Certified Tester

Advanced Level Syllabus
Test Analyst

Version 2012

International Software Testing Qualifications Board

Advanced Level Test Analyst Sub Working Group: Judy McKay (Chair), Mike Smith, Erik Van Veenendaal; 2010-2012.

Revision History

Version	Date	Remarks
ISEB v1.1	04SEP01	ISEB Practitioner Syllabus
ISTQB 1.2E	SEP03	ISTQB Advanced Level Syllabus from EOQ-SG
V2007	12OCT07	Certified Tester Advanced Level syllabus version 2007
D100626	26JUN10	Incorporation of changes as accepted in 2009, separation of chapters for the separate modules
D101227	27DEC10	Acceptance of changes to format and corrections that have no impact on the meaning of the sentences.
D2011	23OCT11	Change to split syllabus, re-worked LOs and text changes to match LOs. Addition of BOs.
Alpha 2012	09MAR12	Incorporation of all comments from NBs received from October release.
Beta 2012	07APR12	Incorporation of all comments from NBs received from the Alpha release.
Beta 2012	07APR12	Beta Version submitted to GA
Beta 2012	08JUN12	Copy edited version released to NBs
Beta 2012	27JUN12	EWG and Glossary comments incorporated
RC 2012	15AUG12	Release candidate version - final NB edits included
GA 2012	19OCT12	Final edits and cleanup for GA release

Table of Contents

Acknowledgements

This document was produced by a core team from the International Software Testing Qualifications Board Advanced Level Sub Working Group - Advanced Test Analyst: Judy McKay (Chair), Mike Smith, Erik van Veenendaal.

The core team thanks the review team and the National Boards for their suggestions and input.

At the time the Advanced Level Syllabus was completed the Advanced Level Working Group had the following membership (alphabetical order):

Graham Bath, Rex Black, Maria Clara Choucair, Debra Friedenberg, Bernard Homès (Vice Chair), Paul Jorgensen, Judy McKay, Jamie Mitchell, Thomas Mueller, Klaus Olsen, Kenji Onishi, Meile Posthuma, Eric Riou du Cosquer, Jan Sabak, Hans Schaefer, Mike Smith (Chair), Geoff Thompson, Erik van Veenendaal, Tsuyoshi Yumoto.

The following persons participated in the reviewing, commenting and balloting of this syllabus:

Graham Bath, Arne Becher, Rex Black, Piet de Roo, Frans Dijkman, Mats Grindal, Kobi Halperin, Bernard Homès, Maria Jönsson, Junfei Ma, Eli Margolin, Rik Marselis, Don Mills, Gary Mogyorodi, Stefan Mohacsi, Reto Mueller, Thomas Mueller, Ingvar Nordstrom, Tal Pe'er, Raluca Madalina Popescu, Stuart Reid, Jan Sabak, Hans Schaefer, Marco Sogliani, Yaron Tsubery, Hans Weiberg, Paul Weymouth, Chris van Bael, Jurian van der Laar, Stephanie van Dijk, Erik van Veenendaal, Wenqiang Zheng, Debi Zylbermann.

This document was formally released by the General Assembly of the ISTQB® on October 19th, 2012.

0. Introduction to this Syllabus

0.1 Purpose of this Document

This syllabus forms the basis for the International Software Testing Qualification at the Advanced Level for the Test Analyst. The ISTQB® provides this syllabus as follows:

1. To National Boards, to translate into their local language and to accredit training providers. National Boards may adapt the syllabus to their particular language needs and modify the references to adapt to their local publications.
2. To Exam Boards, to derive examination questions in their local language adapted to the learning objectives for each syllabus.
3. To training providers, to produce courseware and determine appropriate teaching methods.
4. To certification candidates, to prepare for the exam (as part of a training course or independently).
5. To the international software and systems engineering community, to advance the profession of software and systems testing, and as a basis for books and articles.

The ISTQB® may allow other entities to use this syllabus for other purposes, provided they seek and obtain prior written permission.

0.2 Overview

The Advanced Level is comprised of three separate syllabi:

- Test Manager
- Test Analyst
- Technical Test Analyst

The Advanced Level Overview document [ISTQB_AL_OVIEW] includes the following information:

- Business Outcomes for each syllabus
- Summary for each syllabus
- Relationships among the syllabi
- Description of cognitive levels (K-levels)
- Appendices

0.3 Examinable Learning Objectives

The Learning Objectives support the Business Outcomes and are used to create the examination for achieving the Advanced Test Analyst Certification. In general all parts of this syllabus are examinable at a K1 level. That is, the candidate will recognize, remember and recall a term or concept. The learning objectives at K2, K3 and K4 levels are shown at the beginning of the pertinent chapter.

1. Testing Process - 300 mins.

Keywords

concrete test case, exit criteria, high-level test case, logical test case, low-level test case, test control, test design, test execution, test implementation, test planning

Learning Objectives for Testing Process

1.2 Testing in the Software Development Lifecycle

TA-1.2.1 (K2) Explain how and why the timing and level of involvement for the Test Analyst varies when working with different lifecycle models

1.3 Test Monitoring, Planning and Control

TA-1.3.1 (K2) Summarize the activities performed by the Test Analyst in support of planning and controlling the testing

1.4 Test Analysis

TA-1.4.1 (K4) Analyze a given scenario, including a project description and lifecycle model, to determine appropriate tasks for the Test Analyst during the analysis and design phases

1.5 Test Design

TA-1.5.1 (K2) Explain why test conditions should be understood by the stakeholders

TA-1.5.2 (K4) Analyze a project scenario to determine the most appropriate use for low-level (concrete) and high-level (logical) test cases

1.6 Test Implementation

TA-1.6.1 (K2) Describe the typical exit criteria for test analysis and test design and explain how meeting those criteria affect the test implementation effort

1.7 Test Execution

TA-1.7.1 (K3) For a given scenario, determine the steps and considerations that should be taken when executing tests

1.8 Evaluating Exit Criteria and Reporting

TA-1.8.1 (K2) Explain why accurate test case execution status information is important

1.9 Test Closure Activities

TA-1.9.1 (K2) Provide examples of work products that should be delivered by the Test Analyst during test closure activities

1.1 Introduction

In the ISTQB® Foundation Level syllabus, the fundamental test process was described as including the following activities:

- Planning, monitoring and control
- Analysis and design
- Implementation and execution
- Evaluating exit criteria and reporting
- Test closure activities

At the Advanced Level, some of these activities are considered separately in order to provide additional refinement and optimization of the processes, to better fit the software development lifecycle, and to facilitate effective test monitoring and control. The activities at this level are considered as follows:

- Planning, monitoring and control
- Analysis
- Design
- Implementation
- Execution
- Evaluating exit criteria and reporting
- Test closure activities

These activities can be implemented sequentially or some can be implemented in parallel, e.g., design could be performed in parallel with implementation (e.g., exploratory testing). Determining the right tests and test cases, designing them and executing them are the primary areas of concentration for the Test Analyst. While it is important to understand the other steps in the test process, the majority of the Test Analyst's work usually is done during the analysis, design, implementation and execution activities of the testing project.

Advanced testers face a number of challenges when introducing the different testing aspects described in this syllabus into the context of their own organizations, teams and tasks. It is important to consider the different software development lifecycles as well as the type of system being tested as these factors can influence the approach to testing.

1.2 Testing in the Software Development Lifecycle

The long-term lifecycle approach to testing should be considered and defined as part of the testing strategy. The moment of involvement for the Test Analyst is different for the various lifecycles and the amount of involvement, time required, information available and expectations can be quite varied as well. Because the testing processes do not occur in isolation, the Test Analyst must be aware of the points where information may be supplied to the other related organizational areas such as:

- Requirements engineering and management - requirements reviews
- Project management - schedule input
- Configuration and change management - build verification testing, version control
- Software development - anticipating what is coming, and when
- Software maintenance - defect management, turnaround time (i.e., time from defect discovery to defect resolution)
- Technical support - accurate documentation for workarounds
- Production of technical documentation (e.g., database design specifications) - input to these documents as well as technical review of the documents

Testing activities must be aligned with the chosen software development lifecycle model whose nature may be sequential, iterative, or incremental. For example, in the sequential V-model, the ISTQB® fundamental test process applied to the system test level could align as follows:

- System test planning occurs concurrently with project planning, and test control continues until system test execution and closure are complete.
- System test analysis and design occur concurrently with requirements specification, system and architectural (high-level) design specification, and component (low-level) design specification.
- System test environment (e.g., test beds, test rig) implementation might start during system design, though the bulk of it typically would occur concurrently with coding and component test, with work on system test implementation activities stretching often until just days before the start of system test execution.
- System test execution begins when the system test entry criteria are all met (or waived), which typically means that at least component testing and often also component integration testing are complete. System test execution continues until the system test exit criteria are met.
- Evaluation of system test exit criteria and reporting of system test results occur throughout system test execution, generally with greater frequency and urgency as project deadlines approach.
- System test closure activities occur after the system test exit criteria are met and system test execution is declared complete, though they can sometimes be delayed until after acceptance testing is over and all project activities are finished.

Iterative and incremental models may not follow the same order of tasks and may exclude some tasks. For example, an iterative model may utilize a reduced set of the standard test processes for each iteration. Analysis and design, implementation and execution, and evaluation and reporting may be conducted for each iteration, whereas planning is done at the beginning of the project and the closure reporting is done at the end. In an Agile project, it is common to use a less formalized process and a much closer working relationship that allows changes to occur more easily within the project. Because Agile is a “light weight” process, there is less comprehensive test documentation in favor of having a more rapid method of communication such as daily “stand up” meetings (called “stand up” because they are very quick, usually 10-15 minutes, so no one needs to sit down and everyone stays engaged).

Agile projects, out of all the lifecycle models, require the earliest involvement from the Test Analyst. The Test Analyst should expect to be involved from the initiation of the project, working with the developers as they do their initial architecture and design work. Reviews may not be formalized but are continuous as the software evolves. Involvement is expected to be throughout the project and the Test Analyst should be available to the team. Because of this immersion, members of Agile teams are usually dedicated to single projects and are fully involved in all aspects of the project.

Iterative/incremental models range from the Agile approach, where there is an expectation for change as the software evolves, to iterative/incremental development models that exist within a V-model (sometimes called embedded iterative). In the case with an embedded iterative model, the Test Analyst should expect to be involved in the standard planning and design aspects, but would then move to a more interactive role as the software is developed, tested, changed and deployed.

Whatever the software development lifecycle being used, it is important for the Test Analyst to understand the expectations for involvement as well as the timing of that involvement. There are many hybrid models in use, such as the iterative within a V-model noted above. The Test Analyst often must determine the most effective role and work toward that rather than depending on the definition of a set model to indicate the proper moment of involvement.

1.3 Test Planning, Monitoring and Control

This section focuses on the processes of planning, monitoring and controlling testing.

1.3.1 Test Planning

Test planning for the most part occurs at the initiation of the test effort and involves the identification and planning of all of the activities and resources required to meet the mission and objectives identified in the test strategy. During test planning it is important for the Test Analyst, working with the Test Manager, to consider and plan for the following:

- Be sure the test plans are not limited to functional testing. All types of testing should be considered in the test plan and scheduled accordingly. For example, in addition to functional testing, the Test Analyst may be responsible for usability testing. That type of testing must also be covered in a test plan document.
- Review the test estimates with the Test Manager and ensure adequate time is budgeted for the procurement and validation of the testing environment.
- Plan for configuration testing. If multiple types of processors, operating systems, virtual machines, browsers, and various peripherals can be combined into many possible configurations, plan to apply testing techniques that will provide adequate coverage of these combinations.
- Plan to test the documentation. Users are provided with the software and with documentation. The documentation must be accurate to be effective. The Test Analyst must allocate time to verify the documentation and may need to work with the technical writing staff to help prepare data to be used for screen shots and video clips.
- Plan to test the installation procedures. Installation procedures, as well as backup and restore procedures, must be tested sufficiently. These procedures can be more critical than the software; if the software cannot be installed, it will not be used at all. This can be difficult to plan since the Test Analyst is often doing the initial testing on a system that has been pre-configured without the final installation processes in place.
- Plan the testing to align with the software lifecycle. Sequential execution of tasks does not fit into most schedules. Many tasks often need to be performed (at least partly) concurrently. The Test Analyst must be aware of the selected lifecycle and the expectations for involvement during the design, development and implementation of the software. This also includes allocating time for confirmation and regression testing.
- Allow adequate time for identifying and analyzing risks with the cross-functional team. Although usually not responsible for organizing the risk management sessions, the Test Analyst should expect to be involved actively in these activities.

Complex relationships may exist among the test basis, test conditions and test cases such that many-to-many relationships may exist among these work products. These need to be understood to enable test planning and control to be effectively implemented. The Test Analyst is usually the best person to determine these relationships and to work to separate dependencies as much as possible.

1.3.2 Test Monitoring and Control

While test monitoring and control is usually the job of the Test Manager, the Test Analyst contributes the measurements that make the control possible.

A variety of quantitative data should be gathered throughout the software development lifecycle (e.g., percentage of planning activities completed, percentage of coverage attained, number of test cases that have passed, failed). In each case a baseline (i.e., reference standard) must be defined and then progress tracked with relation to this baseline. While the Test Manager will be concerned with compiling and reporting the summarized metric information, the Test Analyst gathers the information for each metric. Each test case that is completed, each defect report that is written, each milestone

that is achieved will roll up into the overall project metrics. It is important that the information entered into the various tracking tools be as accurate as possible so the metrics reflect reality.

Accurate metrics allow managers to manage a project (monitor) and to initiate changes as needed (control). For example, a high number of defects being reported from one area of the software may indicate that additional testing effort is needed in that area. Requirements and risk coverage information (traceability) may be used to prioritize remaining work and to allocate resources. Root cause information is used to determine areas for process improvement. If the data that is recorded is accurate, the project can be controlled and accurate status information can be reported to the stakeholders. Future projects can be planned more effectively when the planning considers data gathered from past projects. There are myriad uses for accurate data. It is part of the Test Analyst's job to ensure that the data is accurate, timely and objective.

1.4 Test Analysis

During test planning, the scope of the testing project is defined. The Test Analyst uses this scope definition to:

- Analyze the test basis
- Identify the test conditions

In order for the Test Analyst to proceed effectively with test analysis, the following entry criteria should be met:

- There is a document describing the test object that can serve as the test basis
- This document has passed review with reasonable results and has been updated as needed after the review
- There is a reasonable budget and schedule available to accomplish the remaining testing work for this test object

Test conditions are typically identified by analysis of the test basis and the test objectives. In some situations, where documentation may be old or non-existent, the test conditions may be identified by talking to relevant stakeholders (e.g., in workshops or during sprint planning). These conditions are then used to determine what to test, using test design techniques identified within the test strategy and/or the test plan.

While test conditions are usually specific to the item being tested, there are some standard considerations for the Test Analyst.

- It is usually advisable to define test conditions at differing levels of detail. Initially, high-level conditions are identified to define general targets for testing, such as "functionality of screen x". Subsequently, more detailed conditions are identified as the basis of specific test cases, such as "screen x rejects an account number that is one digit short of the correct length". Using this type of hierarchical approach to defining test conditions can help to ensure the coverage is sufficient for the high-level items.
- If product risks have been defined, then the test conditions that will be necessary to address each product risk must be identified and traced back to that risk item.

At the conclusion of the test analysis activities, the Test Analyst should know what specific tests must be designed in order to meet the needs of the assigned areas of the test project.

1.5 Test Design

Still adhering to the scope determined during test planning, the test process continues as the Test Analyst designs the tests which will be implemented and executed. The process of test design includes the following activities:

- Determine in which test areas low-level (concrete) or high-level (logical) test cases are most appropriate
- Determine the test case design technique(s) that provide the necessary test coverage
- Create test cases that exercise the identified test conditions

Prioritization criteria identified during risk analysis and test planning should be applied throughout the process, from analysis and design to implementation and execution.

Depending on the types of tests being designed, one of the entry criteria for test design may be the availability of tools that will be used during the design work.

When designing tests, it is important to remember the following:

- Some test items are better addressed by defining only the test conditions rather than going further into defining scripted tests. In this case, the test conditions should be defined to be used as a guide for the unscripted testing.
- The pass/fail criteria should be clearly identified.
- Tests should be designed to be understandable by other testers, not just the author. If the author is not the person who executes the test, other testers will need to read and understand previously specified tests in order to understand the test objectives and the relative importance of the test.
- Tests must also be understandable by other stakeholders such as developers, who will review the tests, and auditors, who may have to approve the tests.
- Tests should be designed to cover all the interactions of the software with the actors (e.g., end users, other systems), not just the interactions that occur through the user-visible interface. Inter-process communications, batch execution and other interrupts also interact with the software and can contain defects so the Test Analyst must design tests to mitigate these risks.
- Tests should be designed to test the interfaces between the various test objects.

1.5.1 Concrete and Logical Test Cases

One of the jobs of the Test Analyst is to determine the best types of test cases for a given situation. Concrete test cases provide all the specific information and procedures needed for the tester to execute the test case (including any data requirements) and verify the results. Concrete test cases are useful when the requirements are well-defined, when the testing staff is less experienced and when external verification of the tests, such as audits, is required. Concrete test cases provide excellent reproducibility (i.e., another tester will get the same results), but may also require a significant amount of maintenance effort and tend to limit tester ingenuity during execution.

Logical test cases provide guidelines for what should be tested, but allow the Test Analyst to vary the actual data or even the procedure that is followed when executing the test. Logical test cases may provide better coverage than concrete test cases because they will vary somewhat each time they are executed. This also leads to a loss in reproducibility. Logical test cases are best used when the requirements are not well-defined, when the Test Analyst who will be executing the test is experienced with both testing and the product, and when formal documentation is not required (e.g., no audits will be conducted). Logical test cases may be defined early in the requirements process when the requirements are not yet well-defined. These test cases may be used to develop concrete test cases when the requirements become more defined and stable. In this case, the test case creation is done sequentially, flowing from logical to concrete with only the concrete test cases used for execution.

1.5.2 Creation of Test Cases

Test cases are designed by the stepwise elaboration and refinement of the identified test conditions using test design techniques (see Chapter 3) identified in the test strategy and/or the test plan. The

test cases should be repeatable, verifiable and traceable back to the test basis (e.g., requirements) as dictated by the test strategy that is being used.

Test case design includes the identification of the following:

- Objective
- Preconditions, such as either project or localized test environment requirements and the plans for their delivery, state of the system, etc.
- Test data requirements (both input data for the test case as well as data that must exist in the system for the test case to be executed)
- Expected results
- Post-conditions, such as affected data, state of the system, triggers for subsequent processing, etc.

The level of detail of the test cases, which impacts both the cost to develop and the level of repeatability during execution, should be defined prior to actually creating the test cases. Less detail in the test case allows the Test Analyst more flexibility when executing the test case and provides an opportunity to investigate potentially interesting areas. Less detail, however, also tends to lead to less reproducibility.

A particular challenge is often the definition of the expected result of a test. Computing this manually is often tedious and error-prone; if possible, it is preferable to find or create an automated test oracle. In identifying the expected result, testers are concerned not only with outputs on the screen, but also with data and environmental post-conditions. If the test basis is clearly defined, identifying the correct result, theoretically, should be simple. However, test bases are often vague, contradictory, lacking coverage of key areas, or missing entirely. In such cases, a Test Analyst must have, or have access to, subject matter expertise. Also, even where the test basis is well-specified, complex interactions of complex stimuli and responses can make the definition of the expected results difficult; therefore, a test oracle is essential. Test case execution without any way to determine correctness of results has a very low added value or benefit, often generating spurious failure reports or false confidence in the system.

The activities described above may be applied to all test levels, though the test basis will vary. For example, user acceptance tests may be based primarily on the requirements specification, use cases and defined business processes, while component tests may be based primarily on low-level design specifications, user stories and the code itself. It is important to remember that these activities occur throughout all the test levels although the target of the test may vary. For example, functional testing at the unit level is designed to ensure that a particular component provides the functionality as specified in the detailed design for that component. Functional testing at the integration level is verifying that components interact together and provide functionality through their interaction. At the system level, end to end functionality should be a target of the testing. When analyzing and designing tests, it is important to remember the target level for the test as well as the objective of the test. This helps to determine the level of detail required as well as any tools that may be needed (e.g., drivers and stubs at the component test level).

During the development of test conditions and test cases, some amount of documentation is typically created, resulting in test work products. In practice the extent to which test work products are documented varies considerably. This can be affected by any of the following:

- Project risks (what must/must not be documented)
- The “value added” which the documentation brings to the project
- Standards to be followed and/or regulations to be met
- Lifecycle model used (e.g., an Agile approach aims for “just enough” documentation)
- The requirement for traceability from the test basis through test analysis and design

Depending on the scope of the testing, test analysis and design address the quality characteristics for the test object(s). The ISO 25000 standard [ISO25000] (which is replacing ISO 9126) provides a useful reference. When testing hardware/software systems, additional characteristics may apply.

The processes of test analysis and test design may be enhanced by intertwining them with reviews and static analysis. In fact, conducting the test analysis and test design are often a form of static testing because problems may be found in the basis documents during this process. Test analysis and test design based on the requirements specification is an excellent way to prepare for a requirements review meeting. Reading the requirements to use them for creating tests requires understanding the requirement and being able to determine a way to assess fulfillment of the requirement. This activity often uncovers requirements that are not clear, are untestable or do not have defined acceptance criteria. Similarly, test work products such as test cases, risk analyses, and test plans should be subjected to reviews.

Some projects, such as those following an Agile lifecycle, may have only minimally documented requirements. These are sometimes in the form of “user stories” which describe small but demonstrable bits of functionality. A user story should include a definition of the acceptance criteria. If the software is able to demonstrate that it has fulfilled the acceptance criteria, it is usually considered to be ready for integration with the other completed functionality or may already have been integrated in order to demonstrate its functionality.

During test design the required detailed test infrastructure requirements may be defined, although in practice these may not be finalized until test implementation. It must be remembered that test infrastructure includes more than test objects and testware. For example the infrastructure requirements may include rooms, equipment, personnel, software, tools, peripherals, communications equipment, user authorizations, and all other items required to run the tests.

The exit criteria for test analysis and test design will vary depending on the project parameters, but all items discussed in these two sections should be considered for inclusion in the defined exit criteria. It is important that the criteria be measurable and ensure that all the information and preparation required for the subsequent steps have been provided.

1.6 Test Implementation

Test implementation is the fulfillment of the test design. This includes creating automated tests, organizing tests (both manual and automated) into execution order, finalizing test data and test environments, and forming a test execution schedule, including resource allocation, to enable test case execution to begin. This also includes checking against explicit and implicit entry criteria for the test level in question and ensuring that the exit criteria for the previous steps in the process have been met. If the exit criteria have been skipped, either for the test level or for a step in the test process, the implementation effort is likely to be affected with delayed schedules, insufficient quality and unexpected extra effort. It is important to ensure that all exit criteria have been met prior to starting the test implementation effort.

When determining the execution order, there may be many considerations. In some cases, it may make sense to organize the test cases into test suites (i.e., groups of test cases). This can help organize the testing so that related test cases are executed together. If a risk-based testing strategy is being used, risk priority order may dictate the execution order for the test cases. There may be other factors that determine order such as the availability of the right people, equipment, data and the functionality to be tested. It is not unusual for code to be released in sections and the test effort has to be coordinated with the order in which the software becomes available for test. Particularly in incremental lifecycle models, it is important for the Test Analyst to coordinate with the development team to ensure that the software will be released for testing in a testable order. During test implementation, Test Analysts should finalize and confirm the order in which manual and automated

tests are to be run, carefully checking for constraints that might require tests to be run in a particular order. Dependencies must be documented and checked.

The level of detail and associated complexity for work done during test implementation may be influenced by the detail of the test cases and test conditions. In some cases regulatory rules apply, and tests should provide evidence of compliance to applicable standards such as the United States Federal Aviation Administration's DO-178B/ED 12B [RTCA DO-178B/ED-12B].

As specified above, test data is needed for testing, and in some cases these sets of data can be quite large. During implementation, Test Analysts create input and environment data to load into databases and other such repositories. Test Analysts also create data to be used with data-driven automation tests as well as for manual testing.

Test implementation is also concerned with the test environment(s). During this stage the environment(s) should be fully set up and verified prior to test execution. A "fit for purpose" test environment is essential, i.e., the test environment should be capable of enabling the exposure of the defects present during controlled testing, operate normally when failures are not occurring, and adequately replicate, if required, the production or end-user environment for higher levels of testing. Test environment changes may be necessary during test execution depending on unanticipated changes, test results or other considerations. If environment changes do occur during execution, it is important to assess the impact of the changes to tests that have already been run.

During test implementation, testers must ensure that those responsible for the creation and maintenance of the test environment are known and available, and that all the testware and test support tools and associated processes are ready for use. This includes configuration management, defect management, and test logging and management. In addition, Test Analysts must verify the procedures that gather data for exit criteria evaluation and test results reporting.

It is wise to use a balanced approach to test implementation as determined during test planning. For example, risk-based analytical test strategies are often blended with dynamic test strategies. In this case, some percentage of the test implementation effort is allocated to testing which does not follow predetermined scripts (unscripted).

Unscripted testing should not be ad hoc or aimless as this can be unpredictable in duration and coverage unless time boxed and chartered. Over the years, testers have developed a variety of experience-based techniques, such as attacks, error guessing [Myers79], and exploratory testing. Test analysis, test design, and test implementation still occur, but they occur primarily during test execution.

When following such dynamic test strategies, the results of each test influence the analysis, design, and implementation of the subsequent tests. While these strategies are lightweight and often effective at finding defects, there are some drawbacks. These techniques require expertise from the Test Analyst, duration can be difficult to predict, coverage can be difficult to track and repeatability can be lost without good documentation or tool support.

1.7 Test Execution

Test execution begins once the test object is delivered and the entry criteria to test execution are satisfied (or waived). Tests should be executed according to the plan determined during test implementation, but the Test Analyst should have adequate time to ensure coverage of additional interesting test scenarios and behaviors that are observed during testing (any failure detected during such deviations should be described including the variations from the scripted test case that are necessary to reproduce the failure). This integration of scripted and unscripted (e.g., exploratory) testing techniques helps to guard against test escapes due to gaps in scripted coverage and to circumvent the pesticide paradox.

At the heart of the test execution activity is the comparison of actual results with expected results. Test Analysts must bring attention and focus to these tasks, otherwise all the work of designing and implementing the test can be wasted when failures are missed (false-negative result) or correct behavior is misclassified as incorrect (false-positive result). If the expected and actual results do not match, an incident has occurred. Incidents must be carefully scrutinized to determine the cause (which might or might not be a defect in the test object) and to gather data to assist with the resolution of the incident (see Chapter 6 for further details on defect management).

When a failure is identified, the test documentation (test specification, test case, etc.) should be carefully evaluated to ensure correctness. A test document can be incorrect for a number of reasons. If it is incorrect, it should be corrected and the test should be re-run. Since changes in the test basis and the test object can render a test case incorrect even after the test has been run successfully many times, testers should remain aware of the possibility that the observed results could be due to an incorrect test.

During test execution, test results must be logged appropriately. Tests which were run but for which results were not logged may have to be repeated to identify the correct result, leading to inefficiency and delays. (Note that adequate logging can address the coverage and repeatability concerns associated with test techniques such as exploratory testing.) Since the test object, testware, and test environments may all be evolving, logging should identify the specific versions tested as well as specific environment configurations. Test logging provides a chronological record of relevant details about the execution of tests.

Results logging applies both to individual tests and to activities and events. Each test should be uniquely identified and its status logged as test execution proceeds. Any events that affect the test execution should be logged. Sufficient information should be logged to measure test coverage and document reasons for delays and interruptions in testing. In addition, information must be logged to support test control, test progress reporting, measurement of exit criteria, and test process improvement.

Logging varies depending on the level of testing and the strategy. For example, if automated component testing is occurring, the automated tests should produce most of the logging information. If manual testing is occurring, the Test Analyst will log the information regarding the test execution, often into a test management tool that will track the test execution information. In some cases, as with test implementation, the amount of test execution information that is logged is influenced by regulatory or audit requirements.

In some cases, users or customers may participate in test execution. This can serve as a way to build their confidence in the system, though that presumes that the tests find few defects. Such an assumption is often invalid in early test levels, but might be valid during acceptance test.

The following are some specific areas that should be considered during test execution:

- Notice and explore “irrelevant” oddities. Observations or results that may seem irrelevant are often indicators for defects that (like icebergs) are lurking beneath the surface.
- Check that the product is not doing what it is not supposed to do. Checking that the product does what it is supposed to do is a normal focus of testing, but the Test Analyst must also be sure the product is not misbehaving by doing something it should not be doing (for example, additional undesired functions).
- Build the test suite and expect it to grow and change over time. The code will evolve and additional tests will need to be implemented to cover these new functionalities, as well as to check for regressions in other areas of the software. Gaps in testing are often discovered during execution. Building the test suite is a continuous process.

- Take notes for the next testing effort. The testing tasks do not end when the software is provided to the user or distributed to the market. A new version or release of the software will most likely be produced, so knowledge should be stored and transferred to the testers responsible for the next testing effort.
- Do not expect to rerun all manual tests. It is unrealistic to expect that all manual tests will be rerun. If a problem is suspected, the Test Analyst should investigate it and note it rather than assume it will be caught in a subsequent execution of the test cases.
- Mine the data in the defect tracking tool for additional test cases. Consider creating test cases for defects that were discovered during unscripted or exploratory testing and add them to the regression test suite.
- Find the defects before regression testing. Time is often limited for regression testing and finding failures during regression testing can result in schedule delays. Regression tests generally do not find a large proportion of the defects, mostly because they are tests which have already been run (e.g., for a previous version of same software), and defects should have been detected in those previous runs. This does not mean that regression tests should be eliminated altogether, only that the effectiveness of regression tests, in terms of the capacity to detect new defects, is lower than other tests.

1.8 Evaluating Exit Criteria and Reporting

From the point of view of the test process, test progress monitoring entails ensuring the collection of proper information to support the reporting requirements. This includes measuring progress towards completion. When the exit criteria are defined in the planning stages, there may be a breakdown of "must" and "should" criteria. For example, the criteria might state that there "must be no open Priority 1 or Priority 2 bugs" and there "should be 95% pass rate across all test cases". In this case, a failure to meet the "must" criteria should cause the exit criteria to fail whereas a 93% pass rate could still allow the project to proceed to the next level. The exit criteria must be clearly defined so they can be objectively assessed.

The Test Analyst is responsible for supplying the information that is used by the Test Manager to evaluate progress toward meeting the exit criteria and for ensuring that the data is accurate. If, for example, the test management system provides the following status codes for test case completion:

- Passed
- Failed
- Passed with exception

then the Test Analyst must be very clear on what each of these means and must apply that status consistently. Does "passed with exception" mean that a defect was found but it is not affecting the functionality of the system? What about a usability defect that causes the user to be confused? If the pass rate is a "must" exit criterion, counting a test case as "failed" rather than "passed with exception" becomes a critical factor. There must also be consideration for test cases that are marked as "failed" but the cause of the failure is not a defect (e.g., the test environment was improperly configured). If there is any confusion on the metrics being tracked or the usage of the status values, the Test Analyst must clarify with the Test Manager so the information can be tracked accurately and consistently throughout the project.

It is not unusual for the Test Analyst to be asked for a status report during the testing cycles as well as to contribute to the final report at the end of the testing. This may require gathering metrics from the defect and test management systems as well as assessing the overall coverage and progress. The Test Analyst should be able to use the reporting tools and be able to provide the requested information for the Test Manager to extract the information needed.

1.9 Test Closure Activities

Once test execution is determined to be complete, the key outputs from the testing effort should be captured and either passed to the relevant person or archived. Collectively, these are test closure activities. The Test Analyst should expect to be involved in delivering work products to those who will need them. For example, known defects deferred or accepted should be communicated to those who will use and support the use of the system. Tests and test environments should be given to those responsible for maintenance testing. Another work product may be a regression test set (either automated or manual). Information about test work products must be clearly documented, including appropriate links, and appropriate access rights must be granted.

The Test Analyst should also expect to participate in retrospective meetings ("lessons learned") where important lessons (both from within the testing project and across the whole software development lifecycle) can be documented and plans established to reinforce the "good" and eliminate, or at least control, the "bad". The Test Analyst is a knowledgeable source of information for these meetings and must participate if valid process improvement information is to be gathered. If only the Test Manager will attend the meeting, the Test Analyst must convey the pertinent information to the Test Manager so an accurate picture of the project is presented.

Archiving results, logs, reports, and other documents and work products in the configuration management system must also be done. This task often falls to the Test Analyst and is an important closure activity, particularly if a future project will require the use of this information.

While the Test Manager usually determines what information should be archived, the Test Analyst should also think about what information would be needed if the project were to be started up again at a future time. Thinking about this information at the end of a project can save months of effort when the project is started up again at a later time or with another team.

2. Test Management: Responsibilities for the Test Analyst - 90 mins.

Keywords

product risk, risk analysis, risk identification, risk level, risk management, risk mitigation, risk-based testing, test monitoring, test strategy

Learning Objectives for Test Management: Responsibilities for the Test Analyst

2.2 Test Progress Monitoring and Control

TA-2.2.1 (K2) Explain the types of information that must be tracked during testing to enable adequate monitoring and controlling of the project

2.3 Distributed, Outsourced and Insourced Testing

TA-2.3.1 (K2) Provide examples of good communication practices when working in a 24-hour testing environment

2.4 The Test Analyst's Tasks in Risk-Based Testing

TA-2.4.1 (K3) For a given project situation, participate in risk identification, perform risk assessment and propose appropriate risk mitigation

2.1 Introduction

While there are many areas in which the Test Analyst interacts with and supplies data for the Test Manager, this section concentrates on the specific areas of the testing process in which the Test Analyst is a major contributor. It is expected that the Test Manager will seek the information needed from the Test Analyst.

2.2 Test Progress Monitoring and Control

There are five primary dimensions in which test progress is monitored:

- Product (quality) risks
- Defects
- Tests
- Coverage
- Confidence

Product risks, defects, tests, and coverage can be and often are measured and reported in specific ways during the project or operation by the Test Analyst. Confidence, though measurable through surveys, is usually reported subjectively. Gathering the information needed to support these metrics is part of the Test Analyst's daily work. It is important to remember that the accuracy of this data is critical as inaccurate data will create inaccurate trending information and may lead to inaccurate conclusions. At its worst, inaccurate data will result in incorrect management decisions and damage to the credibility of the test team.

When using a risk-based testing approach, the Test Analyst should be tracking:

- Which risks have been mitigated by testing
- Which risks are considered to be unmitigated

Tracking risk mitigation is often done with a tool that also tracks test completion (e.g., test management tools). This requires that the identified risks are mapped to the test conditions which are mapped to the test cases that will mitigate the risks if the test cases are executed and passed. In this way, the risk mitigation information is updated automatically as the test cases are updated. This can be done for both manual and automated tests.

Defect tracking is usually done via a defect tracking tool. As defects are recorded, particular classification information about each defect is recorded as well. This information is used to produce trends and graphs that indicate the progress of testing and the quality of the software. Classification information is discussed in more detail in the Defect Management chapter. The lifecycle may affect the amount of defect documentation that is recorded and the methods used to record the information.

As the testing is conducted, test case status information should be recorded. This is usually done via a test management tool but can be done by manual means if needed. Test case information can include:

- Test case creation status (e.g., designed, reviewed)
- Test case execution status (e.g., passed, failed, blocked, skipped)
- Test case execution information (e.g., date and time, tester name, data used)
- Test case execution artifacts (e.g., screen shots, accompanying logs)

As with the identified risk items, the test cases should be mapped to the requirements items they test. It is important for the Test Analyst to remember that if test case A is mapped to requirement A, and it is the only test case mapped to that requirement, then when test case A is executed and passes, requirement A will be considered to be fulfilled. This may or may not be correct. In many cases, more

test cases are needed to thoroughly test a requirement, but because of limited time, only a subset of those tests is actually created. For example, if 20 test cases were needed to thoroughly test the implementation of a requirement, but only 10 were created and run, then the requirements coverage information will indicate 100% coverage when in fact only 50% coverage was achieved. Accurate tracking of the coverage as well as tracking the reviewed status of the requirements themselves can be used as a confidence measure.

The amount (and level of detail) of information to be recorded depends on several factors, including the software development lifecycle model. For example, in Agile projects typically less status information will be recorded due to the close interaction of the team and more face-to-face communication.

2.3 Distributed, Outsourced and Insourced Testing

In many cases, not all of the test effort is carried out by a single test team, composed of fellow employees of the rest of the project team, at a single and same location as the rest of the project team. If the test effort occurs at multiple locations, that test effort may be called distributed. If it occurs at a single location it may be called centralized. If the test effort is carried out at one or more locations by people who are not fellow employees of the rest of the project team and who are not co-located with the project team, that test effort may be called outsourced. If the test effort is carried out by people who are co-located with the project team but who are not fellow employees, that test effort may be called insourced.

When working in a project in which some of the test team is spread across multiple locations or even across multiple companies, the Test Analyst must pay special attention to effective communication and information transfer. Some organizations work on a “24 hour testing” model in which the team in one time zone is expected to hand off the work to the team in another time zone to allow testing to continue around the clock. This requires special planning on the part of the Test Analyst who will hand off and receive work. Good planning is important to understand responsibilities, but it is vital to ensure that the proper information is available.

When verbal communication is not available, written communication must suffice. This means that email, status reports and effective use of the test management and defect tracking tools must be employed. If the test management tool allows tests to be assigned to individuals, it can also work as a scheduling tool and an easy way to transfer work between people. Defects that are accurately recorded can be routed to co-workers for follow-up as needed. Effective use of these communication systems is vital for an organization that cannot rely on daily personal interaction.

2.4 The Test Analyst’s Tasks in Risk-Based Testing

2.4.1 Overview

The Test Manager often has overall responsibility for establishing and managing a risk-based testing strategy. The Test Manager usually will request the involvement of the Test Analyst to ensure the risk-based approach is implemented correctly.

The Test Analyst should be actively involved in the following risk-based testing tasks:

- Risk identification
- Risk assessment
- Risk mitigation

These tasks are performed iteratively throughout the project lifecycle to deal with emerging risks, changing priorities and to regularly evaluate and communicate risk status.

Test Analysts should work within the risk-based testing framework established by the Test Manager for the project. They should contribute their knowledge of the business domain risks that are inherent in the project such as risks related to safety, business and economic concerns, and political factors.

2.4.2 Risk Identification

By calling on the broadest possible sample of stakeholders, the risk identification process is most likely to detect the largest possible number of significant risks. Because Test Analysts often possess unique knowledge regarding the particular business domain of the system under test, they are particularly well-suited for conducting expert interviews with the domain experts and users, conducting independent assessments, using and facilitating the use of risk templates, conducting risk workshops, conducting brainstorming sessions with potential and current users, defining testing checklists and calling on past experience with similar systems or projects. In particular, the Test Analyst should work closely with the users and other domain experts to determine the areas of business risk that should be addressed during testing. The Test Analyst also can be particularly helpful in identifying the potential effects of risk on the users and stakeholders.

Sample risks that might be identified in a project include:

- Accuracy issues with the software functionality, e.g., incorrect calculations
- Usability issues, e.g., insufficient keyboard shortcuts
- Learnability issues, e.g., lack of instructions for the user at key decision points

Considerations regarding testing the specific quality characteristics are covered in Chapter 4 of this syllabus.

2.4.3 Risk Assessment

While risk identification is about identifying as many pertinent risks as possible, risk assessment is the study of these identified risks. Specifically, categorizing each risk and determining the likelihood and impact associated with each risk.

Determining the level of risk typically involves assessing, for each risk item, the likelihood of occurrence and the impact upon occurrence. The likelihood of occurrence is usually interpreted as the likelihood that the potential problem can exist in the system under test and will be observed when the system is in production. In other words, it arises from technical risk. The Technical Test Analyst should contribute to finding and understanding the potential technical risk level for each risk item whereas the Test Analyst contributes to understanding the potential business impact of the problem should it occur.

The impact upon occurrence is often interpreted as the severity of the effect on the users, customers, or other stakeholders. In other words, it arises from business risk. The Test Analyst should contribute to identifying and assessing the potential business domain or user impact for each risk item. Factors influencing business risk include:

- Frequency of use of the affected feature
- Business loss
- Potential financial, ecological or social losses or liability
- Civil or criminal legal sanctions
- Safety concerns
- Fines, loss of license
- Lack of reasonable workarounds
- Visibility of the feature
- Visibility of failure leading to negative publicity and potential image damage
- Loss of customers

Given the available risk information, the Test Analyst needs to establish the levels of business risk per the guidelines established by the Test Manager. These could be classified with terms (e.g., low, medium, high) or numbers. Unless there is a way to objectively measure the risk on a defined scale it cannot be a true quantitative measure. Accurately measuring probability/likelihood and cost/consequence is usually very difficult, so determining risk level is usually done qualitatively.

Numbers may be assigned to the qualitative value, but that does not make it a true quantitative measure. For example, the Test Manager may determine that business risk should be categorized with a value from 1 to 10, with 1 being the highest, and therefore riskiest, impact to the business. Once the likelihood (the assessment of the technical risk) and impact (the assessment of the business risk) have been assigned, these values may be multiplied together to determine the overall risk rating for each risk item. That overall rating is then used to prioritize the risk mitigation activities. Some risk-based testing models, such as PRISMA® [vanVeenendaal12], do not combine the risk values, allowing the test approach to address the technical and business risks separately.

2.4.4 Risk Mitigation

During the project, Test Analysts should seek to do the following:

- Reduce product risk by using well-designed test cases that demonstrate unambiguously whether test items pass or fail, and by participating in reviews of software artifacts such as requirements, designs, and user documentation
- Implement appropriate risk mitigation activities identified in the test strategy and test plan
- Re-evaluate known risks based on additional information gathered as the project unfolds, adjusting likelihood, impact, or both, as appropriate
- Recognize new risks identified by information obtained during testing

When one is talking about a product (quality) risk, then testing is a form of mitigation for such risks. By finding defects, testers reduce risk by providing awareness of the defects and opportunities to deal with the defects before release. If the testers find no defects, testing then reduces risk by ensuring that, under certain conditions (i.e., the conditions tested), the system operates correctly. Test Analysts help to determine risk mitigation options by investigating opportunities for gathering accurate test data, creating and testing realistic user scenarios and conducting or overseeing usability studies.

2.4.4.1 Prioritizing the Tests

The level of risk is also used to prioritize tests. A Test Analyst might determine that there is a high risk in the area of transactional accuracy in an accounting system. As a result, to mitigate the risk, the tester may work with other business domain experts to gather a strong set of sample data that can be processed and verified for accuracy. Similarly, a Test Analyst might determine that usability issues are a significant risk for a new product. Rather than wait for a user acceptance test to discover any issues, the Test Analyst might prioritize an early usability test to occur during the integration level to help identify and resolve usability problems early in the testing. This prioritization must be considered as early as possible in the planning stages so that the schedule can accommodate the necessary testing at the necessary time.

In some cases, all of the highest risk tests are run before any lower risk tests, and tests are run in strict risk order (often called “depth-first”); in other cases, a sampling approach is used to select a sample of tests across all the identified risks using risk to weight the selection while at the same time ensuring coverage of every risk at least once (often called “breadth-first”).

Whether risk-based testing proceeds depth-first or breadth-first, it is possible that the time allocated for testing might be consumed without all tests being run. Risk-based testing allows testers to report to management in terms of the remaining level of risk at this point, and allows management to decide whether to extend testing or to transfer the remaining risk onto the users, customers, help desk/technical support, and/or operational staff.

2.4.4.2 Adjusting Testing for Future Test Cycles

Risk assessment is not a one-time activity performed before the start of test implementation; it is a continuous process. Each future planned test cycle should be subjected to new risk analysis to take into account such factors as:

- Any new or significantly changed product risks
- Unstable or defect-prone areas discovered during the testing
- Risks from fixed defects
- Typical defects found during testing
- Areas that have been under-tested (low test coverage)

If additional time for testing is allocated it may be possible to expand the risk coverage into areas of lower risk.

3. Test Techniques - 825 mins.

Keywords

boundary value analysis (BVA), cause-effect graphing, checklist-based testing, classification tree method, combinatorial testing, decision table testing, defect taxonomy, defect-based technique, domain analysis, error guessing, equivalence partitioning, experience-based technique, exploratory testing, orthogonal array, orthogonal array testing, pairwise testing, requirements-based testing, specification-based technique, state transition testing, test charter, use case testing, user story testing

Learning Objectives for Test Techniques

3.2 Specification-Based Techniques

TA-3.2.1 (K2) Explain the use of cause-effects graphs
TA-3.2.2 (K3) Write test cases from a given specification item by applying the equivalence partitioning test design technique to achieve a defined level of coverage
TA-3.2.3 (K3) Write test cases from a given specification item by applying the boundary value analysis test design technique to achieve a defined level of coverage
TA-3.2.4 (K3) Write test cases from a given specification item by applying the decision table test design technique to achieve a defined level of coverage
TA-3.2.5 (K3) Write test cases from a given specification item by applying the state transition test design technique to achieve a defined level of coverage
TA-3.2.6 (K3) Write test cases from a given specification item by applying the pairwise test design technique to achieve a defined level of coverage
TA-3.2.7 (K3) Write test cases from a given specification item by applying the classification tree test design technique to achieve a defined level of coverage
TA-3.2.8 (K3) Write test cases from a given specification item by applying the use case test design technique to achieve a defined level of coverage
TA-3.2.9 (K2) Explain how user stories are used to guide testing in an Agile project
TA-3.2.10 (K3) Write test cases from a given specification item by applying the domain analysis test design technique to achieve a defined level of coverage
TA-3.2.11 (K4) Analyze a system, or its requirement specification, in order to determine likely types of defects to be found and select the appropriate specification-based technique(s)

3.3 Defect-Based Techniques

TA-3.3.1 (K2) Describe the application of defect-based testing techniques and differentiate their use from specification-based techniques
TA-3.3.2 (K4) Analyze a given defect taxonomy for applicability in a given situation using criteria for a good taxonomy

3.4 Experience-Based Techniques

TA-3.4.1 (K2) Explain the principles of experience-based techniques, and the benefits and drawbacks compared to specification-based and defect-based techniques
TA-3.4.2 (K3) For a given scenario, specify exploratory tests and explain how the results can be reported
TA-3.4.3 (K4) For a given project situation, determine which specification-based, defect-based or experience-based techniques should be applied to achieve specific goals

3.1 Introduction

The test design techniques considered in this chapter are divided into the following categories:

- Specification-based (or behavior-based or black box)
- Defect-based
- Experience-based

These techniques are complementary and may be used as appropriate for any given test activity, regardless of which level of testing is being performed.

Note that all three categories of techniques can be used to test both functional or non-functional quality characteristics. Testing non-functional characteristics is discussed in the next chapter.

The test design techniques discussed in these sections may focus primarily on determining optimal test data (e.g., equivalence partitions) or deriving test sequences (e.g., state models). It is common to combine techniques to create complete test cases.

3.2 Specification-Based Techniques

Specification-based techniques are applied to the test conditions to derive test cases based on an analysis of the test basis for a component or system without reference to its internal structure.

Common features of specification-based techniques include:

- Models, e.g., state transition diagrams and decision tables, are created during test design according to the test technique
- Test conditions are derived systematically from these models

Some techniques also provide coverage criteria, which can be used for measuring test design and test execution activities. Completely fulfilling the coverage criteria does not mean that the entire set of tests is complete, but rather that the model no longer suggests any additional tests to increase coverage based on that technique.

Specification-based tests are usually based on the system requirements documents. Since the requirements specification should specify how the system is to behave, particularly in the area of functionality, deriving tests from the requirements is often part of testing the behavior of the system. In some cases there may be no documented requirements but there are implied requirements such as replacing the functionality of a legacy system.

There are a number of specification-based testing techniques. These techniques target different types of software and scenarios. The sections below show the applicability for each technique, some limitations and difficulties that the Test Analyst may experience, the method by which test coverage is measured and the types of defects that are targeted.

3.2.1 Equivalence Partitioning

Equivalence partitioning (EP) is used to reduce the number of test cases that is required to effectively test the handling of inputs, outputs, internal values and time-related values. Partitioning is used to create equivalence classes (often called equivalence partitions) which are created of sets of values that are processed in the same manner. By selecting one representative value from a partition, coverage for all the items in the same partition is assumed.

Applicability
This technique is applicable at any level of testing and is appropriate when all the members of a set of values to be tested are expected to be handled in the same way and where the sets of values used by the application do not interact. The selection of sets of values is applicable to valid and invalid

partitions (i.e., partitions containing values that should be considered invalid for the software being tested). This technique is strongest when used in combination with boundary value analysis which expands the test values to include those on the edges of the partitions. This is a commonly used technique for smoke testing a new build or a new release as it quickly determines if basic functionality is working.

Limitations/Difficulties
If the assumption is incorrect and the values in the partition are not handled in exactly the same way, this technique may miss defects. It is also important to select the partitions carefully. For example, an input field that accepts positive and negative numbers would be better tested as two valid partitions, one for the positive numbers and one for the negative numbers, because of the likelihood of different handling. Depending on whether or not zero is allowed, this could become another partition as well. It is important for the Test Analyst to understand the underlying processing in order to determine the best partitioning of the values.

Coverage
Coverage is determined by taking the number of partitions for which a value has been tested and dividing that number by the number of partitions that have been identified. Using multiple values for a single partition does not increase the coverage percentage.

Types of Defects
This technique finds functional defects in the handling of various data values.

3.2.2 Boundary Value Analysis

Boundary value analysis (BVA) is used to test the values that exist on the boundaries of ordered equivalence partitions. There are two ways to approach BVA: two value or three value testing. With two value testing, the boundary value (on the boundary) and the value that is just over the boundary (by the smallest possible increment) are used. For example, if the partition included the values 1 to 10 in increments of 0.5, the two value test values for the upper boundary would be 10 and 10.5. The lower boundary test values would be 1 and 0.5. The boundaries are defined by the maximum and minimum values in the defined equivalence partition.

For three value boundary testing, the values before, on and over the boundary are used. In the previous example, the upper boundary tests would include 9.5, 10 and 10.5. The lower boundary tests would include 1.5, 1 and 0.5. The decision regarding whether to use two or three boundary values should be based on the risk associated with the item being tested, with the three boundary approach being used for the higher risk items.

Applicability
This technique is applicable at any level of testing and is appropriate when ordered equivalence partitions exist. Ordering is required because of the concept of being on and over the boundary. For example, a range of numbers is an ordered partition. A partition that consists of all rectangular objects is not an ordered partition and does not have boundary values. In addition to number ranges, boundary value analysis can be applied to the following:

- Numeric attributes of non-numeric variables (e.g., length)
- Loops, including those in use cases
- Stored data structures
- Physical objects (including memory)
- Time-determined activities

Limitations/Difficulties
Because the accuracy of this technique depends on the accurate identification of the equivalence partitions, it is subject to the same limitations and difficulties. The Test Analyst should also be aware

of the increments in the valid and invalid values to be able to accurately determine the values to be tested. Only ordered partitions can be used for boundary value analysis but this is not limited to a range of valid inputs. For example, when testing for the number of cells supported by a spreadsheet, there is a partition that contains the number of cells up to and including the maximum allowed cells (the boundary) and another partition that begins with one cell over the maximum (over the boundary).

Coverage
Coverage is determined by taking the number of boundary conditions that are tested and dividing that by the number of identified boundary conditions (either using the two value or three value method). This will provide the coverage percentage for the boundary testing.

Types of Defects
Boundary value analysis reliably finds displacement or omission of boundaries, and may find cases of extra boundaries. This technique finds defects regarding the handling of the boundary values, particularly errors with less-than and greater-than logic (i.e., displacement). It can also be used to find non-functional defects, for example tolerance of load limits (e.g., system supports 10,000 concurrent users).

3.2.3 Decision Tables

Decision tables are used to test the interaction between combinations of conditions. Decision tables provide a clear method to verify testing of all pertinent combinations of conditions and to verify that all possible combinations are handled by the software under test. The goal of decision table testing is to ensure that every combination of conditions, relationships and constraints is tested. When trying to test every possible combination, decision tables can become very large. A method of intelligently reducing the number of combinations from all possible to those which are "interesting" is called collapsed decision table testing. When this technique is used, the combinations are reduced to those that will produce differing outputs by removing sets of conditions that are not relevant for the outcome. Redundant tests or tests in which the combination of conditions is not possible are removed. The decision whether to use full decision tables or collapsed decision tables is usually risk-based. [Copeland03]

Applicability
This technique is commonly applied for the integration, system and acceptance test levels. Depending on the code, it may also be applicable for component testing when a component is responsible for a set of decision logic. This technique is particularly useful when the requirements are presented in the form of flow charts or tables of business rules. Decision tables are also a requirements definition technique and some requirements specifications may arrive already in this format. Even when the requirements are not presented in a tabular or flow-charted form, condition combinations are usually found in the narrative. When designing decision tables, it is important to consider the defined condition combinations as well as those that are not expressly defined but will exist. In order to design a valid decision table, the tester must be able to derive all expected outcomes for all condition combinations from the specification or test oracle. Only when all interacting conditions are considered can the decision table be used as a good test design tool.

Limitations/Difficulties
Finding all the interacting conditions can be challenging, particularly when requirements are not well-defined or do not exist. It is not unusual to prepare a set of conditions and determine that the expected result is unknown.

Coverage
Minimum test coverage for a decision table is to have one test case for each column. This assumes that there are no compound conditions and that all possible condition combinations have been recorded in a column. When determining tests from a decision table, it is also important to consider

any boundary conditions that should be tested. These boundary conditions may result in an increase in the number of test cases needed to adequately test the software. Boundary value analysis and equivalence partitioning are complementary to the decision table technique.

Types of Defects
Typical defects include incorrect processing based on particular combinations of conditions resulting in unexpected results. During the creation of the decision tables, defects may be found in the specification document. The most common types of defects are omissions (there is no information regarding what should actually happen in a certain situation) and contradictions. Testing may also find issues with condition combinations that are not handled or are not handled well.

3.2.4 Cause-Effect Graphing

Cause-effect graphs may be generated from any source which describes the functional logic (i.e., the "rules") of a program, such as user stories or flow charts. They can be useful to gain a graphical overview of a program's logical structure and are typically used as the basis for creating decision tables. Capturing decisions as cause-effect graphs and/or decision tables enables systematic test coverage of the program's logic to be achieved.

Applicability
Cause-effect graphs apply in the same situations as decision tables and also apply to the same testing levels. In particular, a cause-effect graph shows condition combinations that cause results (causality), condition combinations that exclude results (not), multiple conditions that must be true to cause a result (and) and alternative conditions that can be true to cause a particular result (or). These relationships can be easier to see in a cause-effect graph than in a narrative description.

Limitations/Difficulties
Cause-effect graphing requires additional time and effort to learn compared to other test design techniques. It also requires tool support. Cause-effect graphs have a particular notation that must be understood by the creator and reader of the graph.

Coverage
Each possible cause to effect line must be tested, including the combination conditions, to achieve minimum coverage. Cause-effect graphs include a means to define constraints on the data and constraints on the flow of logic.

Types of Defects
These graphs find the same types of combinatorial defects as are found with decision tables. In addition, the creation of the graphs helps define the required level of detail in the test basis, and so helps improve the detail and quality of the test basis and helps the tester identify missing requirements.

3.2.5 State Transition Testing

State transition testing is used to test the ability of the software to enter into and exit from defined states via valid and invalid transitions. Events cause the software to transition from state to state and to perform actions. Events may be qualified by conditions (sometimes called guard conditions or transition guards) which influence the transition path to be taken. For example, a login event with a valid username/password combination will result in a different transition than a login event with an invalid password.

State transitions are tracked in either a state transition diagram that shows all the valid transitions between states in a graphical format or a state table which shows all potential transitions, both valid and invalid.

Applicability
State transition testing is applicable for any software that has defined states and has events that will cause the transitions between those states (e.g., changing screens). State transition testing can be used at any level of testing. Embedded software, web software, and any type of transactional software are good candidates for this type of testing. Control systems, i.e., traffic light controllers, are also good candidates for this type of testing.

Limitations/Difficulties
Determining the states is often the most difficult part of defining the state table or diagram. When the software has a user interface, the various screens that are displayed for the user are often used to define the states. For embedded software, the states may be dependent upon the states that the hardware will experience.

Besides the states themselves, the basic unit of state transition testing is the individual transition, also known as a 0-switch. Simply testing all transitions will find some kinds of state transition defects, but more may be found by testing sequences of transactions. A sequence of two successive transitions is called a 1-switch; a sequence of three successive transitions is a 2-switch, and so forth. (These switches are sometimes alternatively designated as N-1 switches, where N represents the number of transitions that will be traversed. A single transition, for instance (a 0-switch), would be a 1-1 switch. [Bath08]

Coverage
As with other types of test techniques, there is a hierarchy of levels of test coverage. The minimum acceptable degree of coverage is to have visited every state and traversed every transition. 100% transition coverage (also known as 100% 0-switch coverage or 100% logical branch coverage) will guarantee that every state is visited and every transition is traversed, unless the system design or the state transition model (diagram or table) are defective. Depending on the relationships between states and transitions, it may be necessary to traverse some transitions more than once in order to execute other transitions a single time.

The term "n-switch coverage" relates to the number of transitions covered. For example, achieving 100% 1-switch coverage requires that every valid sequence of two successive transitions has been tested at least once. This testing may stimulate some types of failures that 100% 0-switch coverage would miss.

"Round-trip coverage" applies to situations in which sequences of transitions form loops. 100% round-trip coverage is achieved when all loops from any state back to the same state have been tested. This must be tested for all states that are included in loops.

For any of these approaches, a still higher degree of coverage will attempt to include all invalid transitions. Coverage requirements and covering sets for state transition testing must identify whether invalid transitions are included.

Types of Defects
Typical defects include incorrect processing in the current state that is a result of the processing that occurred in a previous state, incorrect or unsupported transitions, states with no exits and the need for states or transitions that do not exist. During the creation of the state machine model, defects may be found in the specification document. The most common types of defects are omissions (there is no information regarding what should actually happen in a certain situation) and contradictions.

3.2.6 Combinatorial Testing Techniques

Combinatorial testing is used when testing software with several parameters, each one with several values, which gives rise to more combinations than are feasible to test in the time allowed. The

parameters must be independent and compatible in the sense that any option for any factor can be combined with any option for any other factor. Classification trees allow for some combinations to be excluded, if certain options are incompatible. This does not assume that the combined factors won't affect each other; they very well might, but should affect each other in acceptable ways.

Combinatorial testing provides a means to identify a suitable subset of these combinations to achieve a predetermined level of coverage. The number of items to include in the combinations can be selected by the Test Analyst, including single items, pairs, triples or more [Copeland03]. There are a number of tools available to aid the Test Analyst in this task (see www.pairwise.org for samples). These tools either require the parameters and their values to be listed (pairwise testing and orthogonal array testing) or to be represented in a graphical form (classification trees) [Grochtmann94]. Pairwise testing is a method applied to testing pairs of variables in combination. Orthogonal arrays are predefined, mathematically accurate tables that allow the Test Analyst to substitute the items to be tested for the variables in the array, producing a set of combinations that will achieve a level of coverage when tested [Koomen06]. Classification tree tools allow the Test Analyst to define the size of combinations to be tested (i.e., combinations of two values, three values, etc.).

Applicability
The problem of having too many combinations of parameter values manifests in at least two different situations related to testing. Some test cases contain several parameters each with a number of possible values, for instance a screen with several input fields. In this case, combinations of parameter values make up the input data for the test cases. Furthermore, some systems may be configurable in a number of dimensions, resulting in a potentially large configuration space. In both these situations, combinatorial testing can be used to identify a subset of combinations, feasible in size.

For parameters with a large number of values, equivalence class partitioning, or some other selection mechanism may first be applied to each parameter individually to reduce the number of values for each parameter, before combinatorial testing is applied to reduce the set of resulting combinations.

These techniques are usually applied to the integration, system and system integration levels of testing.

Limitations/Difficulties
The major limitation with these techniques is the assumption that the results of a few tests are representative of all tests and that those few tests represent expected usage. If there is an unexpected interaction between certain variables, it may go undetected with this type of testing if that particular combination is not tested. These techniques can be difficult to explain to a non-technical audience as they may not understand the logical reduction of tests.

Identifying the parameters and their respective values is sometimes difficult. Finding a minimal set of combinations to satisfy a certain level of coverage is difficult to do manually. Tools usually are used to find the minimum set of combinations. Some tools support the ability to force some (sub-) combinations to be included in or excluded from the final selection of combinations. This capability may be used by the Test Analyst to emphasize or de-emphasize factors based on domain knowledge or product usage information.

Coverage
There are several levels of coverage. The lowest level of coverage is called 1-wise or singleton coverage. It requires each value of every parameter be present in at least one of the chosen combinations. The next level of coverage is called 2-wise or pairwise coverage. It requires every pair of values of any two parameters be included in at least one combination. This idea can be generalized to n-wise coverage, which requires every sub-combination of values of any set of n parameters be included in the set of selected combinations. The higher the n, the more combinations needed to

reach 100% coverage. Minimum coverage with these techniques is to have one test case for every combination produced by the tool.

Types of Defects
The most common type of defects found with this type of testing is defects related to the combined values of several parameters.

3.2.7 Use Case Testing

Use case testing provides transactional, scenario-based tests that should emulate usage of the system. Use cases are defined in terms of interactions between the actors and the system that accomplish some goal. Actors can be users or external systems.

Applicability
Use case testing is usually applied at the system and acceptance testing levels. It may be used for integration testing depending on the level of integration and even component testing depending on the behavior of the component. Use cases are also often the basis for performance testing because they portray realistic usage of the system. The scenarios described in the use cases may be assigned to virtual users to create a realistic load on the system.

Limitations/Difficulties
In order to be valid, the use cases must convey realistic user transactions. This information should come from a user or a user representative. The value of a use case is reduced if the use case does not accurately reflect activities of the real user. An accurate definition of the various alternate paths (flows) is important for the testing coverage to be thorough. Use cases should be taken as a guideline, but not a complete definition of what should be tested as they may not provide a clear definition of the entire set of requirements. It may also be beneficial to create other models, such as flow charts, from the use case narrative to improve the accuracy of the testing and to verify the use case itself.

Coverage
Minimum coverage of a use case is to have one test case for the main (positive) path, and one test case for each alternate path or flow. The alternate paths include exception and failure paths. Alternate paths are sometimes shown as extensions of the main path. Coverage percentage is determined by taking the number of paths tested and dividing that by the total number of main and alternate paths.

Types of Defects
Defects include mishandling of defined scenarios, missed alternate path handling, incorrect processing of the conditions presented and awkward or incorrect error reporting.

3.2.8 User Story Testing

In some Agile methodologies, such as Scrum, requirements are prepared in the form of user stories which describe small functional units that can be designed, developed, tested and demonstrated in a single iteration [Cohn04]. These user stories include a description of the functionality to be implemented, any non-functional criteria, and also include acceptance criteria that must be met for the user story to be considered complete.

Applicability
User stories are used primarily in Agile and similar iterative and incremental environments. They are used for both functional testing and non-functional testing. User stories are used for testing at all levels with the expectation that the developer will demonstrate the functionality implemented for the user story prior to handoff of the code to the team members with the next level of testing tasks (e.g., integration, performance testing).

Limitations/Difficulties
Because stories are little increments of functionality, there may be a requirement to produce drivers and stubs in order to actually test the piece of functionality that is delivered. This usually requires an ability to program and to use tools that will help with the testing such as API testing tools. Creation of the drivers and stubs is usually the responsibility of the developer, although a Technical Test Analyst also may be involved in producing this code and utilizing the API testing tools. If a continuous integration model is used, as is the case in most Agile projects, the need for drivers and stubs is minimized.

Coverage
Minimum coverage of a user story is to verify that each of the specified acceptance criteria has been met.

Types of Defects
Defects are usually functional in that the software fails to provide the specified functionality. Defects are also seen with integration issues of the functionality in the new story with the functionality that already exists. Because stories may be developed independently, performance, interface and error handling issues may be seen. It is important for the Test Analyst to perform both testing of the individual functionality supplied as well as integration testing anytime a new story is released for testing.

3.2.9 Domain Analysis

A domain is a defined set of values. The set may be defined as a range of values of a single variable (a one-dimensional domain, e.g., "men aged over 24 and under 66"), or as ranges of values of interacting variables (a multi-dimensional domain, e.g., "men aged over 24 and under 66 AND with weight over 69 kg and under 90 kg"). Each test case for a multi-dimensional domain must include appropriate values for each variable involved.

Domain analysis of a one-dimensional domain typically uses equivalence partitioning and boundary value analysis. Once the partitions are defined, the Test Analyst selects values from each partition that represent a value that is in the partition (IN), outside the partition (OUT), on the boundary of the partition (ON) and just off the boundary of the partition (OFF). By determining these values, each partition is tested along with its boundary conditions. [Black07]

With multi-dimensional domains the number of test cases generated by these methods rises exponentially with the number of variables involved, whereas an approach based on domain theory leads to a linear growth. Also, because the formal approach incorporates a theory of defects (a fault model), which equivalence partitioning and boundary value analysis lack, its smaller test set will find defects in multi-dimensional domains that the larger, heuristic test set would likely miss. When dealing with multi-dimensional domains, the test model may be constructed as a decision table (or "domain matrix"). Identifying test case values for multi-dimensional domains above three dimensions is likely to require computational support.

Applicability
Domain analysis combines the techniques used for decision tables, equivalence partitioning and boundary value analysis to create a smaller set of tests that still cover the important areas and the likely areas of failure. It is often applied in cases where decision tables would be unwieldy because of the large number of potentially interacting variables. Domain analysis can be done at any level of testing but is most frequently applied at the integration and system testing levels.

Limitations/Difficulties
Doing thorough domain analysis requires a strong understanding of the software in order to identify the various domains and potential interaction between the domains. If a domain is left unidentified, the testing can be significantly lacking, but it is likely that the domain will be detected because the OFF and OUT variables may land in the undetected domain. Domain analysis is a strong technique to use when working with a developer to define the testing areas.

Coverage
Minimum coverage for domain analysis is to have a test for each IN, OUT, ON and OFF value in each domain. Where there is an overlap of the values (for example, the OUT value of one domain is an IN value in another domain), there is no need to duplicate the tests. Because of this, the actual tests needed are often less than four per domain.

Types of Defects
Defects include functional problems within the domain, boundary value handling, variable interaction issues and error handling (particularly for the values that are not in a valid domain).

3.2.10 Combining Techniques

Sometimes techniques are combined to create test cases. For example, the conditions identified in a decision table might be subjected to equivalence partitioning to discover multiple ways in which a condition might be satisfied. Test cases would then cover not only every combination of conditions, but also, for those conditions which were partitioned, additional test cases would be generated to cover the equivalence partitions. When selecting the particular technique to be applied, the Test Analyst should consider the applicability of the technique, the limitations and difficulties, and the goals of the testing in terms of coverage and defects to be detected. There may not be a single “best” technique for a situation. Combined techniques will often provide the most complete coverage assuming there is sufficient time and skill to correctly apply the techniques.

3.3 Defect-Based Techniques

3.3.1 Using Defect-Based Techniques

A defect-based test design technique is one in which the type of defect sought is used as the basis for test design, with tests derived systematically from what is known about the type of defect. Unlike specification-based testing which derives its tests from the specification, defect-based testing derives tests from defect taxonomies (i.e., categorized lists) that may be completely independent from the software being tested. The taxonomies can include lists of defect types, root causes, failure symptoms and other defect-related data. Defect-based testing may also use lists of identified risks and risk scenarios as a basis for targeting testing. This test technique allows the tester to target a specific type of defect or to work systematically through a defect taxonomy of known and common defects of a particular type. The Test Analyst uses the taxonomy data to determine the goal of the testing, which is to find a specific type of defect. From this information, the Test Analyst will create the test cases and test conditions that will cause the defect to manifest itself, if it exists.

Applicability
Defect-based testing can be applied at any testing level but is most commonly applied during system testing. There are standard taxonomies that apply to multiple types of software. This non-product specific type of testing helps to leverage industry standard knowledge to derive the particular tests. By adhering to industry-specific taxonomies, metrics regarding defect occurrence can be tracked across projects and even across organizations.

Limitations/Difficulties

Multiple defect taxonomies exist and may be focused on particular types of testing, such as usability. It is important to pick a taxonomy that is applicable to the software being tested, if any are available. For example, there may not be any taxonomies available for innovative software. Some organizations have compiled their own taxonomies of likely or frequently seen defects. Whatever taxonomy is used, it is important to define the expected coverage prior to starting the testing.

Coverage
The technique provides coverage criteria which are used to determine when all the useful test cases have been identified. As a practical matter, the coverage criteria for defect-based techniques tend to be less systematic than for specification-based techniques in that only general rules for coverage are given and the specific decision about what constitutes the limit of useful coverage is discretionary. As with other techniques, the coverage criteria do not mean that the entire set of tests is complete, but rather that defects being considered no longer suggest any useful tests based on that technique.

Types of Defects
The types of defects discovered usually depend on the taxonomy in use. If a user interface taxonomy is used, the majority of the discovered defects would likely be user interface related, but other defects can be discovered as a byproduct of the specific testing.

3.3.2 Defect Taxonomies

Defect taxonomies are categorized lists of defect types. These lists can be very general and used to serve as high-level guidelines or can be very specific. For example, a taxonomy for user interface defects could contain general items such as functionality, error handling, graphics display and performance. A detailed taxonomy could include a list of all possible user interface objects (particularly for a graphical user interface) and could designate the improper handling of these objects, such as:

- Text field
  - Valid data is not accepted
  - Invalid data is accepted
  - Length of input is not verified
  - Special characters are not detected
  - User error messages are not informative
  - User is not able to correct erroneous data
  - Rules are not applied
- Date field
  - Valid dates are not accepted
  - Invalid dates are not rejected
  - Date ranges are not verified
  - Precision data is not handled correctly (e.g., hh:mm:ss)
  - User is not able to correct erroneous data
  - Rules are not applied (e.g., ending date must be greater than starting date)

There are many defect taxonomies available, ranging from formal taxonomies that can be purchased to those designed for specific purposes by various organizations. Internally developed defect taxonomies can also be used to target specific defects commonly found within the organization. When creating a new defect taxonomy or customizing one that is available, it is important to first define the goals or objectives of the taxonomy. For example, the goal might be to identify user interface issues that have been discovered in production systems or to identify issues related to the handling of input fields.

To create a taxonomy:

1. Create a goal and define the desired level of detail
2. Select a given taxonomy to use as a basis

3. Define values and common defects experienced in the organization and/or from practice outside

The more detailed the taxonomy, the more time it will take to develop and maintain it, but it will result in a higher level of reproducibility in the test results. Detailed taxonomies can be redundant, but they allow a test team to divide up the testing without a loss of information or coverage.

Once the appropriate taxonomy has been selected, it can be used for creating test conditions and test cases. A risk-based taxonomy can help the testing focus on a specific risk area. Taxonomies can also be used for non-functional areas such as usability, performance, etc. Taxonomy lists are available in various publications, from IEEE, and on the Internet.

3.4 Experience-Based Techniques

Experience-based tests utilize the skill and intuition of the testers, along with their experience with similar applications or technologies. These tests are effective at finding defects but not as appropriate as other techniques to achieve specific test coverage levels or produce reusable test procedures. In cases where system documentation is poor, testing time is severely restricted or the test team has strong expertise in the system to be tested, experience-based testing may be a good alternative to more structured approaches. Experience-based testing may be inappropriate in systems requiring detailed test documentation, high-levels of repeatability or an ability to precisely assess test coverage.

When using dynamic and heuristic approaches, testers normally use experience-based tests, and testing is more reactive to events than pre-planned testing approaches. In addition execution and evaluation are concurrent tasks. Some structured approaches to experience-based tests are not entirely dynamic, i.e., the tests are not created entirely at the same time as the tester executes the test.

Note that although some ideas on coverage are presented for the techniques discussed here, experience-based techniques do not have formal coverage criteria.

3.4.1 Error Guessing

When using the error guessing technique, the Test Analyst uses experience to guess the potential errors that might have been made when the code was being designed and developed. When the expected errors have been identified, the Test Analyst then determines the best methods to use to uncover the resulting defects. For example, if the Test Analyst expects the software will exhibit failures when an invalid password is entered, tests will be designed to enter a variety of different values in the password field to verify if the error was indeed made and has resulted in a defect that can be seen as a failure when the tests are run.

In addition to being used as a testing technique, error guessing is also useful during risk analysis to identify potential failure modes. [Myers79]

Applicability
Error guessing is done primarily during integration and system testing, but can be used at any level of testing. This technique is often used with other techniques and helps to broaden the scope of the existing test cases. Error guessing can also be used effectively when testing a new release of the software to test for common mistakes and errors before starting more rigorous and scripted testing. Checklists and taxonomies may be helpful in guiding the testing.

Limitations/Difficulties
Coverage is difficult to assess and varies widely with the capability and experience of the Test Analyst. It is best used by an experienced tester who is familiar with the types of defects that are commonly

introduced in the type of code being tested. Error guessing is commonly used, but is frequently not documented and so may be less reproducible than other forms of testing.

Coverage
When a taxonomy is used, coverage is determined by the appropriate data faults and types of defects. Without a taxonomy, coverage is limited by the experience and knowledge of the tester and the time available. The yield from this technique will vary based on how well the tester can target problematic areas.

Types of Defects
Typical defects are usually those defined in the particular taxonomy or "guessed" by the Test Analyst, that might not have been found in specification-based testing.

3.4.2 Checklist-Based Testing

When applying the checklist-based testing technique, the experienced Test Analyst uses a high-level, generalized list of items to be noted, checked, or remembered, or a set of rules or criteria against which a product has to be verified. These checklists are built based on a set of standards, experience, and other considerations. A user interface standards checklist employed as the basis for testing an application is an example of a checklist-based test.

Applicability
Checklist-based testing is used most effectively in projects with an experienced test team that is familiar with the software under test or familiar with the area covered by the checklist (e.g., to successfully apply a user interface checklist, the Test Analyst may be familiar with user interface testing but not the specific software under test). Because checklists are high-level and tend to lack the detailed steps commonly found in test cases and test procedures, the knowledge of the tester is used to fill in the gaps. By removing the detailed steps, checklists are low maintenance and can be applied to multiple similar releases. Checklists can be used for any level of testing. Checklists are also used for regression testing and smoke testing.

Limitations/Difficulties
The high-level nature of the checklists can affect the reproducibility of test results. It is possible that several testers will interpret the checklists differently and will follow different approaches to fulfil the checklist items. This may cause different results, even though the same checklist is used. This can result in wider coverage but reproducibility is sometimes sacrificed. Checklists may also result in over-confidence regarding the level of coverage that is achieved since the actual testing depends on the tester's judgment. Checklists can be derived from more detailed test cases or lists and tend to grow over time. Maintenance is required to ensure that the checklists are covering the important aspects of the software being tested.

Coverage
The coverage is as good as the checklist but, because of the high-level nature of the checklist, the results will vary based on the Test Analyst who executes the checklist.

Types of Defects
Typical defects found with this technique include failures resulting from varying the data, the sequence of steps or the general workflow during testing. Using checklists can help keep the testing fresh as new combinations of data and processes are allowed during testing.

3.4.3 Exploratory Testing

Exploratory testing is characterized by the tester simultaneously learning about the product and its defects, planning the testing work to be done, designing and executing the tests, and reporting the

results. The tester dynamically adjusts test goals during execution and prepares only lightweight documentation. [Whittaker09]

Applicability
Good exploratory testing is planned, interactive, and creative. It requires little documentation about the system to be tested and is often used in situations where the documentation is not available or is not adequate for other testing techniques. Exploratory testing is often used to augment other testing and to serve as a basis for the development of additional test cases.

Limitations/Difficulties
Exploratory testing can be difficult to manage and schedule. Coverage can be sporadic and reproducibility is difficult. Using charters to designate the areas to be covered in a testing session and time-boxing to determine the time allowed for the testing is one method used to manage exploratory testing. At the end of a testing session or set of sessions, the test manager may hold a debriefing session to gather the results of the tests and determine the charters for the next sessions. Debriefing sessions are difficult to scale for large testing teams or large projects.

Another difficulty with exploratory sessions is to accurately track them in a test management system. This is sometimes done by creating test cases that are actually exploratory sessions. This allows the time allocated for the exploratory testing and the planned coverage to be tracked with the other testing efforts.

Since reproducibility may be difficult with exploratory testing, this can also cause problems when needing to recall the steps to reproduce a failure. Some organizations use the capture/playback capability of a test automation tool to record the steps taken by an exploratory tester. This provides a complete record of all activities during the exploratory session (or any experience-based testing session). Digging through the details to find the actual cause of the failure can be tedious, but at least there is a record of all the steps that were involved.

Coverage
Charters may be created to specify tasks, objectives, and deliverables. Exploratory sessions are then planned to achieve those objectives. The charter may also identify where to focus the testing effort, what is in and out of scope of the testing session, and what resources should be committed to complete the planned tests. A session may be used to focus on particular defect types and other potentially problematic areas that can be addressed without the formality of scripted testing.

Types of Defects
Typical defects found with exploratory testing are scenario-based issues that were missed during scripted functional testing, issues that fall between functional boundaries, and workflow related issues. Performance and security issues are also sometimes uncovered during exploratory testing.

3.4.4 Applying the Best Technique

Defect- and experience-based techniques require the application of knowledge about defects and other testing experiences to target testing in order to increase defect detection. They range from “quick tests” in which the tester has no formally pre-planned activities to perform, through pre-planned sessions to scripted sessions. They are almost always useful but have particular value in the following circumstances:

- No specifications are available
- There is poor documentation of the system under test
- Insufficient time is allowed to design and create detailed tests
- Testers are experienced in the domain and/or the technology
- Diversity from scripted testing is a goal to maximize test coverage
- Operational failures are to be analyzed

Defect- and experience-based techniques are also useful when used in conjunction with specification-based techniques, as they fill the gaps in test coverage that result from systematic weaknesses in these techniques. As with the specification-based techniques, there is not one perfect technique for all situations. It is important for the Test Analyst to understand the advantages and disadvantages of each technique and to be able to select the best technique or set of techniques for the situation, considering the project type, schedule, access to information, skills of the tester and other factors that can influence the selection.

4. Testing Software Quality Characteristics - 120 mins.

Keywords

accessibility testing, accuracy testing, attractiveness, heuristic evaluation, interoperability testing, learnability, operability, suitability testing, SUMI, understandability, usability testing, WAMMI

Learning Objectives for Testing Software Quality Characteristics

4.2 Quality Characteristics for Business Domain Testing

TA-4.2.1 (K2) Explain by example what testing techniques are appropriate to test accuracy, suitability, interoperability and compliance characteristics

TA-4.2.2 (K2) For the accuracy, suitability and interoperability characteristics, define the typical defects to be targeted

TA-4.2.3 (K2) For the accuracy, suitability and interoperability characteristics, define when the characteristic should be tested in the lifecycle

TA-4.2.4 (K4) For a given project context, outline the approaches that would be suitable to verify and validate both the implementation of the usability requirements and the fulfillment of the user's expectations

4.1 Introduction

While the previous chapter described specific techniques available to the tester, this chapter considers the application of those techniques in evaluating the principal characteristics used to describe the quality of software applications or systems.

This syllabus discusses the quality characteristics which may be evaluated by a Test Analyst. The attributes to be evaluated by the Technical Test Analyst are considered in the Advanced Technical Test Analyst syllabus. The description of product quality characteristics provided in ISO 9126 is used as a guide to describing the characteristics. Other standards, such as the ISO 25000 [ISO25000] series (which has superseded ISO 9126) may also be of use. The ISO quality characteristics are divided into product quality characteristics (attributes), each of which may have sub-characteristics (sub-attributes). These are shown in the table below, together with an indication of which characteristics/sub-characteristics are covered by the Test Analyst and Technical Test Analyst syllabi:

Characteristic	Sub-Characteristics	Test Analyst	Technical Test Analyst
Functionality	Accuracy, suitability, interoperability, compliance	X	
	Security		X
Reliability	Maturity (robustness), fault-tolerance, recoverability, compliance		X
Usability	Understandability, learnability, operability, attractiveness, compliance	X	
Efficiency	Performance (time behavior), resource utilization, compliance		X
Maintainability	Analyzability, changeability, stability, testability, compliance		X
Portability	Adaptability, installability, co-existence, replaceability, compliance		X

The Test Analyst should concentrate on the software quality characteristics of functionality and usability. Accessibility testing should also be conducted by the Test Analyst. Although it is not listed as a sub-characteristic, accessibility is often considered to be part of usability testing. Testing for the other quality characteristics is usually considered to be the responsibility of the Technical Test Analyst. While this allocation of work may vary in different organizations, it is the one that is followed in these ISTQB syllabi.

The sub-characteristic of compliance is shown for each of the quality characteristics. In the case of certain safety-critical or regulated environments, each quality characteristic may have to comply with specific standards and regulations (e.g., functionality compliance may indicate that the functionality comply with a specific standard such as using a particular communication protocol in order to be able to send/receive data from a chip). Because those standards can vary widely depending on the industry, they will not be discussed in depth here. If the Test Analyst is working in an environment that is affected by compliance requirements, it is important to understand those requirements and to ensure that both the testing and the test documentation will fulfill the compliance requirements.

For all of the quality characteristics and sub-characteristics discussed in this section, the typical risks must be recognized so that an appropriate testing strategy can be formed and documented. Quality characteristic testing requires particular attention to lifecycle timing, required tools, software and documentation availability, and technical expertise. Without planning a strategy to deal with each characteristic and its unique testing needs, the tester may not have adequate planning, ramp up and test execution time built into the schedule. Some of this testing, e.g., usability testing, can require

allocation of special human resources, extensive planning, dedicated labs, specific tools, specialized testing skills and, in most cases, a significant amount of time. In some cases, usability testing may be performed by a separate group of usability, or user experience, experts.

Quality characteristic and sub-characteristic testing must be integrated into the overall testing schedule, with adequate resources allocated to the effort. Each of these areas has specific needs, targets specific issues and may occur at different times during the software development lifecycle, as discussed in the sections below.

While the Test Analyst may not be responsible for the quality characteristics that require a more technical approach, it is important that the Test Analyst be aware of the other characteristics and understand the overlap areas for testing. For example, a product that fails performance testing will also likely fail in usability testing if it is too slow for the user to use effectively. Similarly, a product with interoperability issues with some components is probably not ready for portability testing as that will tend to obscure the more basic problems when the environment is changed.

4.2 Quality Characteristics for Business Domain Testing

Functional testing is a primary focus for the Test Analyst. Functional testing is focused on "what" the product does. The test basis for functional testing is generally a requirements or specification document, specific domain expertise or implied need. Functional tests vary according to the test level in which they are conducted and can also be influenced by the software development lifecycle. For example, a functional test conducted during integration testing will test the functionality of interfacing modules which implement a single defined function. At the system test level, functional tests include testing the functionality of the application as a whole. For systems of systems, functional testing will focus primarily on end to end testing across the integrated systems. In an Agile environment, functional testing is usually limited to the functionality made available in the particular iteration or sprint although regression testing for an iteration may cover all released functionality.

A wide variety of test techniques are employed during functional test (see Chapter 3). Functional testing may be performed by a dedicated tester, a domain expert, or a developer (usually at the component level).

In addition to the functional testing covered in this section, there are also two quality characteristics that are a part of the Test Analyst's area of responsibility that are considered to be non-functional (focused on "how" the product delivers the functionality) testing areas. These two non-functional attributes are usability and accessibility.

The following quality characteristics are considered in this section:

- Functional quality sub-characteristics
  - Accuracy
  - Suitability
  - Interoperability
- Non-functional quality characteristics
  - Usability
  - Accessibility

4.2.1 Accuracy Testing

Functional accuracy involves testing the application's adherence to the specified or implied requirements and may also include computational accuracy. Accuracy testing employs many of the test techniques explained in Chapter 3 and often uses the specification or a legacy system as the test oracle. Accuracy testing can be conducted at any stage in the lifecycle and is targeted on incorrect handling of data or situations.

4.2.2 Suitability Testing

Suitability testing involves evaluating and validating the appropriateness of a set of functions for its intended specified tasks. This testing can be based on use cases. Suitability testing is usually conducted during system testing, but may also be conducted during the later stages of integration testing. Defects discovered in this testing are indications that the system will not be able to meet the needs of the user in a way that will be considered acceptable.

4.2.3 Interoperability Testing

Interoperability testing tests the degree to which two or more systems or components can exchange information and subsequently use the information that has been exchanged. Testing must cover all the intended target environments (including variations in the hardware, software, middleware, operating system, etc.) to ensure the data exchange will work properly. In reality, this may only be feasible for a relatively small number of environments. In that case interoperability testing may be limited to a selected representative group of environments. Specifying tests for interoperability requires that combinations of the intended target environments are identified, configured and available to the test team. These environments are then tested using a selection of functional test cases which exercise the various data exchange points present in the environment.

Interoperability relates to how different software systems interact with each other. Software with good interoperability characteristics can be integrated with a number of other systems without requiring major changes. The number of changes and the effort required to perform those changes may be used as a measure of interoperability.

Testing for software interoperability may, for example, focus on the following design features:

- Use of industry-wide communications standards, such as XML
- Ability to automatically detect the communications needs of the systems it interacts with and adjust accordingly

Interoperability testing may be particularly significant for organizations developing Commercial Off The Shelf (COTS) software and tools and organizations developing systems of systems.

This type of testing is performed during component integration and system testing focusing on the interaction of the system with its environment. At the system integration level, this type of testing is conducted to determine how well the fully developed system interacts with other systems. Because systems may interoperate on multiple levels, the Test Analyst must understand these interactions and be able to create the conditions that will exercise the various interactions. For example, if two systems will exchange data, the Test Analyst must be able to create the necessary data and the transactions required to perform the data exchange. It is important to remember that all interactions may not be clearly specified in the requirements documents. Instead, many of these interactions will be defined only in the system architecture and design documents. The Test Analyst must be able and prepared to examine those documents to determine the points of information exchange between systems and between the system and its environment to ensure all are tested. Techniques such as decision tables, state transition diagrams, use cases and combinatorial testing are all applicable to interoperability testing. Typical defects found include incorrect data exchange between interacting components.

4.2.4 Usability Testing

It is important to understand why users might have difficulty using the system. To gain this understanding it is first necessary to appreciate that the term “user” may apply to a wide range of different types of persons, ranging from IT experts to children to people with disabilities.

Some national institutions (e.g., the British Royal National Institute for the Blind), recommend that web pages be accessible for disabled, blind, partially sighted, mobility impaired, deaf and cognitively-

disabled users. Checking that applications and web sites are usable for the above users may also improve the usability for everyone else. Accessibility is discussed more below.

Usability testing tests the ease by which users can use or learn to use the system to reach a specified goal in a specific context. Usability testing is directed at measuring the following:

- Effectiveness - capability of the software product to enable users to achieve specified goals with accuracy and completeness in a specified context of use
- Efficiency - capability of the product to enable users to expend appropriate amounts of resources in relation to the effectiveness achieved in a specified context of use
- Satisfaction - capability of the software product to satisfy users in a specified context of use

Attributes that may be measured include:

- Understandability - attributes of the software that affect the effort required by the user to recognize the logical concept and its applicability
- Learnability - attributes of software that affect the effort required by the user to learn the application
- Operability - attributes of the software that affect the effort required by the user to conduct tasks effectively and efficiently
- Attractiveness - the capability of the software to be liked by the user

Usability testing is usually conducted in two steps:

- Formative Usability Testing - testing that is conducted iteratively during the design and prototyping stages to help guide (or "form") the design by identifying usability design defects
- Summative Usability Testing - testing that is conducted after implementation to measure the usability and identify problems with a completed component or system

Usability tester skills should include expertise or knowledge in the following areas:

- Sociology
- Psychology
- Conformance to national standards (including accessibility standards)
- Ergonomics

4.2.4.1 Conducting Usability Tests

Validation of the actual implementation should be done under conditions as close as possible to those under which the system will be used. This may involve setting up a usability lab with video cameras, mock up offices, review panels, users, etc., so that development staff can observe the effect of the actual system on real people. Formal usability testing often requires some amount of preparing the “users” (these could be real users or user representatives) either by providing set scripts or instructions for them to follow. Other free form tests allow the user to experiment with the software so the observers can determine how easy or difficult it is for the user to figure out how to accomplish their tasks.

Many usability tests may be executed by the Test Analyst as part of other tests, for example during functional system test. To achieve a consistent approach to the detection and reporting of usability defects in all stages of the lifecycle, usability guidelines may be helpful. Without usability guidelines, it may be difficult to determine what is “unacceptable” usability. For example, is it unreasonable for a user to have to make 10 mouse clicks to log into an application? Without specific guidelines, the Test Analyst can be in the difficult position of defending defect reports that the developer wants to close because the software works “as designed”. It is very important to have the verifiable usability specifications defined in the requirements as well as to have a set of usability guidelines that are applied to all similar projects. The guidelines should include such items as accessibility of instructions, clarity of prompts, number of clicks to complete an activity, error messaging, processing indicators (some type of indicator for the user that the system is processing and cannot accept further inputs at

the time), screen layout guidelines, use of colors and sounds and other factors that affect the user's experience.

4.2.4.2 Usability Test Specification

Principal techniques for usability testing are:

- Inspecting, evaluating or reviewing
- Dynamically interacting with prototypes
- Verifying and validating the actual implementation
- Conducting surveys and questionnaires

Inspecting, evaluating or reviewing

Inspection or review of the requirements specification and designs from a usability perspective that increase the user's level of involvement can be cost effective by finding problems early. Heuristic evaluation (systematic inspection of a user interface design for usability) can be used to find the usability problems in the design so that they can be attended to as part of an iterative design process. This involves having a small set of evaluators examine the interface and judge its compliance with recognized usability principles (the "heuristics"). Reviews are more effective when the user interface is more visible. For example, sample screen shots are usually easier to understand and interpret than a narrative description of the functionality provided by a particular screen. Visualization is important for an adequate usability review of the documentation.

Dynamically interacting with prototypes

When prototypes are developed, the Test Analyst should work with the prototypes and help the developers evolve the prototype by incorporating user feedback into the design. In this way, prototypes can be refined and the user can get a more realistic view of how the finished product will look and feel.

Verifying and validating the actual implementation

Where the requirements specify usability characteristics for the software (e.g., the number of mouse clicks to accomplish a specific goal), test cases should be created to verify that the software implementation has included these characteristics.

For performing validation of the actual implementation, tests specified for functional system test may be developed as usability test scenarios. These test scenarios measure specific usability characteristics, such as learnability or operability, rather than functional outcomes.

Test scenarios for usability may be developed to specifically test syntax and semantics. Syntax is the structure or grammar of the interface (e.g., what can be entered in an input field) whereas semantics describes the meaning and purpose (e.g., reasonable and meaningful system messages and output provided to the user) of the interface.

Black box techniques (for example those described in Section 3.2), particularly use cases which can be defined in plain text or with UML (Unified Modeling Language), are sometimes employed in usability testing.

Test scenarios for usability testing also need to include user instructions, allocation of time for pre- and post-test interviews for giving instructions and receiving feedback and an agreed protocol for conducting the sessions. This protocol includes a description of how the test will be carried out, timings, note taking and session logging, and the interview and survey methods to be used.

Conducting surveys and questionnaires

Survey and questionnaire techniques may be applied to gather observations and feedback regarding user behavior with the system. Standardized and publicly available surveys such as SUMI (Software Usability Measurement Inventory) and WAMMI (Website Analysis and MeasureMent Inventory) permit benchmarking against a database of previous usability measurements. In addition, since SUMI

provides concrete measurements of usability, this can provide a set of completion / acceptance criteria.

4.2.5 Accessibility Testing

It is important to consider the accessibility to software for those with particular needs or restrictions for its use. This includes those with disabilities. Accessibility testing should consider the relevant standards, such as the Web Content Accessibility Guidelines, and legislation, such as Disability Discrimination Acts (UK, Australia) and Section 508 (US). Accessibility, similar to usability, must be considered during the design phases. Testing often occurs during the integration levels and continues through system testing and into the acceptance testing levels. Defects are usually determined when the software fails to meet the designated regulations or standards defined for the software.

5. Reviews - 165 mins.

Keywords
none

Learning Objectives for Reviews

5.1 Introduction
TA-5.1.1 (K2) Explain why review preparation is important for the Test Analyst

5.2 Using Checklists in Reviews
TA-5.2.1 (K4) Analyze a use case or user interface and identify problems according to checklist information provided in the syllabus
TA-5.2.2 (K4) Analyze a requirements specification or user story and identify problems according to checklist information provided in the syllabus

5.1 Introduction

A successful review process requires planning, participation and follow-up. Test Analysts must be active participants in the review process, providing their unique views. They should have formal review training to better understand their respective roles in any review process. All review participants must be committed to the benefits of a well-conducted review. When done properly, reviews can be the single biggest, and most cost-effective, contributor to overall delivered quality.

Regardless of the type of review being conducted, the Test Analyst must allow adequate time to prepare. This includes time to review the work product, time to check cross-referenced documents to verify consistency, and time to determine what might be missing from the work product. Without adequate preparation time, the Test Analyst could be restricted only to editing what is already in the document rather than participating in an efficient review that maximizes the use of the review team's time and provides the best feedback possible. A good review includes understanding what is written, determining what is missing, and verifying that the described product is consistent with other products that are either already developed or are in development. For example, when reviewing an integration level test plan, the Test Analyst must also consider the items that are being integrated. What are the conditions needed for them to be ready for integration? Are there dependencies that must be documented? Is there data available to test the integration points? A review is not isolated to the work product being reviewed; it must also consider the interaction of that item with the others in the system.

It is easy for the author of a product being reviewed to feel criticized. The Test Analyst should be sure to approach any review comments from the view point of working together with the author to create the best product possible. By using this approach, comments will be worded constructively and will be oriented toward the work product and not the author. For example, if a statement is ambiguous, it is better to say "I do not understand what I should be testing to verify that this requirement has been implemented correctly. Can you help me understand it?" rather than "This requirement is ambiguous and no one will be able to figure it out." The Test Analyst's job in a review is to ensure that the information provided in the work product will be sufficient to support the testing effort. If the information is not there, is not clear, or does not provide the necessary level of detail, then this is likely to be a defect that needs to be corrected by the author. By maintaining a positive approach rather than a critical approach, comments will be better received and the meeting will be more productive.

5.2 Using Checklists in Reviews

Checklists are used during reviews to remind the participants to check specific points during the review. Checklists can also help to de-personalize the review, e.g., "This is the same checklist we use for every review, we are not targeting only your work product." Checklists can be generic and used for all reviews or can focus on specific quality characteristics, areas or types of documents. For example, a generic checklist might verify the general document properties such as having a unique identifier, no TBD references, proper formatting and similar conformance items. A specific checklist for a requirements document might contain checks for the proper use of the terms "shall" and "should", checks for the testability of each stated requirement and so forth. The format of the requirements may also indicate the type of checklist to be used. A requirements document that is in narrative text format will have different review criteria than one that is based on diagrams.

Checklists may also be oriented toward a programmer/architect skill set or a tester skill set. In the case of the Test Analyst, the tester skill set checklist would be the most appropriate. These checklists might include such items as shown below.

Checklists used for the requirements, use cases and user stories generally have a different focus than those used for the code or architecture. These requirements-oriented checklists could include the following items:

- Testability of each requirement
- Acceptance criteria for each requirement
- Availability of a use case calling structure, if applicable
- Unique identification of each requirement/use case/user story
- Versioning of each requirement/use case/user story
- Traceability for each requirement from business/marketing requirements
- Traceability between requirements and use cases

The above is meant only to serve as an example. It is important to remember that if a requirement is not testable, meaning that it is defined in such a way that the Test Analyst cannot determine how to test it, then there is a defect in that requirement. For example, a requirement that states "The software should be very user friendly" is untestable. How can the Test Analyst determine if the software is user friendly, or even very user friendly? If, instead, the requirement said "The software must conform to the usability standards stated in the usability standards document", and if the usability standards document really exists, then this is a testable requirement. It is also an overarching requirement because this one requirement applies to every item in the interface. In this case, this one requirement could easily spawn many individual test cases in a non-trivial application. Traceability from this requirement, or perhaps from the usability standards document, to the test cases is also critical because if the referenced usability specification should change, all the test cases will need to be reviewed and updated as needed.

A requirement is also untestable if the tester is unable to determine whether the test passed or failed, or is unable to construct a test that can pass or fail. For example, "System shall be available 100% of the time, 24 hours per day, 7 days per week, 365 (or 366) days a year" is untestable.

.

A simple checklist for use case reviews may include the following questions:

- Is the main path (scenario) clearly defined?
- Are all alternative paths (scenarios) identified, complete with error handling?
- Are the user interface messages defined?
- Is there only one main path (there should be, otherwise there are multiple use cases)?
- Is each path testable?

A simple checklist for usability for a user interface of an application may include:

- Is each field and its function defined?
- Are all error messages defined?
- Are all user prompts defined and consistent?
- Is the tab order of the fields defined?
- Are there keyboard alternatives to mouse actions?
- Are there "shortcut" key combinations defined for the user (e.g., cut and paste)?
- Are there dependencies between fields (such as a certain date has to be later than another date)?
- Is there a screen layout?
- Does the screen layout match the specified requirements?
- Is there an indicator for the user that appears when the system is processing?
- Does the screen meet the minimum mouse click requirement (if defined)?
- Does the navigation flow logically for the user based on use case information?
- Does the screen meet any requirements for learnability?
- Is there help text available for the user?
- Is there hover text available for the user?
- Will the user consider this to be "attractive" (subjective assessment)?

- Is the use of colors consistent with other applications and with organization standards?
- Are the sound effects used appropriately and are they configurable?
- Does the screen meet localization requirements?
- Can the user determine what to do (understandability) (subjective assessment)?
- Will the user be able to remember what to do (learnability) (subjective assessment)?

In an Agile project, requirements usually take the form of user stories. These stories represent small units of demonstrable functionality. Whereas a use case is a user transaction that traverses multiple areas of functionality, a user story is more isolated and is generally scoped by the time it takes to develop it. A checklist for a user story may include:

- Is the story appropriate for the target iteration/sprint?
- Are the acceptance criteria defined and testable?
- Is the functionality clearly defined?
- Are there any dependencies between this story and others?
- Is the story prioritized?
- Does the story contain one item of functionality?

Of course if the story defines a new interface, then using a generic story checklist (such as the one above) and a detailed user interface checklist would be appropriate.

A checklist can be tailored based on the following:

- Organization (e.g., considering company policies, standards, conventions)
- Project / development effort (e.g., focus, technical standards, risks)
- Object being reviewed (e.g., code reviews might be tailored to specific programming languages)

Good checklists will find problems and will also help to start discussions regarding other items that might not have been specifically referenced in the checklist. Using a combination of checklists is a strong way to ensure a review achieves the highest quality work product. Using standard checklists such as those referenced in the Foundation Level syllabus and developing organizationally specific checklists such as the ones shown above will help the Test Analyst be effective in reviews.

For more information on reviews and inspections see [Gilb93] and [Wiegers03].

6. Defect Management – 120 mins.

Keywords

Defect taxonomy, phase containment, root cause analysis

Learning Objectives for Defect Management

6.2 When Can a Defect be Detected?

TA-6.2.1 (K2) Explain how phase containment can reduce costs

6.3 Defect Report Fields

TA-6.3.1 (K2) Explain the information that may be needed when documenting a non-functional defect

6.4 Defect Classification

TA-6.4.1 (K4) Identify, gather and record classification information for a given defect

6.5 Root Cause Analysis

TA-6.5.1 (K2) Explain the purpose of root cause analysis

6.1 Introduction

Test Analysts evaluate the behavior of the system in terms of business and user needs, e.g., would the user know what to do when faced with this message or behavior. By comparing the actual with the expect result, the Test Analyst determines if the system is behaving correctly. An anomaly (also called an incident) is an unexpected occurrence that requires further investigation. An anomaly may be a failure caused by a defect. An anomaly may or may not result in the generation of a defect report. A defect is an actual problem that should be resolved.

6.2 When Can a Defect be Detected?

A defect can be detected through static testing and the symptoms of the defect, the failure, can be detected through dynamic testing. Each phase of the software development lifecycle should provide methods for detecting and eliminating potential failures. For example, during the development phase, code and design reviews should be used to detect defects. During dynamic testing, test cases are used to detect failures.

The earlier a defect is detected and corrected, the lower the cost of quality for the system as a whole. For example, static testing can find defects before dynamic testing is possible. This is one of the reasons why static testing is a cost effective approach to producing high quality software.

The defect tracking system should allow the Test Analyst to record the phase in the lifecycle in which the defect was introduced and the phase in which it was found. If the two phases are the same, then perfect phase containment has been achieved. This means the defect was introduced and found in the same phase and didn't "escape" to a later phase. An example of this would be an incorrect requirement that is identified during the requirements review and is corrected there. Not only is this an efficient use of the requirements review, but it also prevents that defect from incurring additional work which would make it more expensive for the organization. If an incorrect requirement "escapes" from the requirements review and is subsequently implemented by the developer, tested by the Test Analyst, and caught by the user during user acceptance testing, all the work done on that requirement was wasted time (not to mention that the user may now have lost confidence in the system).

Phase containment is an effective way to reduce the costs of defects.

6.3 Defect Report Fields

The fields (parameters) supplied for a defect report are intended to provide enough information so the defect report is actionable. An actionable defect report is:

- Complete - all the necessary information is in the report
- Concise - no extraneous information is in the report
- Accurate - the information in the report is correct and clearly states the expected and actual results as well as the proper steps to reproduce
- Objective - the report is a professionally written statement of fact

The information recorded in a defect report should be divided into fields of data. The more well-defined the fields, the easier it is to report individual defects as well as to produce trend reports and other summary reports. When a defined number of options are available for a field, having drop down lists of the available values can decrease the time needed when recording a defect. Drop down lists are only effective when the number of options is limited and the user will not need to scroll through a long list to find the correct option. Different types of defect reports require different information and the defect management tool should be flexible enough to prompt for the correct fields depending on the

defect type. Data should be recorded in distinct fields, ideally supported by data validation in order to avoid data entry failures, and to ensure effective reporting.

Defect reports are written for failures discovered during functional and non-functional testing. The information in a defect report should always be oriented toward clearly identifying the scenario in which the problem was detected, including steps and data required to reproduce that scenario, as well as the expected and actual results. Non-functional defect reports may require more details regarding the environment, other performance parameters (e.g., size of the load), sequence of steps and expected results. When documenting a usability failure, it is important to state what the user expected the software to do. For example, if the usability standard is that an operation should be completed in less than four mouse clicks, the defect report should state how many clicks were required versus the stated standard. In cases where a standard is not available and the requirements did not cover the non-functional quality aspects of the software, the tester may use the "reasonable person" test to determine that the usability is unacceptable. In that case, the expectations of that "reasonable person" must be clearly stated in the defect report. Because non-functional requirements are sometimes missing in the requirements documentation, documenting non-functional failures presents more challenges for the tester in documenting the "expected" versus the "actual" behavior.

While the usual goal in writing a defect report is to obtain a fix for the problem, the defect information must also be supplied to support accurate classification, risk analysis, and process improvement.

6.4 Defect Classification

There are multiple levels of classification that a defect report may receive throughout its lifecycle. Proper defect classification is an integral part of proper defect reporting. Classifications are used to group defects, to evaluate the effectiveness of testing, to evaluate the effectiveness of the development lifecycle and to determine interesting trends.

Common classification information for newly identified defects includes:

- Project activity that resulted in the defect being detected – e.g., review, audit, inspection, coding, testing
- Project phase in which the defect was introduced (if known) – e.g., requirements, design, detailed design, code
- Project phase in which the defect was detected – e.g., requirements, design, detailed design, code, code review, unit test, integration test, system test, acceptance test
- Suspected cause of defect – e.g., requirements, design, interface, code, data
- Repeatability – e.g., once, intermittent, reproducible
- Symptom – e.g., crash, hang, user interface error, system error, performance

Once the defect has been investigated, further classification may be possible:

- Root cause - the mistake that was made that resulted in the defect, e.g., process, coding error, user error, test error, configuration issue, data issue, third party software, external software problem, documentation issue
- Source – the work product in which the mistake made, e.g., requirements, design, detailed design, architecture, database design, user documentation, test documentation
- Type – e.g., logic problem, computational problem, timing issue, data handling, enhancement

When the defect is fixed (or has been deferred or has failed confirmation), even more classification information may be available, such as:

- Resolution – e.g., code change, documentation change, deferred, not a problem, duplicate
- Corrective action – e.g., requirements review, code review, unit test, configuration documentation, data preparation, no change made

In addition to these classification categories, defects are also frequently classified based on severity and priority. In addition, depending on the project, it may make sense to classify based on mission safety impact, project schedule impact, project costs, project risk and project quality impact. These classifications may be considered in agreements regarding how quickly a fix will be delivered.

The final area of classification is the final resolution. Defects are often grouped together based on their resolution, e.g., fixed/verified, closed/not a problem, deferred, open/unresolved. This classification usually is used throughout a project as the defects are tracked through their lifecycle.

The classification values used by an organization are often customized. The above are only examples of some of the common values used in industry. It is important that the classification values be used consistently in order to be useful. Too many classification fields will make opening and processing a defect somewhat time consuming, so it is important to weigh the value of the data being gathered against the incremental cost for every defect processed. The ability to customize the classification values gathered by a tool is often an important factor in tool selection.

6.5 Root Cause Analysis

The purpose of root cause analysis is to determine what caused the defect to occur and to provide data to support process changes that will remove root causes that are responsible for a significant portion of the defects. Root cause analysis is usually conducted by the person who investigates and either fixes the problem or determines the problem should not or cannot be fixed. This is usually the developer. Setting a preliminary root cause value is commonly done by the Test Analyst who will make an educated guess regarding what probably caused the problem. When confirming the fix, the Test Analyst will verify the root cause setting entered by the developer. At the point the root cause is determined, it is also common to determine or confirm the phase in which the defect was introduced.

Typical root causes include:

- Unclear requirement
- Missing requirement
- Wrong requirement
- Incorrect design implementation
- Incorrect interface implementation
- Code logic error
- Calculation error
- Hardware error
- Interface error
- Invalid data

This root cause information is aggregated to determine common issues that are resulting in the creation of defects. For example, if a large number of defects is caused by unclear requirements, it would make sense to apply more effort to conducting effective requirements reviews. Similarly if interface implementation is an issue across development groups, joint design sessions might be needed.

Using root cause information for process improvement helps an organization to monitor the benefits of effective process changes and to quantify the costs of the defects that can be attributed to a particular root cause. This can help provide funding for process changes that might require purchasing additional tools and equipment as well as changing schedule timing. The ISTQB Expert Level syllabus “Improving the Test Process” [ISTQB_EL_ITP] considers root cause analysis in more detail.

7. Test Tools - 45 mins.

Keywords

keyword-driven testing, test data preparation tool, test design tool, test execution tool

Learning Objectives for Test Tools

7.2 Test Tools and Automation

TA-7.2.1 (K2) Explain the benefits of using test data preparation tools, test design tools and test execution tools
TA-7.2.2 (K2) Explain the Test Analyst's role in keyword-driven automation
TA-7.2.3 (K2) Explain the steps for troubleshooting an automated test execution failure

7.1 Introduction

Test tools can greatly improve the efficiency and accuracy of the test effort, but only if the proper tools are implemented in the proper way. Test tools have to be managed as another aspect of a well-run test organization. The sophistication and applicability of test tools vary widely and the tool market is constantly changing. Tools are usually available from commercial tool vendors as well as from various freeware or shareware tool sites.

7.2 Test Tools and Automation

Much of a Test Analyst's job requires the effective use of tools. Knowing which tools to use, and when, can increase the Test Analyst's efficiency and can help to provide better testing coverage in the time allowed.

7.2.1 Test Design Tools

Test design tools are used to help create test cases and test data to be applied for testing. These tools may work from specific requirements document formats, models (e.g., UML), or inputs provided by the Test Analyst. Test design tools are often designed and built to work with particular formats and particular products such as specific requirements management tools.

Test design tools can provide information for the Test Analyst to use when determining the types of tests that are needed to obtain the targeted level of test coverage, confidence in the system, or product risk mitigation actions. For example, classification tree tools generate (and display) the set of combinations that is needed to reach full coverage based on a selected coverage criterion. This information then can be used by the Test Analyst to determine the test cases that must be executed.

7.2.2 Test Data Preparation Tools

Test data preparation tools provide several benefits. Some test data preparation tools are able to analyze a document such as a requirements document or even the source code to determine the data required during testing to achieve a level of coverage. Other test data preparation tools can take a data set from a production system and "scrub" or "anonymize" it to remove any personal information while still maintaining the internal integrity of that data. The scrubbed data can then be used for testing without the risk of a security leak or misuse of personal information. This is particularly important where large volumes of realistic data are required. Other data generation tools can be used to generate test data from given sets of input parameters (i.e., for use in random testing). Some of these will analyze the database structure to determine what inputs will be required from the Test Analyst.

7.2.3 Automated Test Execution Tools

Test execution tools are mostly used by Test Analysts at all levels of testing to run tests and check the outcome of the tests. The objective of using a test execution tool is typically one or more of the following:

- To reduce costs (in terms of effort and/or time)
- To run more tests
- To run the same test in many environments
- To make test execution more repeatable
- To run tests that would be impossible to run manually (i.e., large data validation tests)

These objectives often overlap into the main objectives of increasing coverage while reducing costs.

7.2.3.1 Applicability

The return on investment for test execution tools is usually highest when automating regression tests because of the low-level of maintenance expected and the repeated execution of the tests. Automating smoke tests can also be an effective use of automation due to the frequent use of the tests, the need for a quick result and, although the maintenance cost may be higher, the ability to have an automated way to evaluate a new build in a continuous integration environment.

Test execution tools are commonly used during the system and integration testing levels. Some tools, particularly API testing tools, may be used at the component testing level as well. Leveraging the tools where they are most applicable will help to improve the return in investment.

7.2.3.2 Test Automation Tool Basics

Test execution tools work by executing a set of instructions written in a programming language, often called a scripting language. The instructions to the tool are at a very detailed level that specifies inputs, order of the input, specific values used for the inputs and the expected outputs. This can make the detailed scripts susceptible to changes in the software under test (SUT), particularly when the tool is interacting with the graphical user interface (GUI).

Most test execution tools include a comparator which provides the ability to compare an actual result to a stored expected result.

7.2.3.3 Test Automation Implementation

The tendency in test execution automation (as in programming) is to move from detailed low-level instructions to more high-level languages, utilizing libraries, macros and sub-programs. Design techniques such as keyword-driven and action word-driven capture a series of instructions and reference those with a particular "keyword" or "action word". This allows the Test Analyst to write test cases in human language while ignoring the underlying programming language and lower level functions. Using this modular writing technique allows easier maintainability during changes to the functionality and interface of the software under test. [Bath08] The use of keywords in automated scripts is discussed more below.

Models can be used to guide the creation of the keywords or action words. By looking at the business process models which are often included in the requirements documents, the Test Analyst can determine the key business processes that must be tested. The steps for these processes can then be determined, including the decision points that may occur during the processes. The decision points can become action words that the test automation can obtain and use from the keyword or action word spreadsheets. Business process modeling is a method of documenting the business processes and can be used to identify these key processes and decision points. The modeling can be done manually or by using tools that will act off inputs based on business rules and process descriptions.

7.2.3.4 Improving the Success of the Automation Effort

When determining which tests to automate, each candidate test case or candidate test suite must be assessed to see if it merits automation. Many unsuccessful automation projects are based on automating the readily available manual test cases without checking the actual benefit from the automation. It may be optimal for a given set of test cases (a suite) to contain manual, semi-automated and fully automated tests.

The following aspects should be considered when implementing a test execution automation project:

Possible benefits:

- Automated test execution time will become more predictable
- Regression testing and defect validation using automated tests will be faster and more reliable late in the project
- The status and technical growth of the tester or test team may be enhanced by the use of automated tools
- Automation can be particularly helpful with iterative and incremental development lifecycles to provide better regression testing for each build or iteration
- Coverage of certain test types may only be possible with automated tools (e.g., large data validation efforts)
- Test execution automation can be more cost-effective than manual testing for large data input, conversion and comparison testing efforts by providing fast and consistent input and verification.

Possible risks:

- Incomplete, ineffective or incorrect manual testing may be automated “as is”
- The testware may be difficult to maintain, requiring multiple changes when the software under test is changed
- Direct tester involvement in test execution may be reduced, resulting in less defect detection
- The test team may have insufficient skills to use the automated tools effectively
- Irrelevant tests that do not contribute to the overall test coverage may be automated because they exist and are stable
- Tests may become unproductive as the software stabilizes (pesticide paradox)

During deployment of a test execution automation tool, it is not always wise to automate manual test cases as is, but to redefine the test cases for better automation use. This includes formatting the test cases, considering re-use patterns, expanding input by using variables instead of using hard-coded values and utilizing the full benefits of the test tool. Test execution tools usually have the ability to traverse multiple tests, group tests, repeat tests and change order of execution, all while providing analysis and reporting facilities.

For many test execution automation tools, programming skills are necessary to create efficient and effective tests (scripts) and test suites. It is common that large automated test suites are very difficult to update and manage if not designed with care. Appropriate training in test tools, programming and design techniques is valuable to make sure the full benefits of the tools are leveraged.

During test planning, it is important to allow time to periodically execute the automated test cases manually in order to retain the knowledge of how the test works and to verify correct operation as well as to review input data validity and coverage.

7.2.3.5 Keyword-Driven Automation

Keywords (sometimes referred to as action words) are mostly, but not exclusively, used to represent high-level business interactions with a system (e.g., “cancel order”). Each keyword is typically used to represent a number of detailed interactions between an actor and the system under test. Sequences of keywords (including relevant test data) are used to specify test cases.[Buwalda01]

In test automation a keyword is implemented as one or more executable test scripts. Tools read test cases written as a sequence of keywords that call the appropriate test scripts which implement the keyword functionality. The scripts are implemented in a highly modular manner to enable easy mapping to specific keywords. Programming skills are needed to implement these modular scripts.

The primary advantages of keyword-driven test automation are:

- Keywords that relate to a particular application or business domain can be defined by domain experts. This can make the task of test case specification more efficient.
- A person with primarily domain expertise can benefit from automatic test case execution (once the keywords have been implemented as scripts) without having to understand the underlying automation code.
- Test cases written using keywords are easier to maintain because they are less likely to need modification if details in the software under test change.
- Test case specifications are independent of their implementation. The keywords can be implemented using a variety of scripting languages and tools.

The automation scripts (the actual automation code) that use the keyword/action word information are usually written by developers or Technical Test Analysts while the Test Analysts usually create and maintain the keyword/action word data. While keyword-driven automation is usually run during the system testing phase, code development may start as early as the integration phases. In an iterative environment, the test automation development is a continuous process.

Once the input keywords and data are created, the Test Analysts usually assume responsibility to execute the keyword-driven test cases and to analyze any failures that may occur. When an anomaly is detected, the Test Analyst must investigate the cause of failure to determine if the problem is with the keywords, the input data, the automation script itself or with the application being tested. Usually the first step in troubleshooting is to execute the same test with the same data manually to see if the failure is in the application itself. If this does not show a failure, the Test Analyst should review the sequence of tests that led up to the failure to determine if the problem occurred in a previous step (perhaps by producing incorrect data), but the problem did not surface until later in the processing. If the Test Analyst is unable to determine the cause of failure, the trouble shooting information should be turned over to the Technical Test Analyst or developer for further analysis.

7.2.3.6 Causes for Failures of the Automation Effort

Test execution automation projects often fail to achieve their goals. These failures may be due to insufficient flexibility in the usage of the testing tool, insufficient programming skills in the testing team or an unrealistic expectation of the problems that can be solved with test execution automation. It is important to note that any test execution automation takes management, effort, skills and attention, just as any software development project. Time has to be devoted to creating a sustainable architecture, following proper design practices, providing configuration management and following good coding practices. The automated test scripts have to be tested because they are likely to contain defects. The scripts may need to be tuned for performance. Tool usability must be considered, not just for the developer but also for the people who will be using the tool to execute scripts. It may be necessary to design an interface between the tool and the user that will provide access to the test cases in a way that is organized logically for the tester but still provides the accessibility needed by the tool.

8. References

8.1 Standards

- [ISO25000] ISO/IEC 25000:2005, Software Engineering - Software Product Quality Requirements and Evaluation (SQuaRE)
  Chapters 1 and 4
- [ISO9126] ISO/IEC 9126-1:2001, Software Engineering - Software Product Quality,
  Chapters 1 and 4
- [RTCA DO-178B/ED-12B]: Software Considerations in Airborne Systems and Equipment Certification, RTCA/EUROCAE ED12B.1992.
  Chapter 1

8.2 ISTQB Documents

- [ISTQB_AL_OVIEW] ISTQB Advanced Level Overview, Version 1.0
- [ISTQB_ALTM_SYL] ISTQB Advanced Level Test Manager Syllabus, Version 1.0
- [ISTQB_ALTTA_SYL] ISTQB Advanced Level Technical Test Analyst Syllabus, Version 1.0
- [ISTQB_FL_SYL] ISTQB Foundation Level Syllabus, Version 2011
- [ISTQB_GLOSSARY] Standard glossary of terms used in Software Testing, Version 2.2, 2012

8.3 Books

[Bath08] Graham Bath, Judy McKay, “The Software Test Engineer’s Handbook”, Rocky Nook, 2008, ISBN 978-1-933952-24-6

[Beizer95] Boris Beizer, “Black-box Testing”, John Wiley & Sons, 1995, ISBN 0-471-12094-4

[Black02]: Rex Black, “Managing the Testing Process (2nd edition)”, John Wiley & Sons: New York, 2002, ISBN 0-471-22398-0

[Black07]: Rex Black, “Pragmatic Software Testing”, John Wiley and Sons, 2007, ISBN 978-0-470-12790-2

[Buwalda01]: Hans Buwalda, “Integrated Test Design and Automation”, Addison-Wesley Longman, 2001, ISBN 0-201-73725-6

[Cohn04]: Mike Cohn, “User Stories Applied: For Agile Software Development”, Addison-Wesley Professional, 2004, ISBN 0-321-20568-5

[Copeland03]: Lee Copeland, “A Practitioner's Guide to Software Test Design”, Artech House, 2003, ISBN 1-58053-791-X

[Craig02]: Rick David Craig, Stefan P. Jaskiel, “Systematic Software Testing”, Artech House, 2002, ISBN 1-580-53508-9

[Gerrard02]: Paul Gerrard, Neil Thompson, “Risk-based e-business Testing”, Artech House, 2002, ISBN 1-580-53314-0

[Gilb93]: Tom Gilb, Graham Dorothy, “Software Inspection”, Addison-Wesley, 1993, ISBN 0-201-63181-4

[Graham07]: Dorothy Graham, Erik van Veenendaal, Isabel Evans, Rex Black “Foundations of Software Testing”, Thomson Learning, 2007, ISBN 978-1-84480-355-2

[Grochmann94]: M. Grochmann (1994), Test case design using Classification Trees, in: conference proceedings STAR 1994

[Koomen06]: Tim Koomen, Leo van der Aalst, Bart Broekman, Michiel Vroon "TMap NEXT, for result driven testing", UTN Publishers, 2006, ISBN 90-72194-80-2

[Myers79]: Glenford J. Myers, “The Art of Software Testing”, John Wiley & Sons, 1979, ISBN 0-471-46912-2

[Splaine01]: Steven Splaine, Stefan P. Jaskiel, “The Web-Testing Handbook”, STQE Publishing, 2001, ISBN 0-970-43630-0

[vanVeenendaal12]: Erik van Veenendaal, “Practical risk-based testing – The PRISMA approach”, UTN Publishers, The Netherlands, ISBN 9789490986070

[Wiegers03]: Karl Wiegers, “Software Requirements 2”, Microsoft Press, 2003, ISBN 0-735-61879-8

[Whittaker03]: James Whittaker, “How to Break Software”, Addison-Wesley, 2003, ISBN 0-201-79619-8

[Whittaker09]: James Whittaker, “Exploratory Software Testing”, Addison-Wesley, 2009, ISBN 0-321-63641-4

8.4 Other References

The following references point to information available on the Internet and elsewhere. Even though these references were checked at the time of publication of this Advanced Level syllabus, the ISTQB cannot be held responsible if the references are not available anymore.

- Chapter 3
  - Czerwonka, Jacek: www.pairwise.org
  - Bug Taxonomy: www.testingeducation.org/a/bsct2.pdf
  - Sample Bug Taxonomy based on Boris Beizer’s work: inet.uni2.dk/~vinter/bugtaxst.doc
  - Good overview of various taxonomies: testingeducation.org/a/bugtax.pdf
  - Heuristic Risk-Based Testing By James Bach
  - From “Exploratory & Risk-Based Testing (2004) www.testingeducation.org"
  - Exploring Exploratory Testing , Cem Kaner and Andy Tikam , 2003
  - Pettichord, Bret, “An Exploratory Testing Workshop Report”, www.testingcraft.com/exploratorypettichord
- Chapter 4
  - www.testingstandards.co.uk

9. Index