time instead of real-time operating system
    - Middleware
    - Database(s)
    - Network software
    - Competing programs and applications (apps)[51]
  - **Different hardware platform**. The hardware test platform significantly differed from the one(s) that were used to execute the delivered software:[52]
    - Processor(s)
    - Memory
    - Motherboard(s)

- Graphic cards
- Network devices (for example, routers and firewalls)
- Disc or tape libraries
- Sensors and actuators
- Battery age[53]

◆ **Different data.** The data stored by the test platform significantly differed from the data that was used with the delivered software:
- Amount of data
- Validity of the data

◆ **Different computing environment:**
- Apps intended for mobile devices were tested using a brand new, out-of-the-box device. However, the apps run on users' devices that contain other apps competing for the device's resources (for example, processing power, memory, and bandwidth) and often have a battery that no longer stores a full charge.
- When using the cloud to provide Testing as a Service (TaaS), the cloud servers had different amounts of utilization by other users and the Internet provided different bandwidth depending on Internet usage.
- The integration environment includes sensors, which were subject to sensor drift (that is, an increase in sensor output error due to a slow degradation of sensor properties).

◆ **Different physical environment.** The test physical environment significantly differed from real-world conditions of the physical environment surrounding the operational system. For example, the actual physical environment has different extremes of:
- Acceleration (whether relatively constant or rapidly changing)
- Ionizing radiation (for example, in outer space, nuclear reactors, or particle accelerators)
- Network connectivity (for example, via radio, microwave, or Wi-Fi) that is intermittent, noisy, does not have sufficient power[54], or is not permitted to be used for security reasons
- Poor electrical power (for example, spikes, surges, sags (or brownouts), blackouts, noise, and EMI)
- Temperature (high, low, or variable)
- Vacuum (for example, of space)
- Vibration

▪ **Causes of poor fidelity.** The lack of adequate test environment fidelity can be due to:

◆ Lack of funding

◆ Inadequate tester expertise

- Difficulty in recreating the potentially huge number of device-internal configurations (for example, existence of unrelated software and data running on the same platform as the SUT[55])
- Poor configuration management of the hardware or software
- Lack of availability of the correct software, hardware, or data
- Lack of availability of sufficient computing resources to match the production system
- Resource contention between SUT and other software executing on the same platform
- The high cost of replicating the physical environment
- Prohibited access to classified networks, such as the US Department of Defense Global Information Grid (GIG) and, more specifically, the SIPRNet, for testing systems or software that has not yet been accredited and certified as being sufficiently secure to operate.[56]

Recommendations

- **Prepare:**
  - Determine how the testers are going to address test-environment fidelity.
  - Document how the testers are going to address test-environment fidelity in the test planning documentation.
  - For testing mobile devices, consider using a commercial company that specializes in testing:
    - On many devices
    - Networks
- **Enable:**
  - Provide the test labs with sufficient numbers of COTS, prototype, or Low Rate of Initial Production (LRIP) system components (subsystems, software, and hardware).
  - Provide sufficient funding to recreate the physical operational environment in the physical test environment(s). Test using a shielded room but realize that operational testing (OT) will still need to be performed.
  - Provide good configuration management of components under test and test environments.
  - Provide tools to evaluate the fidelity of the test environment's behavior.
  - Evaluate commercial companies that specialize in on-device testing and network simulation and testing.
- **Perform:**
  - To the extent practical, use the operational versions of development tools (for example, the compiler and software class libraries).

- To the extent practical, test the software executing on the actual operational platforms, including software, hardware, and data.
- Test on used hardware as well as new, just out-of-the-box devices.
- To the extent practical, test the system in the operational physical environment.
- Perform operationally relevant testing using the actual operational software, hardware, data, and physical environments.[57]
- Recalibrate sensors to compensate for **sensor drift.**

- Verify:
  - To the extent practical, determine whether test simulators, emulators, stubs, and drivers have the same characteristics as the eventual components they are replacing during testing.

Related Pitfalls Inadequate Test Environment Quality (GEN-TTE-7), Unavailable Components (TTS-INT-3)

Inadequate Test Environment Quality (GEN-TTE-7)

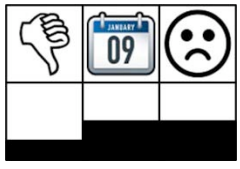
Description The quality of one or more test environments is inadequate due to an excessive number of defects.[58]

Potential Applicability This pitfall is potentially applicable anytime that one or more test environments are being used.

Characteristic Symptoms

- An excessive number of false-positive or false-negative test results are traced back to the poor quality of one or more test environments.
- The test environment often crashes during testing.
- Subsystem or hardware components of an integration test environment are not certified (for example, flight certified, safety certified, security certified) even though the actual subsystem or hardware components must be certified prior to incorporating them into a deliverable system.

Potential Negative Consequences



- There are numerous false-positive or false-negative test results.

Conflict of Interest (TTS-UNT-2)

Description Nothing is done to address the following conflict of interest that exists when software developers test their own work products: They are being asked to demonstrate that their own software is defective.

Potential Applicability This pitfall is potentially applicable anytime that a developer tests his or her own software, which is a very common industry practice. This pitfall is primarily applicable to software unit testing, but it also applies when:

- Requirements engineers test their own executable requirements models.
- Architects test their own executable architectural models.
- Designers test their own executable design models.

Characteristic Symptoms

- Software developers unit test the same units that they personally developed.
- Software developers and managers think that unit testing is not sufficiently important to require that professional testers perform it.
- Software developers spend far less time testing their software than developing it.
- There are few software unit-level test cases.
- The test cases concentrate heavily on demonstrating "sunny-day" paths and largely ignore verifying that "rainy-day" exceptional paths work properly.

Potential Negative Consequences

- Unit testing is poorly and incompletely performed.[86]
- Unit test cases are poorly maintained, in spite of their value for regression testing.
- An unacceptably large number of defects that should have been found during unit testing pass through to integration and system testing, which are thereby slowed down and made less efficient.

Potential Causes

- Developers tested the units that they personally developed.
- Developers expected their software to work correctly (an incorrect mindset), so:

    - They tried to demonstrate that it works rather than show that it doesn't work.
    - They developed as few test cases as practical.
- Developers felt that the testers would catch any defects they missed.[87]
- Developers thought that it was far more fun to write software than to test software.
- Managers or developers thought that unit testing is relatively unimportant, especially in relation to actually developing the software.

Recommendations

- **Prepare:**
    - Establish clear software unit testing success criteria that must be passed before the unit can be delivered for integration and integration testing.
- **Enable:**
    - Provide software developers with training in how to:
        - Perform unit testing
        - Be aware of and counteract their conflict of interest
    - Provide developers with tools to help with automated unit testing.
    - Ensure that the developers understand the importance of finding highly localized defects during unit testing, when they are much easier to localize, analyze, and fix.
- **Perform:**
    - Have units tested by peers of the software developers who produced them.
    - Institute pair programming.
    - Require that the software developers institute unit-level Test Driven Development (TDD).
    - Incentivize software developers to do a better job of testing their own software.
- **Verify:**
    - Determine whether the software developers are clear about their testing responsibilities.
    - Determine whether sufficient unit testing is taking place.
    - Determine (for example, via observation and conversation) whether the software developers are truly trying to identify defects (that is, break their own software).

Related Pitfalls Wrong Testing Mindset (GEN-SIC-1), Unclear Testing Responsibilities (GEN-STF-2), Testing Does *Not* Drive Design and Implementation (TTS-UNT-1)

- There is no SoS change control board (CCB) to officially mandate which system(s) need to be fixed.

Recommendations

- Prepare:
  - Address fixing SoS-level defects in the individual system development projects' planning documents.
- Enable:
  - Set up an SoS CCB if one does not already exist.
  - Grant the SoS CCB the authority to allocate defects to system-level projects for fixing.
  - Work to develop an SoS mindset among the members of the individual system development projects.
- Perform:
  - Assign representatives of the individual system projects to the SoS CCB and involve them in SoS defect allocation.
- Verify:
  - Determine whether an SoS CCB exists and has adequate authority to allocate defects to individual systems.

Related Pitfalls Inadequate Support from Individual System Projects (TTS-SoS-6)

3.4.6 Regression Testing Pitfalls

The following pitfalls are specific to performing regression testing, including testing during maintenance:

- Inadequate Regression Test Automation (TTS-REG-1)
- Regression Testing Not Performed (TTS-REG-2)
- Inadequate Scope of Regression Testing (TTS-REG-3)
- Only Low-Level Regression Tests (TTS-REG-4)
- Test Resources Not Delivered for Maintenance (TTS-REG-5)
- Only Functional Regression Testing (TTS-REG-6)

Inadequate Regression Test Automation (TTS-REG-1)

Description Testers and developers have automated an insufficient number of tests to enable adequate regression testing.[104]

Potential Applicability This pitfall is potentially applicable anytime regression testing is needed (that is, almost always).

Characteristic Symptoms

- Many or even most of the tests are being performed manually.

Potential Negative Consequences

- Manual regression testing takes so much time and effort that it is not done.
- If performed, regression testing is rushed, incomplete, and inadequate to uncover a sufficient number of defects.
- Testers make an excessive number of mistakes while manually performing the tests.
- Defects introduced while making changes in previously tested subsystems or software remain in the operational system.
- The lack of adequate test automation prevents the use of an agile evolutionary (iterative and incremental) development cycle.

Potential Causes

- Testing stakeholders (for example, managers and the developers of unit tests):
  - Mistakenly believed that regression testing is neither necessary nor cost effective because:
    - Most changes are minor in scope.
    - System testing will catch any inadvertently introduced integration defects.
    - They are overconfident that their changes have not introduced any new defects.
  - Were not aware of the:
    - Importance of regression testing
    - Value of automating regression testing
    - Dependence of agile evolutionary development processes on test automation
- Automated regression testing was not an explicit part of the testing process.
- Automated regression testing was not incorporated into the test planning documentation.
- The schedule contained little or no time for developing and maintaining automated tests.
- Tool support for automated regression testing was lacking (for example, due to insufficient test budget) or impractical to use.

- The initially developed automated tests were not maintained.
- The initially developed automated tests were not delivered with the system.
- The system was locked down (for example, Apple iPad and iPhone), thereby making it difficult to perform automated installation and testing.

Recommendations

- **Prepare:**
  - Explicitly address automated regression testing in the project's:
    - Test planning documentation
    - Test process documentation (for example, procedures and guidelines)
    - Master schedule
    - Work Breakdown Structure (WBS)
- **Enable:**
  - Provide training or mentoring to the testing stakeholders in the importance and value of automated regression testing.
  - Provide sufficient time in the schedule for automating and maintaining the tests.
  - Provide sufficient funding to pay for tools that support test automation.
  - Ensure that adequate resources (staffing, budget, and schedule) are planned and available for automating and maintaining the tests.
- **Perform:**
  - Have testers and developers collaborate on automating regression testing whereby each plays the role for which they have adequate expertise and experience:[105]
    - Testers determine types of regression testing; test-case-selection criteria; test cases, including test preconditions, inputs, postconditions, and outputs; test-completion criteria, and so on.
    - Developers create automated regression tests, including configuring the test automation tools, programming in the test cases, writing test scripts, and whatever else is necessary.
  - Automate as many of the regression tests as is practical.
  - Make running the regression tests as easy as is practical so that they can be run frequently (for example, every night).
  - Where appropriate, use commercially available test tools to automate testing.
  - Ensure that both automated and manual test results are integrated into the same overall test results database so that test reporting and monitoring are seamless.
  - Maintain the automated tests as the system changes.
  - Deliver the automated tests with the tested system.

- Verify:
  - Determine whether the test planning documentation, test process documentation, and WBS adequately address automated regression testing.
  - Determine whether the schedule provides sufficient time to automate and maintain the tests.
  - Determine whether a sufficient number of the tests have been automated.
  - Determine whether the automated tests function properly.
  - Determine whether the automated tests are properly maintained.
  - Determine whether the automated tests are delivered with the system.

Related Pitfalls No Separate Test Planning Documentation (GEN-TPS-1), Incomplete Test Planning (GEN-TPS-2), Inadequate Test Schedule (GEN-TPS-5), Unrealistic Testing Expectations (GEN-SIC-2), Inadequate Test Resources (GEN-MGMT-1), Inadequate Maintenance of Test Assets (GEN-PRO-9), Over-Reliance on Manual Testing (GEN-TTE-1), Test Assets Not Delivered (GEN-TTE-8), Inadequate Test Configuration Management (GEN-TTE-9)

Regression Testing Not Performed (TTS-REG-2)

Description Testers and maintainers perform insufficient regression testing to determine if new defects have been accidentally introduced when changes are made to the system.[106]

Potential Applicability This pitfall is potentially applicable anytime regression testing is needed (that is, almost always).

Characteristic Symptoms

- No regression testing is being performed.
- Parts of the system are not retested after they are changed (for example, additions, modifications, and deletions due to refactoring and defect fixes).
- Appropriate parts of the system are not retested after interfacing parts are changed.
- Previously tested software is being reused without modification.
- Defects trace to previously tested changed components and components interfacing with changed components.