Common Testing Problems Checklists: Symptoms and Recommendations

This set of checklists identifies commonly occurring testing problems, organized into sets of similar problems. Although these problems have been observed on multiple projects, there is no guarantee that the set is exhaustive and that you will not have testing problems not addressed by this document.

This document is divided into two main sections. The first provides nine checklists, each of which addresses a set of similar problems and which provides a set of symptoms of the specific problem. The second section provides nine corresponding tables, each of which lists recommendations to help solve the corresponding problems.

The goal of testing is not to prove that something works but rather to demonstrate that it does not. A good tester assumes that there are always defects (an extremely safe assumption), and it is the tester's responsibility to uncover them. Thus, a good test is one that causes the thing being tested to fail so that the underlying defect(s) can be found and fixed. Given that testers are looking for problems, it thus seems fitting that these testing checklists are designed to help find testing problems rather than to show that no such testing problems exist. These checklists are therefore written in terms of problems, and a "yes" signifies that a potential problem has been found, not that the absence of a problem has been shown. Nevertheless, a "yes" should not be viewed negatively but rather as a "positive" result in the following sense: a previously unknown problem is no longer unknown and can therefore be fixed. Surely, that is a positive step forward.

1 Checklists

The following checklists are used to identify commonly occurring testing problems. These checklists can be used during the development and review of test plans and test process documents as well the testing sections of system engineering management plans (SEMPs) and software development plans (SDPs). These checklists can also be used during the oversight and evaluation of the actual testing.

Given that these checklists are used to identify testing problems, a *yes* observation is bad (symptoms indicating the probable existence of a problem) whereas a *no* observation is good (no symptom of a potential problem found). The checklist results are intended to be used to help identify possible testing risks and thus the probable need to fix the specific problems found. These results are not intended for use as input to some quantitative scoring scheme.

1.1 General Testing Problems

#	Testing Problems	Symptoms of Testing Problems	Y/N
1.1	**Wrong Test Mindset / "Happy Path Testing"**	Testing being used to prove that system/software works rather than to show where and how it fails.[1] Only nominal ("sunny day") behavior rather than off-nominal behavior is being tested. Test input only includes middle of the road values rather than boundary values and corner cases.	
1.2	**Defects Discovered Late**	Large numbers of requirements, architecture, and design defects are being found that should have been discovered (during reviews) and fixed prior to current testing. Defects that should have been discovered during lower-level testing are slipping through until higher-level testing or even after delivery.	
1.3	**Inadequate Testing Expertise**	Contractor testers or Government representatives have inadequate testing expertise. Little or no training in testing has taken place.	

[1] This is especially a problem with unit testing and with small cross-functional or agile teams where developers test their own software.

1.4	**Inadequate Schedule**	Testing is inadequately incomplete because there is insufficient time allocated in the schedule for all appropriate tests to be performed.[2] For example, an agile (i.e., iterative, incremental, and concurrent) development/life cycle greatly increases the amount of regression testing needed.	
1.5	**Inadequate Test Metrics**	There are insufficient test metrics being produced, analyzed, and reported. The primary test metrics (e.g., number of tests passed, number of defects found) show neither the productivity of the testers nor their effectiveness at finding defects (e.g., defects found per test or per day). The number of latent *undiscovered* defects remaining is not estimated (e.g., using COQUALMO).	
1.6	**Inadequate Test-related Risk Management**	There are *no* test-related risks identified in the project's official risk repository. The number of test-related risks is unrealistically low. The identified test-related risks have inappropriately low probabilities, harm severities, and priorities.	
1.7	**Unrealistic Expectations**	Non-testers (e.g., managers and customer representatives) falsely believe that (1) testing detects all (or the majority of) defects,[3] (2) testing can be effectively exhaustive, and (3) testing *proves* that the system works. Testing is being relied upon for *all* verification.	
1.8	**False Sense of Security**	Non-testers expect that passing testing *proves* that there are no remaining defects. Non-testers do *not* realize that a passed test could result from a weak test rather than a lack of defects. Non-testers do *not* understand that a truly successful test is one that finds one or more defects.	
1.9	**Over-reliance on COTS Testing Tools**	Testers place too much reliance on testing tools and the automation of test case creation. Testers are relying on the tool as their test oracle (to determine the correct test result). Testers let the tool drive the test methodology rather than the other way around. Testers are using the tool to automate test case selection and completion ("coverage") criteria.	
1.10	**Inadequate Documentation**	Testing assets (e.g., test documents, environments, and test cases) are not sufficiently documented to be useful during maintenance and to be reusable (e.g., within product	

[2] Although testing is never exhaustive, more time is typically needed for adequate testing unless testing can be made more efficient. For example, fewer defects could be produced and these defects can be found and fixed earlier and thereby be prevented from reaching the current testing.

[3] Testing typically finds less than half of all latent defects and is not the most efficient way of detecting many defects.

		lines).	
1.11	**Inadequate Maintenance**	Testing assets (e.g., test software and documents such as test cases, test procedures, test drivers, and test stubs) are not being adequately maintained as defects are found and system changes are introduced (e.g., due to refactoring). The testing assets are no longer consistent with the current requirements, architecture, design, and implementation. Regression test assets are not updated when an agile development cycle with numerous increments is used.	
1.12	**Inadequate Prioritization**	All types of testing are given the same priority. All test cases for the system or a subsystem are given the same priority. The most important tests of a given type are not being performed first. Difficult but important testing is postponed until late in the schedule.	
1.13	**Inadequate Configuration Management (CM)**	Test plans, procedures, test cases, and other testing work products are *not* being placed under configuration control.	
1.14	**Lack of Review**	No [peer-level] review of the test assets (e.g., test inputs, preconditions (pre-test state), and test oracle including expected test outputs and postconditions) is being performed prior to actual testing.	
1.15	**Poor Communication**	There is inadequate testing-related communication between: • Teams within large or geographically-distributed programs • Contractually separated teams (prime vs. subcontractor, system of systems) • Between testers and: — Other developers (requirements engineers, architects, designers, and implementers) — Other testers — Customers, user representatives, and subject matter experts (SMEs)	
1.16	**External Influences**	Managers or developers are dictating to the testers what constitutes a bug or a defect *worth reporting*. Oppose any pressure to not find defects just prior to delivery.	

1.2 Test Planning Problems

#	Testing Problems	Symptoms of Testing Problems	Y/N
2.1	**No Separate Test Plan**	There is *no* separate Test and Evaluation Master Plan (TEMP) or Software Test Plan (STP). There are only incomplete high-level overviews of testing in System Engineering Master Plans (SEMPs) and Software Development Plans (SDPs).	
2.2	**Incomplete Test Planning**	The test planning documents *lack* clear and specific: • test objectives • testing methods and techniques (e.g., testing is ad hoc, and planning documents merely list the different types of testing rather than state how the testing will be performed) • test case selection criteria (e.g., single nominal test case vs. boundary value testing) • test completion (e.g., "coverage") criteria (e.g., does it include both nominal and off-nominal testing)	
2.3	**One-Size-Fits-All Test Planning**	The test planning documents contain only generic boilerplate rather than appropriate system-specific information. Are mission-, safety-, and security-critical software are not required to be tested more completely and rigorously than other less-critical software?	
2.4	**Inadequate Resources Planned**	The test planning documents and schedules *fail* to provide for adequate test resources such as: • test time in schedule with inadequate schedule reserves • trained and experienced testers and reviewers • funding • test tools and environments (e.g., integration test beds)	

1.3 Requirements-Related Problems

#	Testing Problems	Symptoms of Testing Problems	Y/N
3.1	**Ambiguous Requirements**	Testers are misinterpreting requirements that are ambiguous due to the use of: • inherently ambiguous words • undefined technical terms and acronyms	

		• quantities without associated units of measure • synonyms	
3.2	**Missing Requirements**	Testing is incomplete (e.g., missing test cases) because of missing requirements: • Use case analysis primarily addressed normal (sunny day) paths as opposed to fault tolerant and failure (rainy day) paths. • Requirements for off-nominal behavior (e.g., fault and failure detection and reaction) are missing. • Quality requirements (e.g., availability, interoperability, maintainability, performance, portability, reliability, robustness, safety, security, and usability) are missing. • Data requirements are missing.	
3.3	**Incomplete Requirements**	Testing is incomplete or incorrect due to incomplete requirements that lack: • Preconditions and trigger events • Quantitative thresholds • Postconditions	
3.3	**Requirements Lack Good Characteristics**	Testing is difficult or impossible because many requirements lack the characteristics of good requirements such as being complete, consistent, correct, feasible, mandatory, testable and unambiguous.	
3.4	**Unstable Requirements**	Test cases (test inputs, preconditions, and expected test outputs) and automated regression tests are being obsoleted because the requirements are constantly changing.	
3.5	**Poorly Derived Requirements**	Testing is difficult because derived requirements merely restate their parent requirement and newly allocated requirements are not at the proper level of abstraction.	

1.4 Unit Testing Problems

#	Testing Problems	Symptoms of Testing Problems	Y/N

4.1	**Unstable Design**	Test cases must be constantly updated and test hooks are lost due to design changes (e.g., refactoring and new capabilities).[4]	
4.2	**Inadequate Detailed Design**	Unit testing is difficult to perform and repeat because there is insufficient design detail to drive the testing.	
4.3	**Poor Fidelity of Test Environment**	Unit testing is experiencing many false positives because it is being performed using a: • different compiler [version] than the delivered code • software test environment with poor hardware simulation	

1.5 Integration Testing Problems

#	Testing Problems	Symptoms of Testing Problems	Y/N
5.1	**Defect Localization**	It is difficult to determine the location of the defect. Is the defect in the new or updated operational software under test, the operational hardware under test, in the COTS OS and middleware, in the software test bed (e.g., in software simulations of hardware), in the hardware test beds (e.g., in pre-production hardware), in the tests themselves (e.g., in the test inputs, preconditions, expected outputs, and expected postconditions), or in a configuration/version mismatch among them?	
5.2	**Insufficient Test Environments**	There are an insufficient number of test environments. There is an excessive amount of competition between and among the integration testers and other testers for time on the test environment.	
5.3	**Schedule Conflicts**	Too many test teams are sharing the same test beds. It is difficult to optimally schedule the allocation of test teams to test beds. Too much time is wasted reconfiguring the test bed for the next team's use.	
5.4	**Unavailable Components**	The operational software, simulation software, test hardware, and actual hardware components are *not* available for integration into the test environments prior to scheduled integration testing.	
5.5	**Poor Test Bed Quality**	The test environments contain excessive numbers of defects making it difficult to	

[4] This is especially true with agile development cycles with many short-duration increments and with projects where off-nominal behavior is postponed until late increments.

		schedule and perform testing.	
5.6	**Inadequate Self-Test**	Failures are difficult to cause, reproduce, and localize because the operational software or subsystem does not contain sufficient test hooks, built-in-test (BIT), or prognostics and health management (PHM) software.	

1.6 Specialty Engineering Testing Problems

#	Testing Problems	Symptoms of Testing Problems	Y/N
6.1	**Inadequate Capacity Testing**	There is little or no testing to determine performance as capacity limits are approached, reached, and exceeded.	
6.2	**Inadequate Reliability Testing**	There is little or no long duration testing under operational profiles to estimate the system's reliability.	
6.3	**Inadequate Robustness Testing**	The testing is not based on robustness analysis such as off-nominal (i.e., fault, degraded mode, and failure) use case paths, Event Tree Analysis (ETA), Fault Tree Analysis (FTA), or Failure Modes Effects Criticality Analysis (FMECA). There is little or no testing *Robustness Testing*: • *Error Tolerance Testing*, the goal of which is to show that system does not detect or react properly to input errors (a subtype of which is *Fuzz Testing*) • *Fault Tolerance Testing*, the goal of which is to show that system does not detect or react properly to system faults (bad states) • *Failure Tolerance Testing*, the goal of which is to show that system does not detect or react properly to system failures • *Environmental Tolerance Testing*, the goal of which is to show that system does not detect or react properly to dangerous environmental conditions	
6.4	**Inadequate Safety Testing**	There is little or no: • testing based on safety analysis (e.g., abuse/mishap cases, ETA, or FTA) • testing of safeguards (e.g., interlocks) • fail-safe behavior • safety-specific testing: — *Vulnerability Testing*, the goal of which is to expose a system vulnerability (i.e.,	

		defect or weakness) — *Hazard Testing*, the goal of which is to make the system cause a hazard to come into existence — *Mishap Testing*, the goal of which is to make the system cause an accident or near miss	
6.5	**Inadequate Security Testing**	There is little or no: • testing based on security analysis (e.g., attack trees or abuse/misuse cases) • testing of security controls (e.g., access control, encryption/decryption, or intrusion detection) • fail-secure behavior • security-specific testing: — *Penetration Testing*, the goal of which is to penetrate the systems defenses — *Fuzz Testing*, the goal of which is to cause the system to fail due to random input — *Vulnerability Testing*, the goal of which is to expose a system vulnerability (i.e., defect or weakness)	
6.6	**Inadequate Usability Testing**	There is little or no explicit usability testing of human interfaces for user friendliness, learnability, etc.	

1.7 System Testing Problems

#	Testing Problems	Symptoms of Testing Problems	Y/N
7.1	**Testing Fault Tolerance is Difficult**	It is difficult for tests of the integrated system to cause local faults (i.e., internal to a subsystem) in order to test for fault tolerance.	
7.2	**Testing Code Coverage is Difficult**	It is difficult for tests of the integrated system to demonstrate code coverage, which is very important for safety and security.	
7.3	**Lack of Test Hooks**	It is difficult to test locally implemented requirements because internal test hooks and testing software has been removed.	

1.8 System of System (SoS) Testing Problems

#	Testing Problems	Symptoms of Testing Problems	Y/N
8.1	**Inadequate SoS Planning**	Little or no planning has occurred for testing above the individual system level. There are no clear test completion/acceptance criteria at the system of systems level.	
8.2	**Poor or Missing SoS Requirements**	Requirements-based testing is difficult because little or no requirements exist above the system level so that there are no official approved SoS requirements to verify.	
8.3	**Unclear Testing Responsibilities**	No project is explicitly tasked with testing end-to-end SoS behavior.	
8.4	**Testing not Funded**	No program is funded to perform end-to-end SoS testing.	
8.5	**Testing not Properly Scheduled**	SoS testing is not in the individual systems' integrated master schedules, and there is no SoS master schedule. SoS testing must be fit into the uncoordinated schedules of the individual systems.	
8.6	**Inadequate Test Support from Individual Systems**	It is difficult to obtain the necessary test resources (e.g., people and test beds) from individual projects because they are already committed and there is no funding for such support.	
8.7	**Poor Defect Tracking Across Projects**	There is little or no coordination of defect tracking and associated regression testing across multiple projects.	
8.8	**Finger-Pointing**	There is a significant amount of finger pointing across project boundaries regarding where defects lie (i.e., in which systems and in which project's testing).	

1.9 Regression/Maintenance Testing Problems

#	Testing Problems	Symptoms of Testing Problems	Y/N
9.1	**Insufficient Automation**	Testing is not sufficiently automated to decrease the testing effort, especially when an agile (iterative, incremental, and parallel) development cycle results in large numbers of increments that must be retested. Regression testing takes so much time and effort that it is rarely done.	
9.2	**Regression Tests Not**	Regression testing is not being done because:	

	Rerun	• There is insufficient time and staffing to perform it. • Managers or developers do not believe that it is necessary because of the minor scope of most changes. • There is insufficient automation of regression tests.	
9.3	**Inadequate Scope of Regression Testing**	Only test the changed code because the "change can't effect the rest of the system."	
9.4	**Only Low-Level Regression Tests**	Only unit tests and some integration tests are rerun. System and/or the SoS tests are not rerun	
9.5	**Disagreement over Funding**	There is disagreement as to whether the budget for testing during maintenance should come from development or sustainment funding.	

2 Recommended Solutions

The following list and tables recommend solutions to the commonly occurring testing problems found in the preceding tables.

2.1 General Solutions

The following are general solutions applicable to most of the common testing problems:

- **Prevention Solutions:**
  - **Formally require the solutions** – Acquirers formally require the solutions to the testing problems in the appropriate documentation such as the *Request for Proposals* and *Contract*.
  - **Mandate the solutions** – Managers, chief engineers (development team leaders), or chief testers (test team leaders) explicitly mandate the solutions to the testing problems in the appropriate documentation such as the *System Engineering Management Plan (SEMP)*, *System/Software Development Plan (SDP)*, *Test Plan*, and/or *Test Strategy*).
  - **Provide training** – Chief testers or trainers provide appropriate amounts and levels of test training to relevant personnel (such as to acquisition staff, management, testers, and quality assurance) that covers the *potential* testing problems and how to prevent, detect, and react to them.
  - **Management support** – Managers explicitly state (and provide) their support for testing and the need to avoid the commonly occurring test problems.
- **Detection Solutions:**
  - **Evaluate documentation** – Review, inspect, or walk through the test-related documentation (e.g., *Test Plan* and test sections of development plans).
  - **Oversight** – Provide acquirer, management, quality assurance, and peer oversight of the as-performed testing process.
  - **Metrics** – Collect, analyze, and report relevant test metrics to stakeholders (e.g., acquirers, managers, chief and chief testers).
- **Reaction Solutions:**
  - **Reject test documentation** – Acquirers, managers, and chief engineers refuse to accept test-related documentation until identified problems are solved.

- **Reject test results** – Acquirers, managers, and chief engineers refuse to accept test results until identified problems (e.g., in test environments, test procedures, or test cases) are solved.
- **Provide training** – Chief testers or trainers provide appropriate amounts and levels of remedial test training to relevant personnel (such as to acquisition staff, management, testers, and quality assurance) that covers the *observed* testing problems and how to prevent, detect, and react to them.
- **Update process** – Chief engineers, chief testers, and/or process engineers update to test process documentation to minimize the likelihood of reoccurrence of the observed testing problems.

2.2 Solutions to General Testing Problems

#	Testing Problem	Recommended Solutions
1.1	**Wrong Test Mindset**	Explicitly state that the goal of testing is to find defects by causing failures, not to prove that there are no defects (i.e., to break the system rather than show that it works). Provide test training including an overview for non-testers. Emphasize looking for defects where they are most likely to hide (e.g., boundary values and corner cases).
1.2	**Defects Discovered Late**	Improve the effectiveness or earlier disciplines and types of testing (e.g., by improving methods and providing training). Increase the amount of earlier verification of the requirements, architecture, and design (e.g., with peer-level reviews and inspections). Measure the number of defects slipping through multiple disciplines and types of testing (e.g., where the defect was introduced and where it was found).
1.3	**Inadequate Testing Expertise**	Hire full time (i.e., professional) testers. Obtain independent support for testing oversight. Provide appropriate amounts of test training (both classroom and on-the-job).
1.4	**Inadequate Schedule**	Ensure that adequate time for testing is included in the program master schedule including the testing of off-nominal behavior and the specialty engineering testing of quality requirements. Automate as much of the testing as is practical.
1.5	**Inadequate Test Metrics**	Incorporate a robust metrics program in the test plan that covers leading indicators. Ensure that the metrics cover the productivity of the testers and their effectiveness at finding defects (e.g., defects found per test or per day). Also ensure that the number of latent *undiscovered* defects remaining is estimated (e.g., using COQUALMO).

1.6	**Inadequate Test-related Risk Management**	Ensure that test-related risks are identified, incorporated into the official project database, and provided realistic probabilities, harm severities, and priorities.
1.7	**Unrealistic Expectations**	Ensure via training and consulting that non-testers (e.g., managers and customer representatives) understand that (1) testing will not detect all (or even a majority of) defects, (2) *no* testing is truly exhaustive, and therefore (3) testing cannot *prove* that the system works. Do not rely on testing for the verification of all requirements, especially architecturally-significant quality requirements.
1.8	**False Sense of Security**	Ensure that non-testers understand that (1) passing testing does *not prove* (or demonstrate) that there are no remaining defects, (2) realize that a passed test could result from a weak test rather than a lack of defects, and (3) a truly successful test is one that finds one or more defects.
1.9	**Over-reliance on COTS Testing Tools**	Ensure that testers (e.g., via training and test planning) understand the limits of testing tools and the automation of test case creation. Ensure that testers need to use the requirements, architecture, and design as the test oracle (to determine the correct test result). Let the test methodology drive tool drive tool selection. Ensure that testers are not relying on test tools to automate test case selection and completion ("coverage") criteria.
1.10	**Inadequate Documentation**	Ensure via contract, planning, and training that testing assets (e.g., test documents, environments, and test cases) are sufficiently documented to be useful during maintenance and to be reusable (e.g., within product lines).
1.11	**Inadequate Maintenance**	Ensure that testing assets (e.g., test software and documents such as test cases, test procedures, test drivers, and test stubs) are adequately maintained as defects are found and system changes are introduced (e.g., due to refactoring). Ensure that testing assets remain consistent with the current requirements, architecture, design, and implementation. When an agile development cycle with numerous increments is used, ensure that regression test assets are updated as needed.
1.12	**Inadequate Prioritization**	Prioritize testing according to the criticality (e.g., mission, safety, and security) of the subsystem or software being tested and the degree to which the test is likely to elicit important failures. Perform the highest priority tests of a given type first. Do not postpone difficult but important testing until late in the schedule.

1.13	**Inadequate Configuration Management (CM)**	Ensure that all test plans, procedures, test cases, and other testing work products are placed under configuration control before they are used.
1.14	**Lack of Review**	Ensure that the following test assets are reviewed prior to actual testing: test inputs, preconditions (pre-test state), and test oracle including expected test outputs and postconditions.
1.15	**Poor Communication**	Ensure that there is sufficient testing-related communication between: • Teams within large or geographically-distributed programs • Contractually separated teams (prime vs. subcontractor, system of systems) • Between testers and: — Other developers (requirements engineers, architects, designers, and implementers) — Other testers — Customers, user representatives, and subject matter experts (SMEs)
1.16	**External Influences**	Ensure that trained testers determine what constitutes a bug or a defect *worth reporting*. Managerial pressure exists to not find defects (e.g., until after delivery because the project so far behind schedule that there is no time to fix any defects found).

2.3 Solutions to Test Planning Problems

#	Testing Problems	Recommended Solutions
2.1	**No Separate Test Plan**	Ensure that there is a separate *Test and Evaluation Master Plan* (TEMP) or *Software Test Plan* (STP). Do not be satisfied with incomplete high-level overviews of testing in *System Engineering Master Plans* (SEMPs) and *Software Development Plans* (SDPs).
2.2	**Incomplete Test Planning**	Ensure that all test planning documents include clear and specific: • test objectives • testing methods and techniques (e.g., testing is ad hoc, and planning documents merely list the different types of testing rather than state how the testing will be performed) • test case selection criteria (e.g., single nominal test case vs. boundary value testing) • test completion (“coverage”) criteria (e.g., does it include both nominal and off-nominal testing)
2.3	**One-Size-Fits-All Test**	Ensure that the test planning documents contain appropriate system-specific information and

	Planning	are not limited to generic boilerplate. Ensure that mission-, safety-, and security-critical software are required to be tested more completely and rigorously than other less-critical software.
2.4	**Inadequate Resources Planned**	Ensure that the test planning documents and schedules provide for adequate test resources. Specifically, ensure that the test plans provide sufficient: • test time in schedule with inadequate schedule reserves • trained and experienced testers and reviewers • funding • test tools and environments (e.g., integration test beds)

2.4 Solutions to Requirements-Related Problems

#	Testing Problems	Recommended Solutions
3.1	**Ambiguous Requirements**	Promote testability by ensuring that requirements are unambiguous and do not include: • inherently ambiguous words • undefined technical terms and acronyms • quantities without associated units of measure • synonyms
3.2	**Missing Requirements**	Promote testability by ensuring that use case analysis adequately addresses fault tolerant and failure (i.e., rainy day) paths as well as normal (sunny day) paths. Ensure that the requirements repository includes a sufficient amount of the following types of requirements: • Requirements for off-nominal behavior including error, fault, and failure detection and reaction • Quality requirements (e.g., availability, interoperability, maintainability, performance, portability, reliability, robustness, safety, security, and usability) • Data requirements
3.3	**Incomplete Requirements**	Promote testability by ensuring that requirements are complete, including (where appropriate): • Preconditions and trigger events • Quantitative thresholds

		• Postconditions
3.3	**Requirements Lack Good Characteristics**	Promote testability by ensuring that the requirements exhibit the characteristics of good requirements such as being complete, consistent, correct, feasible, mandatory, testable and unambiguous.
3.4	**Unstable Requirements**	Promote testability by ensuring that requirements are reasonably stable so that test cases (test inputs, preconditions, and expected test outputs) and automated regression tests are not constantly being obsoleted because of requirements changes.
3.5	**Poorly Derived Requirements**	Promote testability by ensuring that the derived and allocated requirements are at the proper level of abstraction.

2.5 Solutions to Unit Testing Problems

#	Testing Problems	Recommended Solutions
4.1	**Unstable Design**	Promote testability by ensuring that the design is reasonably stable so that test cases do not need to be constantly updated and test hooks are not lost due to refactoring and new capabilities.[5]
4.2	**Inadequate Detailed Design**	Ensure that sufficient design detail exists to drive the unit testing.
4.3	**Poor Fidelity of Test Environment**	Ensure adequate fidelity of the test environment so that unit testing does not experience many false positives due to using a: • different compiler [version] than the delivered code • software test environment with poor hardware simulation

2.6 Solutions to Integration Testing Problems

#	Testing Problems	Recommended Solutions
5.1	**Defect Localization**	Ensure that the design and implementation (with exception handling, BIT, and test hooks), the

[5] This is especially important with agile development cycles with many short-duration increments and with projects where off-nominal behavior is postponed until late increments.

		tests, and the test tools make it relatively easy to determine the location of defects: • the new or updated operational software under test • the operational hardware under test • the COTS OS and middleware • the software test bed (e.g., in software simulations of hardware) • the hardware test beds (e.g., in pre-production hardware) • the tests themselves (e.g., in the test inputs, preconditions, expected outputs, and expected postconditions) • a configuration/version mismatch among them
5.2	**Insufficient Test Environments**	Ensure that there are a sufficient number of test environments so that there is not excessive completion between the integration testers and other testers for time on them.
5.3	**Schedule Conflicts**	Ensure that there are a sufficient number of test environments so that it is reasonably easy to optimally schedule the allocation of test teams to test beds so that excessive time is not wasted reconfiguring the test environment for the next team's use.
5.4	**Unavailable Components**	Ensure that the operational software, simulation software, test hardware, and actual hardware components are available for integration into the test environments prior to scheduled integration testing.
5.5	**Poor Test Bed Quality**	Ensure that the test environments are of sufficient quality (e.g., via good development practices, adequate testing, and careful tool selection) so that defects in the test environments do not make it difficult to schedule and perform testing.
5.6	**Inadequate Self-Test**	Ensure that the operational software or subsystem contains sufficient test hooks, built-in-test (BIT), or prognostics and health management (PHM) software so that failures are reasonably easy to cause, reproduce, and localize.

2.7 Solutions to Specialty Engineering Testing Problems

#	Testing Problems	Recommended Solutions
6.1	**Inadequate Capacity Testing**	Ensure that all capacity requirements are adequately tested to determine performance as capacity limits are approached, reached, and exceeded.
6.2	**Inadequate Reliability**	To the degree that testing as opposed to analysis is practical as a verification method, ensure

	Testing	that all reliability requirements undergo sufficient long duration reliability testing under operational profiles to estimate the system's reliability.
6.3	**Inadequate Robustness Testing**	Ensure that there is sufficient testing of all robustness requirements to verify adequate error, fault, failure, and environmental tolerance. Ensure that this testing is based on proper robustness analysis such as off-nominal (i.e., fault, degraded mode, and failure) use case paths, Event Tree Analysis (ETA), Fault Tree Analysis (FTA), or Failure Modes Effects Criticality Analysis (FMECA).
6.4	**Inadequate Safety Testing**	Ensure that there is sufficient blackbox testing of all safety requirements and sufficient graybox/whitebox testing of safeguards (e.g., interlocks) and fail-safe behavior. Ensure that this testing is based on adequate safety analysis (e.g., abuse/mishap cases) as well as the safety architecture and design.
6.5	**Inadequate Security Testing**	Ensure that there is sufficient security testing (e.g., penetration testing) of all security requirements, security features, security controls, and fail-secure behavior. Ensure that this testing is based on adequate security analysis (e.g., attack trees, abuse/misuse cases). Note: use static vulnerability analysis tools to identify commonly occurring security vulnerabilities.
6.6	**Inadequate Usability Testing**	Ensure that there is sufficient usability testing of the human interfaces to test all usability requirements including user friendliness, learnability, etc.

2.8 Solutions to System Testing Problems

#	Testing Problems	Recommended Solutions
7.1	**Testing Fault Tolerance is Difficult**	Ensure adequate test tool support or that sufficient robustness including error, fault, and failure logging is incorporated into the system to enable adequate testing for tolerance (e.g., by causing encapsulated errors and faults, and observing the resulting robustness).
7.2	**Testing Code Coverage is Difficult**	Ensure that unit and integration testing (including regression testing) have achieved sufficient code coverage so that tests of the integrated system need not demonstrate code coverage.
7.3	**Lack of Test Hooks**	Ensure that unit and integration testing have adequately tested locally implemented and encapsulated requirements that are difficult to verify during system testing due to low controllability and observability.

2.9 Solutions to System of System (SoS) Testing Problems

#	Testing Problems	Recommended Solutions
8.1	**Inadequate SoS Planning**	Ensure that adequate SoS test planning has occurred above the individual system level. Ensure that there are clear test completion/acceptance criteria at the SoS level.
8.2	**Poor or Missing SoS Requirements**	Ensure that there are sufficient officially approved SoS requirements to drive requirements-based SoS testing.
8.3	**Unclear Testing Responsibilities**	Ensure that responsibilities for testing the end-to-end SoS behavior are clearly assigned to some organization and project.
8.4	**Testing not Funded**	Ensure that adequate funding for testing the end-to-end SoS behavior is clearly supplied to the responsible organization and project.
8.5	**Testing not Properly Scheduled**	Ensure that SoS testing is on the individual systems' integrated master schedules. Ensure that SoS testing is also on the SoS master schedule. Ensure that SoS testing is coordinated with the schedules of the individual systems.
8.6	**Inadequate Test Support from Individual Systems**	Ensure that the individual projects provide adequate test resources (e.g., people and test beds) to support SoS testing. Ensure that these resources are no already committed elsewhere.
8.7	**Poor Defect Tracking Across Projects**	Ensure that defect tracking and associated regression testing across the individual projects of the systems making up the SoS are adequately coordinated.
8.8	**Finger-Pointing**	Work to prevent finger pointing across project boundaries regarding where defects lie (i.e., in which systems and in which project's testing).

2.10 Solutions to Regression/Maintenance Testing Problems

#	Testing Problems	Recommended Solutions
9.1	**Insufficient Automation**	Ensure sufficient automation of regression/maintenance to keep the testing effort acceptable, especially when an agile (iterative, incremental, and parallel) development cycle results in large numbers of increments that must be retested.
9.2	**Regression Tests Not Rerun**	Ensure that sufficient regression testing is being performed by providing sufficient time and staffing to perform it as well as ensuring adequate automation. Resist efforts to skip regression

		testing because of the “minor scope of most changes”; defects have a way of causing propagating faults and failures.
9.3	**Inadequate Scope of Regression Testing**	Resist efforts to limit the scope of regression testing because of the “change can’t effect the rest of the system”; defects have a way of causing propagating faults and failures.
9.4	**Only Low-Level Regression Tests**	Ensure that all relevant levels of regression testing (e.g., unit, integration, system, specialty, and SoS) are rerun when changes are made.
9.5	**Disagreement over Funding**	Ensure that the funding for maintenance testing is clearly assigned to either the development or sustainment budget.