Certified Tester

Advanced Level Syllabus
Technical Test Analyst

Version 2012

International Software Testing Qualifications Board

Advanced Level Technical Test Analyst Sub Working Group: Graham Bath (Chair), Paul Jorgensen, Jamie Mitchell; 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 each chapters for the separate modules
D101227	27DEC10	Acceptance of changes to format and corrections that have no impact on the meaning of the sentences.
Draft V1	17SEP11	First version of the new separate TTA syllabus based on agreed scoping document. AL WG Review
Draft V2	20NOV11	NB Review version
Alpha 2012	09MAR12	Incorporation of all comments from NBs received from October 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
RC 2012	02SEP12	Comments from BNLTB and Stuart Reid incorporated Paul Jorgensen cross-check
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 - Technical Test Analyst: Graham Bath (Chair), Paul Jorgensen, Jamie Mitchell.

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:

Dani Almog, Graham Bath, Franz Dijkman, Erwin Engelsma, Mats Grindal, Dr. Suhaimi Ibrahim, Skule Johansen, Paul Jorgensen, Kari Kakkonen, Eli Margolin, Rik Marselis, Judy McKay, Jamie Mitchell, Reto Mueller, Thomas Müller, Ingvar Nordstrom, Raluca Popescu, Meile Posthuma, Michael Stahl, Chris van Bael, Erik van Veenendaal, Rahul Verma, Paul Weymouth, Hans Weiberg, Wenqiang Zheng, Shaomin Zhu.

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 Technical 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 between 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 Technical 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.

0.4 Expectations

Some of the learning objectives for the Technical Test Analyst assume that basic experience is available in the following areas:

- General programming concepts
- General concepts of system architectures

1. The Technical Test Analyst's Tasks in Risk-Based Testing - 30 mins.

Keywords

product risk, risk analysis, risk assessment, risk identification, risk level, risk mitigation, risk-based testing

Learning Objectives for the Technical Test Analyst's Tasks in Risk-Based Testing

1.3 Risk Assessment

TTA-1.3.1 (K2) Summarize the generic risk factors that the Technical Test Analyst typically needs to consider

Common Learning Objectives

The following learning objective relates to content covered in more than one section of this chapter.

TTA-1.x.1 (K2) Summarize the activities of the Technical Test Analyst within a risk-based approach for planning and executing testing

1.1 Introduction

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

Because of their particular technical expertise, Technical Test Analysts are actively involved in the following risk-based testing tasks:

- Risk identification
- Risk assessment
- Risk mitigation

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

Technical Test Analysts work within the risk-based testing framework established by the Test Manager for the project. They contribute their knowledge of the technical risks that are inherent in the project, such as risks related to security, system reliability and performance.

1.2 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 Technical Test Analysts possess unique technical skills, they are particularly well-suited for conducting expert interviews, brainstorming with co-workers and also analyzing the current and past experiences to determine where the likely areas of product risk lie. In particular, the Technical Test Analysts work closely with their technical peers (e.g., developers, architects, operations engineers) to determine the areas of technical risk.

Sample risks that might be identified include:

- Performance risks (e.g., inability to achieve response times under high load conditions)
- Security risks (e.g., disclosure of sensitive data through security attacks)
- Reliability risks (e.g., application unable to meet availability specified in the Service Level Agreement)

Risk areas relating to specific software quality characteristics are covered in the relevant chapters of this syllabus.

1.3 Risk Assessment

While risk identification is about identifying as many pertinent risks as possible, risk assessment is the study of those identified risks in order to categorize each risk and determine 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.

The Technical Test Analyst contributes to finding and understanding the potential technical risk for each risk item whereas the Test Analyst contributes to understanding the potential business impact of the problem should it occur.

Generic factors that typically need to be considered include:

- Complexity of technology
- Complexity of code structure
- Conflict between stakeholders regarding technical requirements
- Communication problems resulting from the geographical distribution of the development organization
- Tools and technology
- Time, resource and management pressure
- Lack of earlier quality assurance
- High change rates of technical requirements
- Large number of defects found relating to technical quality characteristics
- Technical interface and integration issues

Given the available risk information, the Technical Test Analyst establishes the levels of risk according to the guidelines established by the Test Manager. For example, the Test Manager may determine that risks should be categorized with a value from 1 to 10, with 1 being highest risk.

1.4 Risk Mitigation

During the project, Technical Test Analysts influence how testing responds to the identified risks. This generally involves the following:

- Reducing risk by executing the most important tests and by putting into action appropriate mitigation and contingency activities as stated in the test strategy and test plan
- Evaluating risks based on additional information gathered as the project unfolds, and using that information to implement mitigation actions aimed at decreasing the likelihood or impact of the risks previously identified and analyzed

2. Structure-Based Testing - 225 mins.

Keywords
atomic condition, condition testing, control flow testing, decision condition testing, multiple condition testing, path testing, short-circuiting, statement testing, structure-based technique

Learning Objectives for Structure-Based Testing

2.2 Condition Testing
TTA-2.2.1 (K2) Understand how to achieve condition coverage and why it may be less rigorous testing than decision coverage

2.3 Decision Condition Testing
TTA-2.3.1 (K3) Write test cases by applying the Decision Condition testing test design technique to achieve a defined level of coverage

2.4 Modified Condition/Decision Coverage (MC/DC) Testing
TTA-2.4.1 (K3) Write test cases by applying the Modified Condition/Decision Coverage (MC/DC) testing test design technique to achieve a defined level of coverage

2.5 Multiple Condition Testing
TTA-2.5.1 (K3) Write test cases by applying the Multiple Condition testing test design technique to achieve a defined level of coverage

2.6 Path Testing
TTA-2.6.1 (K3) Write test cases by applying the Path testing test design technique

2.7 API Testing
TTA-2.7.1 (K2) Understand the applicability of API testing and the kinds of defects it finds

2.8 Selecting a Structure-Based Technique
TTA-2.8.1 (K4) Select an appropriate structure-based technique according to a given project situation

2.1 Introduction

This chapter principally describes structure-based test design techniques, which are also known as white box or code-based test techniques. These techniques use the code, the data and the architecture and/or system flow as the basis for test design. Each specific technique enables test cases to be derived systematically and focuses on a particular aspect of the structure to be considered. The techniques provide coverage criteria which have to be measured and associated with an objective defined by each project or organization. Achieving full coverage does not mean that the entire set of tests is complete, but rather that the technique being used no longer suggests any useful tests for the structure under consideration.

With the exception of condition coverage, the structure-based test design techniques considered in this syllabus are more rigorous than the statement and decision coverage techniques covered in the Foundation Syllabus [ISTQB_FL_SYL].

The following techniques are considered in this syllabus:

- Condition testing
- Decision Condition testing
- Modified Condition/Decision Coverage (MC/DC) testing
- Multiple Condition testing
- Path testing
- API testing

The first four of the techniques listed above are based on decision predicates and broadly find the same type of defects. No matter how complex a decision predicate may be, it will evaluate to either TRUE or FALSE, with one path being taken through the code and the other not. A defect is detected when the intended path is not taken because a complex decision predicate does not evaluate as expected.

In general the first four techniques are successively more thorough; they require more tests to be defined in order to achieve their intended coverage and to find more subtle instances of this type of defect.

Refer to [Bath08], [Beizer90], [Beizer95], [Copeland03] and [Koomen06].

2.2 Condition Testing

Compared to decision (branch) testing, which considers the entire decision as a whole and evaluates the TRUE and FALSE outcomes in separate test cases, condition testing considers how a decision is made. Each decision predicate is made up of one or more simple “atomic” conditions, each of which evaluates to a discrete Boolean value. These are logically combined to determine the final outcome of the decision. Each atomic condition must be evaluated both ways by the test cases to achieve this level of coverage.

Applicability
Condition testing is probably interesting only in the abstract because of the difficulties noted below. Understanding it, however, is necessary to achieving greater levels of coverage that build upon it.

Limitations/Difficulties
When there are two or more atomic conditions in a decision, an unwise choice in selecting test data during test design can result in achieving condition coverage while failing to achieve decision coverage. For example, assume the decision predicate, "A and B".

	A	B	A and B
Test 1	FALSE	TRUE	FALSE
Test 2	TRUE	FALSE	FALSE

To achieve 100% condition coverage, run the two tests shown in the table above. While these two tests achieve 100% condition coverage, they fail to achieve decision coverage, since in both cases the predicate evaluates to FALSE.

When a decision consists of a single atomic condition, condition testing is identical to decision testing.

2.3 Decision Condition Testing

Decision Condition testing specifies that testing must achieve condition coverage (see above), and also requires that decision coverage (see Foundation syllabus [ISTQB_FL_SYL]) also be satisfied. A thoughtful choice of test data values for atomic conditions may result in achieving this level of coverage without adding extra test cases beyond that needed to achieve condition coverage.

The example below tests the same decision predicate seen above, “A and B”. Decision Condition coverage can be achieved with the same number of tests by selecting different test values.

	A	B	A and B
Test 1	TRUE	TRUE	TRUE
Test 2	FALSE	FALSE	FALSE

This technique therefore may have an efficiency advantage.

Applicability
This level of coverage should be considered when the code being tested is important but not critical.

Limitations/Difficulties
Because it may require more test cases than testing to the decision level, it may be problematic when time is an issue.

2.4 Modified Condition/Decision Coverage (MC/DC) Testing

This technique provides a stronger level of control flow coverage. Assuming N unique atomic conditions, MC/DC can usually be achieved in N+1 unique test cases. MC/DC achieves Decision Condition coverage, but then requires the following also be fulfilled:

1. At least one test where the decision outcome would change if the atomic condition X were TRUE
2. At least one test where the decision outcome would change if the atomic condition X were FALSE
3. Each different atomic condition has tests that meet requirements 1 and 2.

	A	B	C	(A or B) and C
Test 1	TRUE	FALSE	TRUE	TRUE
Test 2	FALSE	TRUE	TRUE	TRUE
Test 3	FALSE	FALSE	TRUE	FALSE
Test 4	TRUE	FALSE	FALSE	FALSE

In the example above, decision coverage is achieved (the outcome of the decision predicate is both TRUE and FALSE), and condition coverage is achieved (A, B, and C are all evaluated both TRUE and FALSE).

In Test 1, A is TRUE and the output is TRUE. If A is changed to FALSE (as in Test 3, holding other values unchanged) the result turns to FALSE.
In Test 2, B is TRUE and the output is TRUE. If B is changed to FALSE (as in Test 3, holding other values unchanged) the result turns to FALSE.

In Test 1, C is TRUE and the output is TRUE. If C is changed to FALSE (as in Test 4, holding other values unchanged) the result turns to FALSE.

Applicability
This technique is used extensively in the aerospace software industry and many other safety-critical systems. It should be used when dealing with safety critical software where any failure may cause a catastrophe.

Limitations/Difficulties
Achieving MC/DC coverage may be complicated when there are multiple occurrences of a specific term in an expression; when this occurs, the term is said to be “coupled”. Depending on the decision statement in the code, it may not be possible to vary the value of the coupled term such that it alone causes the decision outcome to change. One approach in addressing this issue is to specify that only uncoupled atomic conditions must be tested to the MC/DC level. The other approach is to analyze each decision in which coupling occurs on a case-by-case basis.

Some programming languages and/or interpreters are designed such that they exhibit short-circuiting behavior when evaluating a complex decision statement in the code. That is, the executing code may not evaluate an entire expression if the final outcome of the evaluation can be determined after evaluating only a portion of the expression. For example, if evaluating the decision “A and B”, there is no reason to evaluate B if A evaluates to FALSE. No value of B can change the final value so the code may save execution time by not evaluating B. Short-circuiting may affect the ability to attain MC/DC coverage since some required tests may not be achievable.

2.5 Multiple Condition Testing

In rare instances, it might be required to test all possible combinations of values that a decision may contain. This exhaustive level of testing is called multiple condition coverage. The number of required tests is dependent on the number of atomic conditions in the decision statement and can be determined by calculating 2^n where n is the number of uncoupled atomic conditions. Using the same example as before, the following tests are required to achieve multiple condition coverage:

	A	B	C	(A or B) and C
Test 1	TRUE	TRUE	TRUE	TRUE
Test 2	TRUE	TRUE	FALSE	FALSE
Test 3	TRUE	FALSE	TRUE	TRUE
Test 4	TRUE	FALSE	FALSE	FALSE
Test 5	FALSE	TRUE	TRUE	TRUE
Test 6	FALSE	TRUE	FALSE	FALSE
Test 7	FALSE	FALSE	TRUE	FALSE
Test 8	FALSE	FALSE	FALSE	FALSE

If the language uses short-circuiting, the number of actual test cases will often be reduced, depending on the order and grouping of logical operations that are performed on the atomic conditions.

Applicability
Traditionally, this technique was used to test embedded software which was expected to run reliably without crashing for long periods of time (e.g., telephone switches that were expected to last 30 years). This type of testing is likely to be replaced by MC/DC testing in most critical applications.

Limitations/Difficulties
Because the number of test cases can be derived directly from a truth table containing all of the atomic conditions, this level of coverage can easily be determined. However, the sheer number of test cases required makes MC/DC coverage more applicable to most situations.

2.6 Path Testing

Path testing consists of identifying paths through the code and then creating tests to cover them. Conceptually, it would be useful to test every unique path through the system. In any non-trivial system, however, the number of test cases could become excessively large due to the nature of looping structures.

By setting aside the issue of indefinite looping, however, it is realistic to perform some path testing. To apply this technique [Beizer90] recommends that tests are created which follow many paths through the module, from entry to exit. To simplify what might be a complex task, he recommends that this can be done systematically, using the following procedure:

1. Pick the first path as the simplest, functionally sensible path from entry to exit.
2. Pick each additional path as a small variation on the previous path. Try to change only one branch in the path that is different for each successive test. Favor short paths over long paths when possible. Favor paths that make functional sense over ones that do not.
3. Pick paths that don't make functional sense only when required for coverage. Beizer notes in this rule that such paths might be extraneous and should be questioned.
4. Use intuition when choosing paths (i.e., which paths are most likely to be executed).

Note that some path segments are likely to be executed more than once using this strategy. The key point of this strategy is to test every possible branch through the code at least once and possibly many times.

Applicability
Partial path testing—as defined above—is often performed in safety critical software. It is a good addition to the other methods covered in this chapter because it looks at paths through the software rather than just at the way decisions are made,

Limitations/Difficulties
While it is possible to use a control flow graph to determine the paths, realistically a tool is required to calculate them for complex modules.

Coverage
Creating sufficient tests to cover all paths (disregarding loops) guarantees that both statement and branch coverage is achieved. Path testing provides more thorough testing than branch coverage, with a relatively small increase in the number of tests. [NIST 96]

2.7 API Testing

An Application Programming Interface (API) is code which enables communication between different processes, programs and/or systems. APIs are often utilized in a client/server relationship where one process supplies some kind of functionality to other processes.

In certain respects API testing is quite similar to testing a graphical user interface (GUI). The focus is on the evaluation of input values and returned data.

Negative testing is often crucial when dealing with APIs. Programmers who use APIs to access services external to their own code may try to use API interfaces in ways for which they were not intended. That means that robust error handling is essential to avoid incorrect operation. Combinatorial testing of many different interfaces may be required because APIs are often used in conjunction with other APIs, and because a single interface may contain several parameters, where values of these may be combined in many ways.

APIs frequently are loosely coupled, resulting in the very real possibility of lost transactions or timing glitches. This necessitates thorough testing of the recovery and retry mechanisms. An organization that provides an API interface must ensure that all services have a very high availability; this often requires strict reliability testing by the API publisher as well as infrastructure support.

Applicability
API testing is becoming more important as more systems become distributed or use remote processing as a way of off-loading some work to other processors. Examples include operating systems calls, service-oriented architectures (SOA), remote procedure calls (RPC), web services, and virtually every other distributed application. API testing is particularly applicable for testing systems of systems.

Limitations/Difficulties
Testing an API directly usually requires a Technical Test Analyst to use specialized tools. Because there is typically no direct graphical interface associated with an API, tools may be required to setup the initial environment, marshal the data, invoke the API, and determine the result.

Coverage
API testing is a description of a type of testing; it does not denote any specific level of coverage. At a minimum the API test should include exercise of all calls to the API as well as all valid and reasonable invalid values.

Types of Defects
The types of defects that can be found by testing APIs are quite disparate. Interface issues are common, as are data handling issues, timing problems, loss of transactions and duplication of transactions.

2.8 Selecting a Structure-Based Technique

The context of the system under test will determine the level of structure-based testing coverage that should be achieved. The more critical the system, the higher the level of coverage needed. In general, the higher the level of coverage required, the more time and resources will be needed to achieve that level.

Sometimes the level of coverage required may be derived from applicable standards that apply to the software system. For example, if the software were to be used in an airborne environment, it may be required to conform to standard DO-178B (in Europe, ED-12B.) This standard contains the following five failure conditions:

A. Catastrophic: failure may cause lack of critical function needed to safely fly or land the plane
B. Hazardous: failure may have a large negative impact on safety or performance
C. Major: failure is significant, but less serious than A or B
D. Minor: failure is noticeable, but with less impact than C
E. No effect: failure has no impact on safety

If the software system is categorized as level A, it must be tested to MC/DC coverage. If it is level B, it must be tested to decision level coverage although MC/DC is optional. Level C requires statement coverage at minimum.

Likewise, IEC-61508 is an international standard for the functional safety of programmable, electronic, safety-related systems. This standard has been adapted in many different areas, including automotive, rail, manufacturing process, nuclear power plants, and machinery. Criticality is defined using a graduated Safety Integrity Level (SIL) continuum (1 being the least critical, 4 being the most) and coverage is recommended as follows:

1. Statement and branch coverage recommended
2. Statement coverage highly recommended, branch coverage recommended
3. Statement and branch coverage highly recommended
4. MC/DC highly recommended

In modern systems, it is rare that all processing will be done on a single system. API testing should be instituted anytime some of the processing is going to be done remotely. The criticality of the system should determine how much effort should be invested in API testing.

As always, the context of the software system under test should guide the Technical Test Analyst on the methods used in testing.

3. Analytical Techniques - 255 mins.

Keywords

control flow analysis, cyclomatic complexity, data flow analysis, definition-use pairs, dynamic analysis, memory leak, pairwise integration testing, neighborhood integration testing, static analysis, wild pointer

Learning Objectives for Analytical Techniques

3.2 Static Analysis

TTA-3.2.1 (K3) Use control flow analysis to detect if code has any control flow anomalies
TTA-3.2.2 (K3) Use data flow analysis to detect if code has any data flow anomalies
TTA-3.2.3 (K3) Propose ways to improve the maintainability of code by applying static analysis
TTA-3.2.4 (K2) Explain the use of call graphs for establishing integration testing strategies

3.3 Dynamic Analysis

TTA-3.3.1 (K3) Specify goals to be achieved by the use of dynamic analysis

3.1 Introduction

There are two types of analysis: static analysis and dynamic analysis.

Static analysis (Section 3.2) encompasses the analytical testing that can occur without executing the software. Because the software is not executing, it is examined either by a tool or by a person to determine if it will process correctly when it is executed. This static view of the software allows detailed analysis without having to create the data and preconditions that would cause the scenario to be exercised.

Note that the different forms of review which are relevant for the Technical Test Analyst are covered in Chapter 5.

Dynamic analysis (Section 3.3) requires the actual execution of the code and is used to find coding faults which are more easily detected when the code is executing (e.g., memory leaks). Dynamic analysis, as with static analysis, may rely on tools or may rely on an individual monitoring the executing system watching for such indicators as rapid memory growth.

3.2 Static Analysis

The objective of static analysis is to detect actual or potential faults in code and system architecture and to improve their maintainability. Static analysis is generally supported by tools.

3.2.1 Control Flow Analysis

Control flow analysis is the static technique where the control flow through a program is analyzed, either through the use of a control flow graph or a tool. There are a number of anomalies which can be found in a system using this technique, including loops that are badly designed (e.g., having multiple entry points), ambiguous targets of function calls in certain languages (e.g., Scheme), incorrect sequencing of operations, etc.

One of the most common uses of control flow analysis is to determine cyclomatic complexity. The cyclomatic complexity value is a positive integer which represents the number of independent paths in a strongly connected graph with loops and iterations ignored as soon as they have been traversed once. Each independent path, from entry to exit, represents a unique path through the module. Each unique path should be tested.

The cyclomatic complexity value is generally used to understand the overall complexity of a module. Thomas McCabe's theory [McCabe 76] was that the more complex the system, the harder it would be to maintain and the more defects it would contain. Many studies over the years have noted this correlation between complexity and the number of contained defects. The NIST (National Institute of Standards and Technology) recommends a maximum complexity value of 10. Any module that is measured with a higher complexity may need to be divided into multiple modules.

3.2.2 Data Flow Analysis

Data flow analysis covers a variety of techniques which gather information about the use of variables in a system. Scrutiny is given to the lifecycle of the variables, (i.e., where they are declared, defined, read, evaluated and destroyed), since anomalies can occur during any of those operations.

One common technique is called define-use notation where the lifecycle of each variable is split into three different atomic actions:

- d: when the variable is declared, defined or initialized
- u: when the variable is used or read in either a computation or a decision predicate
- k: when the variable is killed, destroyed or goes out of scope

These three atomic actions are combined into pairs (“definition-use pairs”) to illustrate the data flow. For example, a "du-path" represents a fragment of the code where a data variable is defined and then subsequently used.

Possible data anomalies include performing the correct action on a variable at the wrong time or carrying out an incorrect action on data in a variable. These anomalies include:

- Assigning an invalid value to a variable
- Failing to assign a value to a variable before using it
- Taking an incorrect path due to an incorrect value in a control predicate
- Trying to use a variable after it is destroyed
- Referencing a variable when it is out of scope
- Declaring and destroying a variable without using it
- Redefining a variable before it has been used
- Failing to kill a dynamically allocated variable (causing a possible memory leak)
- Modifying a variable, which results in unexpected side effects (e.g., ripple effects when changing a global variable without considering all uses of the variable)

The development language being used may guide the rules used in data flow analysis. Programming languages may allow the programmer to perform certain actions with variables which are not illegal, but may cause the system to behave differently than the programmer expected under certain circumstances. For example, a variable might be defined twice without actually being used when a certain path is followed. Data flow analysis will often label these uses "suspicious". While this may be a legal use of the variable assignment capability, it can lead to future maintainability issues in the code.

Data flow testing "uses the control flow graph to explore the unreasonable things that can happen to data" [Beizer90] and therefore finds different defects than control flow testing. A Technical Test Analyst should include this technique when planning testing since many of these defects cause intermittent failures that are difficult to find while performing dynamic testing.

However, data flow analysis is a static technique; it may miss some issues that occur to data in the run-time system. For example, the static data variable may contain a pointer into a dynamically created array that does not even exist until run-time. Multi-processor usage and pre-emptive multi-tasking may create race conditions which will not be found by data flow or control flow analysis.

3.2.3 Using Static Analysis for Improving Maintainability

Static analysis can be applied in a number of ways to improve the maintainability of code, architecture and web sites.

Poorly written, uncommented and unstructured code tends to be harder to maintain. It may require more effort for developers to locate and analyze defects in the code and the modification of the code to correct a defect or add a new feature may result in further defects being introduced.

Static analysis is used with tool support to improve code maintainability by verifying compliance to coding standards and guidelines. These standards and guidelines describe required coding practices such as naming conventions, commenting, indentation and code modularization. Note that static analysis tools generally flag warnings rather than errors even though the code may be syntactically correct.

Modular designs generally result in more maintainable code. Static analysis tools support the development of modular code in the following ways:

- They search for repeated code. These sections of code may be candidates for refactoring into modules (although the run-time overhead imposed by module calls may be an issue for real-time systems).
- They generate metrics which are valuable indicators of code modularization. These include measures of coupling and cohesion. A system that is to have good maintainability is more likely to have a low measure of coupling (the degree to which modules rely on each other during execution) and a high measure of cohesion (the degree to which a module is self-contained and focused on a single task).
- They indicate, in object-oriented code, where derived objects may have too much or too little visibility into parent classes.
- They highlight areas in code or architecture with a high level of structural complexity, which is generally considered to be an indicator for poor maintainability and a higher potential for containing faults. Acceptable levels of cyclomatic complexity (see Section 3.2.1.) may be specified in guidelines to ensure that code is developed in a modular fashion with maintainability and defect prevention in mind. Code with high levels of cyclomatic complexity may be candidates for modularization.

Maintenance of a web site can also be supported using static analysis tools. Here the objective is to check if the tree-like structure of the site is well-balanced or if there is an imbalance that will lead to:

- More difficult testing tasks
- Increased maintenance workload
- Difficult navigation for the user

3.2.4 Call Graphs

Call graphs are a static representation of communication complexity. They are directed graphs in which nodes represent program units and edges represent communication among units.

Call graphs may be used in unit testing where different functions or methods call each other, in integration and system testing when separate modules call each other, or in system integration testing when separate systems call each other.

Call graphs can be used for the following purposes:

- Designing tests that call a specific module or system
- Establishing the number of locations within the software from where a module or system is called
- Evaluating the structure of the code and of the system architecture
- Providing suggestions for the order of integration (pairwise and neighborhood integration. These are discussed in more detail below.

In the Foundation Level syllabus [ISTQB_FL_SYL], two different categories of integration testing were discussed: incremental (top-down, bottom-up, etc.) and non-incremental (big bang). The incremental methods were said to be preferred because they introduce code in increments thus making fault isolation easier since the amount of code involved is limited.

In this Advanced syllabus, three more non-incremental methods using call graphs are introduced. These may be preferable to incremental methods which likely will require additional builds to complete testing and non-shippable code to be written to support the testing. These three methods are:

- Pairwise integration testing (not to be confused with the black box test technique “pairwise testing”), targets pairs of components that work together as seen in the call graph for

integration testing. While this method reduces the number of builds only by a small amount, it reduces the amount of test harness code needed.
- Neighborhood integration tests all of the nodes that connect to a given node as the basis for the integration testing. All predecessor and successor nodes of a specific node in the call graph are the basis for the test.
- McCabe's design predicate approach uses the theory of cyclomatic complexity as applied to a call graph for modules. This requires the construction of a call graph that shows the different ways that modules can call each other, including:
  - Unconditional call: the call of one module to another always happens
  - Conditional call: the call of one module to another sometimes happens
  - Mutually exclusive conditional call: a module will call one (and only one) of a number of different modules
  - Iterative call: one module calls another at least once but may call it multiple times
  - Iterative conditional call: one module can call another zero to many times

  After creating the call graph, the integration complexity is calculated, and tests are created to cover the graph.

Refer to [Jorgensen07] for more information on using call graphs and pairwise integration testing.

3.3 Dynamic Analysis

3.3.1 Overview

Dynamic analysis is used to detect failures where the symptoms may not be immediately visible. For example, the possibility of memory leaks may be detectable by static analysis (finding code that allocates but never frees memory), but a memory leak is readily apparent with dynamic analysis.

Failures that are not immediately reproducible can have significant consequences on the testing effort and on the ability to release or productively use software. Such failures may be caused by memory leaks, incorrect use of pointers and other corruptions (e.g., of the system stack) [Kaner02]. Due to the nature of these failures, which may include the gradual worsening of system performance or even system crashes, testing strategies must consider the risks associated with such defects and, where appropriate, perform dynamic analysis to reduce them (typically by using tools). Since these failures often are the most expensive failures to find and to correct, it is recommended to perform dynamic analysis early in the project.

Dynamic analysis may be applied to accomplish the following:
- Prevent failures from occurring by detecting wild pointers and loss of system memory
- Analyze system failures which cannot easily be reproduced
- Evaluate network behavior
- Improve system performance by providing information on run-time system behavior

Dynamic analysis may be performed at any test level and requires technical and system skills to do the following:
- Specify the testing objectives of dynamic analysis
- Determine the proper time to start and stop the analysis
- Analyze the results

During system testing, dynamic analysis tools can be used even if the Technical Test Analysts have minimal technical skills; the tools utilized usually create comprehensive logs which can be analyzed by those with the needed technical skills.

3.3.2 Detecting Memory Leaks

A memory leak occurs when the areas of memory (RAM) available to a program are allocated by that program but are not subsequently released when no longer needed. This memory area is left as allocated and is not available for re-use. When this occurs frequently or in low memory situations, the program may run out of usable memory. Historically, memory manipulation was the responsibility of the programmer. Any dynamically allocated areas of memory had to be released by the allocating program within the correct scope to avoid a memory leak. Many modern programming environments include automatic or semi-automatic "garbage collection" where allocated memory is freed without the programmer's direct intervention. Isolating memory leaks can be very difficult in cases where existing allocated memory is freed by the automatic garbage collection.

Memory leaks cause problems which develop over time and may not always be immediately obvious. This may be the case if, for example, the software has been recently installed or the system restarted, which often occurs during testing. For these reasons, the negative effects of memory leaks may often first be noticed when the program is in production.

The symptoms of a memory leak are a steadily worsening of system response time which may ultimately result in system failure. While such failures may be resolved by re-starting (re-booting) the system, this may not always be practical or even possible.

Many dynamic analysis tools identify areas in the code where memory leaks occur so that they can be corrected. Simple memory monitors can also be used to obtain a general impression of whether available memory is declining over time, although a follow-up analysis would still be required to determine the exact cause of the decline.

There are other sources for leaks that also should be considered. Examples include file handles, semaphores and connection pools for resources.

3.3.3 Detecting Wild Pointers

"Wild" pointers within a program are pointers which must not be used. For example, a wild pointer may have "lost" the object or function to which it should be pointing or it does not point to the area of memory intended (e.g., it points to an area that is beyond the allocated boundaries of an array). When a program uses wild pointers, a variety of consequences may occur:

- The program may perform as expected. This may be the case where the wild pointer accesses memory which is currently not used by the program and is notionally "free" and/or contains a reasonable value.
- The program may crash. In this case the wild pointer may have caused a part of the memory to be incorrectly used which is critical to the running of the program (e.g., the operating system).
- The program does not function correctly because objects required by the program cannot be accessed. Under these conditions the program may continue to function, although an error message may be issued.
- Data in the memory location may be corrupted by the pointer and incorrect values subsequently used.

Note that any changes made to the program's memory usage (e.g., a new build following a software change) may trigger any of the four consequences listed above. This is particularly critical where initially the program performs as expected despite the use of wild pointers, and then crashes unexpectedly (perhaps even in production) following a software change. It is important to note that such failures are often symptoms of an underlying defect (i.e., the wild pointer) (Refer to [Kaner02], "Lesson 74"). Tools can help identify wild pointers as they are used by the program, irrespective of their impact on the program's execution. Some operating systems have built-in functions to check for memory access violations during run-time. For instance, the operating system may throw an exception

when an application tries to access a memory location that is outside of that application's allowed memory area.

3.3.4 Analysis of Performance

Dynamic analysis is not just useful for detecting failures. With the dynamic analysis of program performance, tools help identify performance bottlenecks and generate a wide range of performance metrics which can be used by the developer to tune the system performance. For example, information can be provided about the number of times a module is called during execution. Modules which are frequently called would be likely candidates for performance enhancement.

By merging the information about the dynamic behavior of the software with information obtained from call graphs during static analysis (see Section 3.2.4), the tester can also identify the modules which might be candidates for detailed and extensive testing (e.g., modules which are frequently called and have many interfaces).

Dynamic analysis of program performance is often done while conducting system tests, although it may also be done when testing a single sub-system in earlier phases of testing using test harnesses.

4. Quality Characteristics for Technical Testing - 405 mins.

Keywords

adaptability, analyzability, changeability, co-existence, efficiency, installability, maintainability testing, maturity, operational acceptance test, operational profile, performance testing, portability testing, recoverability testing, reliability growth model, reliability testing, replaceability, resource utilization testing, robustness, security testing, stability, testability

Learning Objectives for Quality Characteristics for Technical Testing

4.2 General Planning Issues

TTA-4.2.1 (K4) For a particular project and system under test, analyze the non-functional requirements and write the respective sections of the test plan

4.3 Security Testing

TTA-4.3.1 (K3) Define the approach and design high-level test cases for security testing

4.4 Reliability Testing

TTA-4.4.1 (K3) Define the approach and design high-level test cases for the reliability quality characteristic and its corresponding ISO 9126 sub-characteristics

4.5 Performance Testing

TTA-4.5.1 (K3) Define the approach and design high-level operational profiles for performance testing

Common Learning Objectives

The following learning objectives relate to content covered in more than one section of this chapter.

TTA-4.x.1 (K2) Understand and explain the reasons for including maintainability, portability and resource utilization tests in a testing strategy and/or test approach
TTA-4.x.2 (K3) Given a particular product risk, define the particular non-functional test type(s) which are most appropriate
TTA-4.x.3 (K2) Understand and explain the stages in an application's lifecycle where non-functional tests should be applied
TTA-4.x.4 (K3) For a given scenario, define the types of defects you would expect to find by using non-functional testing types

4.1 Introduction

In general, the Technical Test Analyst focuses testing on "how" the product works, rather than the functional aspects of "what" it does. These tests can take place at any test level. For example, during component testing of real time and embedded systems, conducting performance benchmarking and testing resource usage is important. During system test and Operational Acceptance Test (OAT), testing for reliability aspects, such as recoverability, is appropriate. The tests at this level are aimed at testing a specific system, i.e., combinations of hardware and software. The specific system under test may include various servers, clients, databases, networks and other resources. Regardless of the test level, testing should be conducted according to the risk priorities and the available resources.

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 series (which has superseded ISO 9126) may also be of use. The ISO 9126 quality characteristics are divided into characteristics, each of which may have sub-characteristics. 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

While this allocation of work may vary in different organizations, it is this 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. Because those standards can vary widely depending on the industry, they will not be discussed in depth here. If the Technical 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, preparation and test execution time built into the schedule [Bath08]. Some of this testing, e.g., performance testing, requires extensive planning, dedicated equipment, specific tools, specialized testing skills and, in most cases, a significant amount of time. Testing of the quality characteristics and sub-

characteristics 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 lifecycle, as discussed in the sections below.

While the Test Manager will be concerned with compiling and reporting the summarized metric information concerning quality characteristics and sub-characteristics, the Test Analyst or the Technical Test Analyst (according to the table above) gathers the information for each metric.

Measurements of quality characteristics gathered in pre-production tests by the Technical Test Analyst may form the basis for Service Level Agreements (SLA) between the supplier and the stakeholders (e.g., customers, operators) of the software system. In some cases, the tests may continue to be executed after the software has entered production, often by a separate team or organization. This is usually seen for efficiency and reliability testing which may show different results in the production environment than in the testing environment.

4.2 General Planning Issues

Failure to plan for non-functional tests can put the success of an application at considerable risk. The Technical Test Analyst may be requested by the Test Manager to identify the principal risks for the relevant quality characteristics (see table in Section 4.1) and address any planning issues associated with the proposed tests. These may be used in creating the Master Test Plan. The following general factors are considered when performing these tasks:

- Stakeholder requirements
- Required tool acquisition and training
- Test environment requirements
- Organizational considerations
- Data security considerations

4.2.1 Stakeholder Requirements

Non-functional requirements are often poorly specified or even non-existent. At the planning stage, Technical Test Analysts must be able to obtain expectation levels relating to technical quality characteristics from affected stakeholders and evaluate the risks that these represent.

A common approach is to assume that if the customer is satisfied with the existing version of the system, they will continue to be satisfied with new versions, as long as the achieved quality levels are maintained. This enables the existing version of the system to be used as a benchmark. This can be a particularly useful approach to adopt for some of the non-functional quality characteristics such as performance, where stakeholders may find it difficult to specify their requirements.

It is advisable to obtain multiple viewpoints when capturing non-functional requirements. They must be elicited from stakeholders such as customers, users, operations staff and maintenance staff; otherwise some requirements are likely to be missed.

4.2.2 Required Tool Acquisition and Training

Commercial tools or simulators are particularly relevant for performance and certain security tests. Technical Test Analysts should estimate the costs and timescales involved for acquiring, learning and implementing the tools. Where specialized tools are to be used, planning should account for the learning curves for new tools and/or the cost of hiring external tool specialists.

The development of a complex simulator may represent a development project in its own right and should be planned as such. In particular the testing and documentation of the developed tool must be accounted for in the schedule and resource plan. Sufficient budget and time should be planned for

upgrading and retesting the simulator as the simulated product changes. The planning for simulators to be used in safety-critical applications must take into account the acceptance testing and possible certification of the simulator by an independent body.

4.2.3 Test Environment Requirements

Many technical tests (e.g., security tests, performance tests) require a production-like test environment in order to provide realistic measures. Depending on the size and complexity of the system under test, this can have a significant impact on the planning and funding of the tests. Since the cost of such environments may be high, the following alternatives may be considered:

- Using the production environment
- Using a scaled-down version of the system. Care must then be taken that the test results obtained are sufficiently representative of the production system.

The timing of such test executions must be planned carefully and it is quite likely that such tests can only be executed at specific times (e.g., at low usage times).

4.2.4 Organizational Considerations

Technical tests may involve measuring the behavior of several components in a complete system (e.g., servers, databases, networks). If these components are distributed across a number of different sites and organizations, the effort required to plan and co-ordinate the tests may be significant. For example, certain software components may only be available for system testing at particular times of day or year, or organizations may only offer support for testing for a limited number of days. Failing to confirm that system components and staff (i.e., “borrowed” expertise) from other organizations are available “on call” for testing purposes may result in severe disruption to the scheduled tests.

4.2.5 Data Security Considerations

Specific security measures implemented for a system should be taken into account at the test planning stage to ensure that all testing activities are possible. For example, the use of data encryption may make the creation of test data and the verification of results difficult.

Data protection policies and laws may preclude the generation of any required test data based on production data. Making test data anonymous is a non-trivial task which must be planned for as part of the test implementation.

4.3 Security Testing

4.3.1 Introduction

Security testing differs from other forms of functional testing in two significant areas:

1. Standard techniques for selecting test input data may miss important security issues
2. The symptoms of security defects are very different from those found with other types of functional testing

Security testing assesses a system's vulnerability to threats by attempting to compromise the system's security policy. The following is a list of potential threats which should be explored during security testing:

- Unauthorized copying of applications or data
- Unauthorized access control (e.g., ability to perform tasks for which the user does not have rights). User rights, access and privileges are the focus of this testing. This information should be available in the specifications for the system.

- Software which exhibits unintended side-effects when performing its intended function. For example, a media player which correctly plays audio but does so by writing files out to unencrypted temporary storage exhibits a side-effect which may be exploited by software pirates.
- Code inserted into a web page which may be exercised by subsequent users (cross-site scripting or XSS). This code may be malicious.
- Buffer overflow (buffer overrun) which may be caused by entering strings into a user interface input field which are longer than the code can correctly handle. A buffer overflow vulnerability represents an opportunity for running malicious code instructions.
- Denial of service, which prevents users from interacting with an application (e.g., by overloading a web server with “nuisance” requests).
- The interception, mimicking and/or altering and subsequent relaying of communications (e.g., credit card transactions) by a third party such that a user remains unaware of that third party’s presence (“Man in the Middle” attack)
- Breaking the encryption codes used to protect sensitive data
- Logic bombs (sometimes called Easter Eggs), which may be maliciously inserted into code and which activate only under certain conditions (e.g., on a specific date). When logic bombs activate, they may perform malicious acts such as the deletion of files or formatting of disks.

4.3.2 Security Test Planning

In general the following aspects are of particular relevance when planning security tests:

- Because security issues can be introduced during the architecture, design and implementation of the system, security testing may be scheduled for the unit, integration and system testing levels. Due to the changing nature of security threats, security tests may also be scheduled on a regular basis after the system has entered production.
- The test strategies proposed by the Technical Test Analyst may include code reviews and static analysis with security tools. These can be effective in finding security issues in architecture, design documents and code that are easily missed during dynamic testing.
- The Technical Test Analyst may be called upon to design and execute certain security “attacks” (see below) which require careful planning and coordination with stakeholders. Other security tests may be performed in cooperation with developers or with Test Analysts (e.g., testing user rights, access and privileges). Planning of the security tests must include careful consideration of organizational issues such as these.
- An essential aspect of security test planning is obtaining approvals. For the Technical Test Analyst, this means obtaining explicit permission from the Test Manager to perform the planned security tests. Any additional, unplanned tests performed could appear to be actual attacks and the person conducting those tests could be at risk for legal action. With nothing in writing to show intent and authorization, the excuse "We were performing a security test" may be difficult to explain convincingly.
- It should be noted that improvements which may be made to the security of a system may affect its performance. After making security improvements it is advisable to consider the need for conducting performance tests (see Section 4.5 below).

4.3.3 Security Test Specification

Particular security tests may be grouped [Whittaker04] according to the origin of the security risk:

- User interface related - unauthorized access and malicious inputs
- File system related - access to sensitive data stored in files or repositories
- Operating system related - storage of sensitive information such as passwords in non-encrypted form in memory which could be exposed when the system is crashed through malicious inputs
- External software related - interactions which may occur among external components that the system utilizes. These may be at the network level (e.g., incorrect packets or messages

passed) or at the software component level (e.g., failure of a software component on which the software relies).

The following approach [Whittaker04] may be used to develop security tests:

- Gather information which may be useful in specifying tests, such as names of employees, physical addresses, details regarding the internal networks, IP numbers, identity of software or hardware used, and operating system version.
- Perform a vulnerability scan using widely available tools. Such tools are not used directly to compromise the system(s), but to identify vulnerabilities that are, or that may result in, a breach of security policy. Specific vulnerabilities can also be identified using checklists such as those provided by the National Institute of Standards and Technology (NIST) [Web-2].
- Develop "attack plans" (i.e., a plan of testing actions intended to compromise a particular system's security policy) using the gathered information. Several inputs via various interfaces (e.g., user interface, file system) need to be specified in the attack plans to detect the most severe security faults. The various "attacks" described in [Whittaker04] are a valuable source of techniques developed specifically for security testing.

Security issues can also be exposed by reviews (see Chapter 5) and/or the use of static analysis tools (see Section 3.2). Static analysis tools contain an extensive set of rules which are specific to security threats and against which the code is checked. For example, buffer overflow issues, caused by failure to check buffer size before data assignment, can be found by the tool.

Static analysis tools can be used for web code to check for possible exposure to security vulnerabilities such as code injection, cookie security, cross site scripting, resource tampering and SQL code injection.

4.4 Reliability Testing

The ISO 9126 classification of product quality characteristics defines the following sub-characteristics of reliability:

- Maturity
- Fault tolerance
- Recoverability

4.4.1 Measuring Software Maturity

An objective of reliability testing is to monitor a statistical measure of software maturity over time and compare this to a desired reliability goal which may be expressed as a Service Level Agreement (SLA). The measures may take the form of a Mean Time Between Failures (MTBF), Mean Time To Repair (MTTR) or any other form of failure intensity measurement (e.g., number of failures of a particular severity occurring per week). These may be used as exit criteria (e.g., for production release).

4.4.2 Tests for Fault Tolerance

In addition to the functional testing that evaluates the software's tolerance to faults in terms of handling unexpected input values (so-called negative tests), additional testing is needed to evaluate a system's tolerance to faults which occur externally to the application under test. Such faults are typically reported by the operating system (e.g., disk full, process or service not available, file not found, memory not available). Tests of fault tolerance at the system level may be supported by specific tools.

Note that the terms "robustness" and "error tolerance" are also commonly used when discussing fault tolerance (see [ISTQB_GLOSSARY] for details).

4.4.3 Recoverability Testing

Further forms of reliability testing evaluate the software system's ability to recover from hardware or software failures in a predetermined manner which subsequently allows normal operations to be resumed. Recoverability tests include Failover and Backup and Restore tests.

Failover tests are performed where the consequences of a software failure are so negative that specific hardware and/or software measures have been implemented to ensure system operation even in the event of failure. Failover tests may be applicable, for example, where the risk of financial losses is extreme or where critical safety issues exist. Where failures may result from catastrophic events, this form of recoverability testing may also be called "disaster recovery" testing.

Typical preventive measures for hardware failures might include load balancing across several processors and clustering servers, processors or disks so that one can immediately take over from another if it should fail (redundant systems). A typical software measure might be the implementation of more than one independent instance of a software system (for example, an aircraft's flight control system) in so-called redundant dissimilar systems. Redundant systems are typically a combination of software and hardware measures and may be called duplex, triplex or quadruplex systems, depending on the number of independent instances (two, three or four respectively). The dissimilar aspect for the software is achieved when the same software requirements are provided to two (or more) independent and not connected development teams, with the objective of having the same services provided with different software. This protects the redundant dissimilar systems in that a similar defective input is less likely to have the same result. These measures taken to improve the recoverability of a system may directly influence its reliability as well and should also be considered when performing reliability testing.

Failover testing is designed to explicitly test systems by simulating failure modes or actually causing failures in a controlled environment. Following a failure, the failover mechanism is tested to ensure that data is not lost or corrupted and that any agreed service levels are maintained (e.g., function availability or response times). For more information on failover testing, see [Web-1].

Backup and Restore tests focus on the procedural measures set up to minimize the effects of a failure. Such tests evaluate the procedures (usually documented in a manual) for taking different forms of backup and for restoring that data if data loss or corruption should occur. Test cases are designed to ensure that critical paths through each procedure are covered. Technical reviews may be performed to "dry-run" these scenarios and validate the manuals against the actual procedures. Operational Acceptance Tests (OAT) exercise the scenarios in a production or production-like environment to validate their actual use.

Measures for Backup and Restore tests may include the following:

- Time taken to perform different types of backup (e.g., full, incremental)
- Time taken to restore data
- Levels of guaranteed data backup (e.g., recovery of all data no more than 24 hours old, recovery of specific transaction data no more than one hour old)

4.4.4 Reliability Test Planning

In general the following aspects are of particular relevance when planning reliability tests:

- Reliability can continue to be monitored after the software has entered production. The organization and staff responsible for operation of the software must be consulted when gathering reliability requirements for test planning purposes.
- The Technical Test Analyst may select a reliability growth model which shows the expected levels of reliability over time. A reliability growth model can provide useful information to the Test Manager by enabling comparison of the expected and achieved reliability levels.

- Reliability tests should be conducted in a production-like environment. The environment used should remain as stable as possible to enable reliability trends to be monitored over time.
- Because reliability tests often require use of the entire system, reliability testing is most commonly done as part of system testing. However, individual components can be subjected to reliability testing as well as integrated sets of components. Detailed architecture, design and code reviews can also be used to remove some of the risk of reliability issues occurring in the implemented system.
- In order to produce test results that are statistically significant, reliability tests usually require long execution times. This may make it difficult to schedule within other planned tests.

4.4.5 Reliability Test Specification

Reliability testing may take the form of a repeated set of predetermined tests. These may be tests selected at random from a pool or test cases generated by a statistical model using random or pseudo-random methods. Tests may also be based on patterns of use which are sometimes referred to as “Operational Profiles” (see Section 4.5.4).

Certain reliability tests may specify that memory-intensive actions be executed repeatedly so that possible memory leaks can be detected.

4.5 Performance Testing

4.5.1 Introduction

The ISO 9126 classification of product quality characteristics includes performance (time behavior) as a sub-characteristic of efficiency. Performance testing focuses on the ability of a component or system to respond to user or system inputs within a specified time and under specified conditions.

Performance measurements vary according to the objectives of the test. For individual software components, performance may be measured according to CPU cycles, while for client-based systems performance may be measured according to the time taken to respond to a particular user request. For systems whose architectures consist of several components (e.g., clients, servers, databases) performance measurements are taken for transactions between individual components so that performance “bottlenecks” can be identified.

4.5.2 Types of Performance Testing

4.5.2.1 Load Testing

Load testing focuses on the ability of a system to handle increasing levels of anticipated realistic loads resulting from the transaction requests generated by numbers of concurrent users or processes. Average response times for users under different scenarios of typical use (operational profiles) can be measured and analyzed. See also [Splaine01].

4.5.2.2 Stress Testing

Stress testing focuses on the ability of a system or component to handle peak loads at or beyond the limits of its anticipated or specified workloads, or with reduced availability of resources such as accessible computer capacity and available bandwidth. Performance levels should degrade slowly and predictably without failure as stress levels are increased. In particular, the functional integrity of the system should be tested while the system is under stress in order to find possible faults in functional processing or data inconsistencies.

One possible objective of stress testing is to discover the limits at which a system actually fails so that the "weakest link in the chain" can be determined. Stress testing allows additional capacity to be added to the system in a timely manner (e.g., memory, CPU capability, database storage).

4.5.2.3 Scalability Testing

Scalability testing focuses on the ability of a system to meet future efficiency requirements, which may be beyond those currently required. The objective of the tests is to determine the system's ability to grow (e.g., with more users, larger amounts of data stored) without exceeding the currently specified performance requirements or failing. Once the limits of scalability are known, threshold values can be set and monitored in production to provide a warning of impending problems. In addition the production environment may be adjusted with appropriate amounts of hardware.

4.5.3 Performance Test Planning

In addition to the general planning issues described in Section 4.2, the following factors can influence the planning of performance tests:

- Depending on the test environment used and the software being tested, (see Section 4.2.3) performance tests may require the entire system to be implemented before effective testing can be done. In this case, performance testing is usually scheduled to occur during system test. Other performance tests which can be conducted effectively at the component level may be scheduled during unit testing.
- In general it is desirable to conduct initial performance tests as early as possible, even if a production-like is not yet available. These early tests may find performance problems (e.g. bottlenecks) and reduce project risk by avoiding time-consuming corrections in the later stages of software development or production.
- Code reviews, in particular those which focus on database interaction, component interaction and error handling, can identify performance issues (particularly regarding "wait and retry" logic and inefficient queries) and should be scheduled early in the software lifecycle.
- The hardware, software and network bandwidth needed to run the performance tests should be planned and budgeted. Needs depend primarily on the load to be generated, which may be based on the number of virtual users to be simulated and the amount of network traffic they are likely to generate. Failure to account for this may result in unrepresentative performance measurements being taken. For example, verifying the scalability requirements of a much-visited Internet site may require the simulation of hundreds of thousands of virtual users.
- Generating the required load for performance tests may have a significant influence on hardware and tool acquisition costs. This must be considered in the planning of performance tests to ensure that adequate funding is available.
- The costs of generating the load for performance tests may be minimized by renting the required test infrastructure. This may involve, for example, renting "top-up" licenses for performance tools or by using the services of a third-party provider for meeting hardware needs (e.g., cloud-services). If this approach is taken, the available time for conducting the performance tests may be limited and must therefore be carefully planned.
- Care should be taken at the planning stage to ensure that the performance tool to be used provides the required compatibility with the communications protocols used by the system under test.
- Performance-related defects often have significant impact on the system under test. When performance requirements are imperative, it is often useful to conduct performance tests on the critical components (via drivers and stubs) instead of waiting for system tests.

4.5.4 Performance Test Specification

The specification of tests for different performance test types such as load and stress are based on the definition of operational profiles. These represent distinct forms of user behavior when interacting with an application. There may be multiple operational profiles for a given application.

The numbers of users per operational profile may be obtained by using monitoring tools (where the actual or comparable application is already available) or by predicting usage. Such predictions may be based on algorithms or provided by the business organization, and are especially important for specifying the operational profile(s) to be used for scalability testing.

Operational profiles are the basis for the number and types of test cases to be used during performance testing. These tests are often controlled by test tools that create "virtual" or simulated users in quantities that will represent the profile under test (see Section 6.3.2).

4.6 Resource Utilization

The ISO 9126 classification of product quality characteristics includes resource utilization as a sub-characteristic of efficiency. Tests relating to resource utilization evaluate the usage of system resources (e.g., usage of memory, disk capacity, network bandwidth, connections) against a predefined benchmark. These are compared under both normal loads and stress situations, such as high levels of transaction and data volumes, to determine if unnatural growth in usage is occurring.

For example for real-time embedded systems, memory usage (sometimes referred to as a “memory footprint”) plays a significant role in performance testing. If the memory footprint exceeds the allowed measure, the system may have insufficient memory needed to perform its tasks within the specified time periods. This may slow down the system or even lead to a system crash.

Dynamic analysis may also be applied to the task of investigating resource utilization (see Section 3.3.4) and detecting performance bottlenecks.

4.7 Maintainability Testing

Software often spends substantially more of its lifetime being maintained than being developed. Maintenance testing is performed to test the changes to an operational system or the impact of a changed environment to an operational system. To ensure that the task of conducting maintenance is as efficient as possible, maintainability testing is performed to measure the ease with which code can be analyzed, changed and tested.

Typical maintainability objectives of affected stakeholders (e.g., the software owner or operator) include:

- Minimizing the cost of owning or operating the software
- Minimizing down time required for software maintenance

Maintainability tests should be included in a test strategy and/or test approach where one or more of the following factors apply:

- Software changes are likely after the software enters production (e.g., to correct defects or introduce planned updates)
- The benefits of achieving maintainability objectives (see above) over the software lifecycle are considered by the affected stakeholders to outweigh the costs of performing the maintainability tests and making any required changes
- The risks of poor software maintainability (e.g., long response times to defects reported by users and/or customers) justify conducting maintainability tests

Appropriate techniques for maintainability testing include static analysis and reviews as discussed in Sections 3.2 and 5.2. Maintainability testing should be started as soon as the design documents are available and should continue throughout the code implementation effort. Since maintainability is built into the code and the documentation for each individual code component, maintainability can be evaluated early in the lifecycle without having to wait for a completed and running system.

Dynamic maintainability testing focuses on the documented procedures developed for maintaining a particular application (e.g., for performing software upgrades). Selections of maintenance scenarios are used as test cases to ensure the required service levels are attainable with the documented procedures. This form of testing is particularly relevant where the underlying infrastructure is complex, and support procedures may involve multiple departments/organizations. This form of testing may take place as part of Operational Acceptance Testing (OAT). [Web-1]

4.7.1 Analyzability, Changeability, Stability and Testability

The maintainability of a system can be measured in terms of the effort required to diagnose problems identified within a system (analyzability), implement the code changes (changeability) and test the changed system (testability). Stability relates specifically to the system's response to change. Systems with low stability exhibit large numbers of downstream problems (also known as the "ripple effect") whenever a change is made. [ISO9126] [Web-1].

The effort required to perform maintenance tasks is dependent on a number of factors such as software design methodology (e.g., object orientation) and coding standards used.

Note that "stability" in this context should not be confused with the terms "robustness" and "fault tolerance", which are covered in Section 4.4.2.

4.8 Portability Testing

Portability tests in general relate to the ease with which software can be transferred into its intended environment, either initially or from an existing environment. Portability tests include tests for installability, co-existence/compatibility, adaptability and replaceability. Portability testing can start with the individual components (e.g., replaceability of a particular component such as changing from one database management system to another) and will expand in scope as more code becomes available. Installability may not be testable until all the components of the product are functionally working. Portability must be designed and built into the product and so must be considered early in the design and architecture phases. Architecture and design reviews can be particularly productive for identifying potential portability requirements and issues (e.g., dependency on a particular operating system).

4.8.1 Installability Testing

Installability testing is conducted on the software and written procedures used to install the software on its target environment. This may include, for example, the software developed to install an operating system onto a processor, or an installation "wizard" used to install a product onto a client PC.

Typical installability testing objectives include:

- Validating that the software can be successfully installed by following the instructions in an installation manual (including the execution of any installation scripts), or by using an installation wizard. This includes exercising installation options for different hardware/software configurations and for various degrees of installation (e.g., initial or update).
- Testing whether failures which occur during installation (e.g., failure to load particular DLLs) are dealt with by the installation software correctly without leaving the system in an undefined state (e.g., partially installed software or incorrect system configurations)
- Testing whether a partial installation/de-installation can be completed
- Testing whether an installation wizard can successfully identify invalid hardware platforms or operating system configurations
- Measuring whether the installation process can be completed within a specified number of minutes or in less than a specified number of steps

- Validating that the software can be successfully downgraded or uninstalled

Functionality testing is normally conducted after the installation test to detect any faults which may have been introduced by the installation (e.g., incorrect configurations, functions not available). Usability testing is normally conducted in parallel with installability testing (e.g., to validate that users are provided with understandable instructions and feedback/error messages during the installation).

4.8.2 Co-existence/Compatibility Testing

Computer systems which are not related to each other are said to be compatible when they can run in the same environment (e.g., on the same hardware) without affecting each other's behavior (e.g., resource conflicts). Compatibility should be performed when new or upgraded software will be rolled out into environments which already contain installed applications.

Compatibility problems may arise when the application is tested in an environment where it is the only installed application (where incompatibility issues are not detectable) and then deployed onto another environment (e.g., production) which also runs other applications.

Typical compatibility testing objectives include:

- Evaluation of possible adverse impact on functionality when applications are loaded in the same environment (e.g., conflicting resource usage when a server runs multiple applications)
- Evaluation of the impact to any application resulting from the deployment of operating system fixes and upgrades

Compatibility issues should be analyzed when planning the targeted production environment but the actual tests are normally performed after system and user acceptance testing have been successfully completed.

4.8.3 Adaptability Testing

Adaptability testing checks whether a given application can function correctly in all intended target environments (hardware, software, middleware, operating system, etc.). An adaptive system is therefore an open system that is able to fit its behavior according to changes in its environment or in parts of the system itself. Specifying tests for adaptability requires that combinations of the intended target environments are identified, configured and available to the testing team. These environments are then tested using a selection of functional test cases which exercise the various components present in the environment.

Adaptability may relate to the ability of the software to be ported to various specified environments by performing a predefined procedure. Tests may evaluate this procedure.

Adaptability tests may be performed in conjunction with installability tests and are typically followed by functional tests to detect any faults which may have been introduced in adapting the software to a different environment.

4.8.4 Replaceability Testing

Replaceability testing focuses on the ability of software components within a system to be exchanged for others. This may be particularly relevant for systems which use commercial off-the-shelf (COTS) software for specific system components.

Replaceability tests may be performed in parallel with functional integration tests where more than one alternative component is available for integration into the complete system. Replaceability may be evaluated by technical review or inspection at the architecture and design levels, where the emphasis is placed on the clear definition of interfaces to potential replaceable components.

5. Reviews - 165 mins.

Keywords
anti-pattern

Learning Objectives for Reviews

5.1 Introduction

TTA 5.1.1 (K2) Explain why review preparation is important for the Technical Test Analyst

5.2 Using Checklists in Reviews

TTA 5.2.1 (K4) Analyze an architectural design and identify problems according to a checklist provided in the syllabus

TTA 5.2.2 (K4) Analyze a section of code or pseudo-code and identify problems according to a checklist provided in the syllabus

5.1 Introduction

Technical 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 technical review process. All review participants must be committed to the benefits of a well-conducted technical review. For a complete description of technical reviews, including numerous review checklists, see [Wiegers02]. Technical Test Analysts normally participate in technical reviews and inspections where they bring an operational (behavioral) viewpoint that may be missed by developers. In addition, Technical Test Analysts play an important role in the definition, application, and maintenance of review checklists and defect severity information.

Regardless of the type of review being conducted, the Technical Test Analyst must be allowed 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 review can become an editing exercise rather than a true review. 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 Technical Test Analyst must also consider the items that are being integrated. Are they 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 not unusual for the authors of a product being reviewed to feel they are being criticized. The Technical Test Analyst should be sure to approach any review comments from the viewpoint 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 don’t 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 Technical 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 or is not clear, then this is likely 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 verify 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, and we are not targeting only your work product." Checklists can be generic and used for all reviews or focused on specific quality characteristics or areas. For example, a generic checklist might verify the proper usage of the terms "shall" and "should", verify proper formatting and similar conformance items. A targeted checklist might concentrate on security issues or performance issues.

The most useful checklists are those gradually developed by an individual organization, because they reflect:

- The nature of the product
- The local development environment
  - Staff
  - Tools

  - Priorities
- History of previous successes and defects
- Particular issues (e.g., performance, security)

Checklists should be customized for the organization and perhaps for the particular project. The checklists provided in this chapter are meant only to serve as examples.

Some organizations extend the usual notion of a software checklist to include "anti-patterns" that refer to common mistakes, poor techniques, and other ineffective practices. The term derives from the popular concept of "design patterns" which are reusable solutions to common problems that have been shown to be effective in practical situations [Gamma94]. An anti-pattern, then, is a commonly made mistake, often implemented as an expedient short-cut.

It is important to remember that if a requirement is not testable, meaning that it is not defined in such a way that the Technical Test Analyst can determine how to test it, then it is a defect. For example, a requirement that states "The software should be fast" cannot be tested. How can the Technical Test Analyst determine if the software is fast? If, instead, the requirement said "The software must provide a maximum response time of three seconds under specific load conditions", then the testability of this requirement is substantially better, if we define the "specific load conditions" (e.g. number of concurrent users, activities performed by the users). It is also an overarching requirement because this one requirement could easily spawn many individual test cases in a non-trivial application. Traceability from this requirement to the test cases is also critical because if the requirement should change, all the test cases will need to be reviewed and updated as needed.

5.2.1 Architectural Reviews

Software architecture consists of the fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution. [ANSI/IEEE Std 1471-2000], [Bass03].

Checklists used for architecture reviews could, for example, include verification of the proper implementation of the following items, which are quoted from [Web-3]:

- "Connection pooling - reducing the execution time overhead associated with establishing database connections by establishing a shared pool of connections
- Load balancing – spreading the load evenly between a set of resources
- Distributed processing
- Caching – using a local copy of data to reduce access time
- Lazy instantiation
- Transaction concurrency
- Process isolation between Online Transactional Processing (OLTP) and Online Analytical Processing (OLAP)
- Replication of data"

More details, (not relevant for the certification exam), can be found in [Web-4], which refers to a paper that surveys 117 checklists from 24 sources. Different categories of checklist items are discussed and examples are provided of good checklist items as well as those that should be avoided.

5.2.2 Code Reviews

Checklists for code reviews are necessarily very detailed, and, as with checklists for architecture reviews, are most useful when they are language, project and company-specific. The inclusion of code-level anti-patterns is helpful, particularly for less experienced software developers.

Checklists used for code reviews could include the following items six items: (based on [Web-5]).

1. Structure
   - Does the code completely and correctly implement the design?
   - Does the code conform to any pertinent coding standards?
   - Is the code well-structured, consistent in style, and consistently formatted?
   - Are there any uncalled or unneeded procedures or any unreachable code?
   - Are there any leftover stubs or test routines in the code?
   - Can any code be replaced by calls to external reusable components or library functions?
   - Are there any blocks of repeated code that could be condensed into a single procedure?
   - Is storage use efficient?
   - Are symbolics used rather than “magic number” constants or string constants?
   - Are any modules excessively complex and should be restructured or split into multiple modules?

2. Documentation
   - Is the code clearly and adequately documented with an easy-to-maintain commenting style?
   - Are all comments consistent with the code?
   - Does the documentation conform to applicable standards?

3. Variables
   - Are all variables properly defined with meaningful, consistent, and clear names?
   - Are there any redundant or unused variables?

4. Arithmetic Operations
   - Does the code avoid comparing floating-point numbers for equality?
   - Does the code systematically prevent rounding errors?
   - Does the code avoid additions and subtractions on numbers with greatly different magnitudes?
   - Are divisors tested for zero or noise?

5. Loops and Branches
   - Are all loops, branches, and logic constructs complete, correct, and properly nested?
   - Are the most common cases tested first in IF-ELSEIF chains?
   - Are all cases covered in an IF-ELSEIF or CASE block, including ELSE or DEFAULT clauses?
   - Does every case statement have a default?
   - Are loop termination conditions obvious and invariably achievable?
   - Are indices or subscripts properly initialized, just prior to the loop?
   - Can any statements that are enclosed within loops be placed outside the loops?
   - Does the code in the loop avoid manipulating the index variable or using it upon exit from the loop?

6. Defensive Programming
   - Are indices, pointers, and subscripts tested against array, record, or file bounds?
   - Are imported data and input arguments tested for validity and completeness?
   - Are all output variables assigned?
   - Is the correct data element operated on in each statement?
   - Is every memory allocation released?
   - Are timeouts or error traps used for external device access?
   - Are files checked for existence before attempting to access them?
   - Are all files and devices left in the correct state upon program termination?

For further examples of checklists used for code reviews at different testing levels see [Web-6].

6. Test Tools and Automation - 195 mins.

Keywords

data-driven testing, debugging tool, fault seeding tool, hyperlink test tool, keyword-driven testing, performance testing tool, record/playback tool, static analyzer, test execution tool, test management tool

Learning Objectives for Test Tools and Automation

6.1 Integration and Information Interchange Between Tools

TTA-6.1.1 (K2) Explain technical aspects to consider when multiple tools are used together

6.2 Defining the Test Automation Project

TTA-6.2.1 (K2) Summarize the activities that the Technical Test Analyst performs when setting up a test automation project
TTA-6.2.2 (K2) Summarize the differences between data-driven and keyword-driven automation
TTA-6.2.3 (K2) Summarize common technical issues that cause automation projects to fail to achieve the planned return on investment
TTA-6.2.4 (K3) Create a keyword table based on a given business process

6.3 Specific Test Tools

TTA-6.3.1 (K2) Summarize the purpose of tools for fault seeding and fault injection
TTA-6.3.2 (K2) Summarize the main characteristics and implementation issues for performance testing and monitoring tools
TTA-6.3.3 (K2) Explain the general purpose of tools used for web-based testing
TTA-6.3.4 (K2) Explain how tools support the concept of model-based testing
TTA-6.3.5 (K2) Outline the purpose of tools used to support component testing and the build process

6.1 Integration and Information Interchange Between Tools

While the responsibility for selecting and integrating tools belongs to the Test Manager, the Technical Test Analyst may be called upon to review the integration of a tool or set of tools to ensure accurate tracking of data resulting from various testing areas such as static analysis, test execution automation and configuration management. In addition, depending on the programming skills of the Technical Test Analyst, there may also be involvement in creating the code that will integrate tools together that do not integrate "out of the box".

An ideal toolset should eliminate the duplication of information across the tools. It takes more effort and it is more error-prone to store test execution scripts both in a test management database and in the configuration management system. It would be better to have a test management system that includes a configuration management component or that is able to integrate with a configuration management tool already in place in the organization. Well-integrated defect tracking and test management tools will allow a tester to launch a defect report during test case execution without having to leave the test management tool. Well-integrated static analysis tools should be able to report any discovered incidents and warnings directly to the defect management system (although this should be configurable due to the many warnings that may be generated).

Buying a suite of test tools from the same vendor does not automatically mean that the tools work together adequately. When considering the approach to integrating tools together, a data-centric focus is preferable. The data must be exchanged without manual intervention in a timely manner with guaranteed accuracy including fault recovery. While it is helpful to have a consistent user experience, capture, storage, protection and presentation of data should be the primary focus of tool integration.

An organization should evaluate the cost of automating the information interchange compared to the risk of losing information or allowing the data to get out of synchronization due to necessary manual intervention. As integration may be expensive or difficult, it should be a prime consideration in the overall tool strategy.

Some integrated development environments (IDE) may simplify integration between tools that are capable of working in that environment. This helps to unify the look and feel of the tools that share the same framework. However, a similar user interface will not guarantee a smooth information exchange between components. Coding may be required to complete the integration.

6.2 Defining the Test Automation Project

In order to be cost-effective, test tools and particularly test automation tools, must be carefully architected and designed. Implementing a test automation strategy without a solid architecture usually results in a tool set that is costly to maintain, insufficient for the purpose and unable to achieve the target return on investment.

A test automation project should be considered a software development project. This includes the need for architecture documents, detailed design documents, design and code reviews, component and component integration testing as well as final system testing. Testing can be needlessly delayed or complicated when unstable or inaccurate test automation code is used. There are multiple activities that the Technical Test Analyst performs regarding test automation. These include:

- Determining who will be responsible for the test execution
- Selecting the appropriate tool for the organization, timeline, skills of the team, maintenance requirements (note this could mean deciding to create a tool to use rather than acquiring one)
- Defining the interface requirements between the automation tool and other tools such as the test management and defect management tools

- Selecting the automation approach, i.e., keyword-driven or data-driven (see Section 6.2.1 below)
- Working with the Test Manager to estimate the cost of the implementation, including training
- Scheduling the automation project and allocating the time for maintenance
- Training the Test Analysts and Business Analysts to use and supply data for the automation
- Determining how the automated tests will be executed
- Determining how the automated test results will be combined with the manual test results

These activities and the resulting decisions will influence the scalability and maintainability of the automation solution. Sufficient time must be spent researching the options, investigating available tools and technologies and understanding the future plans for the organization. Some of these activities require more consideration than others, particularly during the decision process. These are discussed in more detail in the following sections.

6.2.1 Selecting the Automation Approach

Test automation is not limited to testing through the GUI. Tools exist to help automate testing at the API level, through a Command Line Interface (CLI) and other interface points in the software under test. One of the first decisions the Technical Test Analyst must make is the most effective interface to be accessed to automate the testing.

One of the difficulties of testing through the GUI is the tendency for the GUI to change as the software evolves. Depending on the way the test automation code is designed, this can result in a significant maintenance burden. For example, using the record/playback capability of a test automation tool may result in automated test cases (often called test scripts) that no longer run as desired if the GUI changes. This is because the recorded script captures interactions with the graphical objects when the tester executes the software manually. If the objects being accessed change, the recorded scripts may also need updating to reflect those changes.

Capture and Playback tools may be used as a convenient starting point for developing automation scripts. The tester records a test session and recorded script is then modified to improve maintainability (e.g., by replacing sections in the recorded script with reusable functions).

Depending on the software being tested, the data used for each test may be different although the executed test steps are virtually identical (e.g., testing error handling for an input field by entering multiple invalid values and checking the error returned for each). It is inefficient to develop and maintain an automated test script for each of these values to be tested. A common technical solution to this problem is to move the data from the scripts to an external store such as a spreadsheet or a database. Functions are written to access the specific data for each execution of the test script, which enables a single script to work through a set of test data that supplies the input values and expected result values (e.g., a value shown in a text field or an error message). This approach is called data-driven. When using this approach, a test script is developed that will process the supplied data, as well as a harness and the infrastructure needed to support the execution of the script or set of scripts. The actual data held in the spreadsheet or database is created by Test Analysts who are familiar with the business function of the software. This division of labor allows those responsible for developing test scripts (e.g., the Technical Test Analyst) to focus on the implementation of intelligent automation scripts while the Test Analyst maintains ownership of the actual test. In most cases, the Test Analyst will be responsible for executing the test scripts once the automation is implemented and tested.

Another approach, called keyword- or action word-driven, goes a step further by also separating the action to be performed on the supplied data from the test script [Buwalda01]. In order to accomplish this further separation, a high-level meta language is created by domain experts (e.g., Test Analysts) which is descriptive rather than directly executable. Each statement of this language describes a full or partial business process of the domain that may require testing. For example, business process

keywords could include "Login", "CreateUser", and "DeleteUser". A keyword describes a high-level action that will be performed in the application domain. Lower level actions which denote interaction with the software interface itself, such as: “ClickButton”, “SelectFromList”, or “TraverseTree” may also be defined and may be used to test GUI capabilities that do not neatly fit into business process keywords.

Once the keywords and data to be used have been defined, the test automator (e.g., Technical Test Analyst) translates the business process keywords and lower level actions into test automation code. The keywords and actions, along with the data to be used, may be stored in spreadsheets or entered using specific tools which support keyword-driven test automation. The test automation framework implements the keyword as a set of one or more executable functions or scripts. Tools read test cases written with keywords and call the appropriate test functions or scripts which implement them. The executables are implemented in a highly modular manner to enable easy mapping to specific keywords. Programming skills are needed to implement these modular scripts.

This separation of the knowledge of the business logic from the actual programming required to implement the test automation scripts provides the most effective use of the test resources. The Technical Test Analyst, in the role as the test automator, can effectively apply programming skills without having to become a domain expert across many areas of the business.

Separating the code from the changeable data helps to insulate the automation from changes, improving the overall maintainability of the code and improving the return on the automation investment.

In any test automation design, it is important to anticipate and handle software failures. If a failure occurs, the automator must determine what the software should do. Should the failure be recorded and the tests continue? Should the tests be terminated? Can the failure be handled with a specific action (such as clicking a button in a dialog box) or perhaps by adding a delay in the test? Unhandled software failures may corrupt subsequent test results as well as causing a problem with the test that was executing when the failure occurred.

It is also important to consider the state of the system at the start and end of the tests. It may be necessary to ensure the system is returned to a pre-defined state after the test execution is completed. This will allow a suite of automated tests to be run repeatedly without manual intervention to reset the system to a known state. To do this, the test automation may have to, for example, delete the data it created or alter the status of records in a database. The automation framework should ensure that a proper termination has been accomplished at the end of the tests (i.e., logging out after the tests complete).

6.2.2 Modeling Business Processes for Automation

In order to implement a keyword-driven approach for test automation, the business processes to be tested must be modeled in the high-level keyword language. It is important that the language is intuitive to its users who are likely to be the Test Analysts working on the project.

Keywords are generally used to represent high-level business interactions with a system. For example, “Cancel_Order” may require checking the existence of the order, verifying the access rights of the person requesting the cancellation, displaying the order to be cancelled and requesting confirmation of the cancellation. Sequences of keywords (e.g., “Login”, “Select_Order”, “Cancel_Order”), and the relevant test data are used by the Test Analyst to specify test cases. The following is a simple keyword-driven input table that could be used to test the ability of the software to add, reset and delete user accounts:

Keyword	User	Password	Result
Add_User	User1	Pass1	User added message
Add_User	@Rec34	@Rec35	User added message
Reset_Password	User1	Welcome	Password reset confirmation message
Delete_User	User1		Invalid username/password message
Add_User	User3	Pass3	User added message
Delete_User	User2		User not found message

The automation script that uses this table would look for the input values to be used by the automation script. For example, when it gets to the row with the keyword “Delete_User”, only the user name is required. To add a new user both user name and password are required. Input values may also be referenced from a data store as shown with the second "Add_User" keyword where a reference to the data is entered rather than the data itself providing more flexibility to access data that may be changing as the tests execute. This allows data-driven techniques to be combined with the keyword scheme.

Issues to consider include:

- The more granular the keywords, the more specific the scenarios that can be covered, but the high-level language may become more complex to maintain.
- Allowing Test Analysts to specify low-level actions ("ClickButton", "SelectFromList", etc.) makes the keyword tests much more capable of handling different situations. However, because these actions are tied directly to the GUI, it also may cause the tests to require more maintenance when changes occur.
- Use of aggregated keywords may simplify development but complicate maintenance. For example, there may be six different keywords that collectively create a record. Should a keyword that actually calls all six keywords consecutively be created to simplify that action?
- No matter how much analysis goes into the keyword language, there will often be times when new and different keywords will be needed. There are two separate domains to a keyword (i.e., the business logic behind it and the automation functionality to execute it). Therefore, a process must be created to deal with both domains.

Keyword-based test automation can significantly reduce the maintenance costs of test automation, but it is more costly, more difficult to develop, and takes more time to design correctly in order to gain the expected return on investment.

6.3 Specific Test Tools

This section contains information on tools that are likely to be used by a Technical Test Analyst beyond what is discussed in the Advanced Level Test Analyst [ISTQB_ALTA_SYL] and Foundation Level [ISTQB_FL_SYL] syllabi.

6.3.1 Fault Seeding/Fault Injection Tools

Fault seeding tools are mainly used at the code level to create single or limited types of code faults in a systematic way. These tools deliberately insert defects into the test object for the purpose of evaluating the quality of the test suites (i.e., their ability to detect the defects).

Fault injection is focused on testing any fault handling mechanism built into the test object by subjecting it to abnormal conditions. Fault injection tools deliberately supply incorrect inputs to the software to ensure the software can cope with the fault.

Both of these types of tools are generally used the Technical Test Analyst, but may also be used by the developer when testing newly developed code.

6.3.2 Performance Testing Tools

Performance test tools have two main functions:

- Load generation
- Measurement and analysis of the system response to a given load

Load generation is performed by implementing a pre-defined operational profile (see Section 4.5.4) as a script. The script may initially be captured for a single user (possibly using a record/playback tool) and is then implemented for the specified operational profile using the performance test tool. This implementation must take into account the variation of data per transaction (or sets of transactions).

Performance tools generate a load by simulating large numbers of multiple users ("virtual" users) following their designated operational profiles to generate specific volumes of input data. In comparison with individual test execution automation scripts, many performance testing scripts reproduce user interaction with the system at the communications protocol level and not by simulating user interaction via a graphical user interface. This usually reduces the number of separate "sessions" needed during the testing. Some load generation tools can also drive the application using its user interface to more closely measure response time while the system is under load.

A wide range of measurements are taken by a performance test tool to enable analysis during or after execution of the test. Typical metrics taken and reports provided include:

- Number of simulated users throughout the test
- Number and type of transactions generated by the simulated users and the arrival rate of the transactions
- Response times to particular transaction requests made by the users
- Reports and graphs of load against response times
- Reports on resource usage (e.g., usage over time with minimum and maximum values)

Significant factors to consider in the implementation of performance test tools include:

- The hardware and network bandwidth required to generate the load
- The compatibility of the tool with the communications protocol used by the system under test
- The flexibility of the tool to allow different operational profiles to be easily implemented
- The monitoring, analysis and reporting facilities required

Performance test tools are typically acquired rather than developed in-house due to the effort required to develop them. It may, however, be appropriate to develop a specific performance tool if technical restrictions prevent an available product being used, or if the load profile and facilities to be provided are relatively simple.

6.3.3 Tools for Web-Based Testing

A variety of open source and commercial specialized tools are available for web testing. The following list shows the purpose of some of the common web-based testing tools:

- Hyperlink test tools are used to scan and check that no broken or missing hyperlinks are present on a web site
- HTML and XML checkers are tools which check compliance to the HTML and XML standards of the pages that are created by a web site
- Load simulators to test how the server will react when large numbers of users connect
- Lightweight automation execution tools that work with different browsers
- Tools to scan through the server, checking for orphaned (unlinked) files
- HTML specific spell checkers
- Cascading Style Sheet (CSS) checking tools
- Tools to check for standards violations e.g., Section 508 accessibility standards in the U.S. or M/376 in Europe

- Tools that find a variety of security issues

A good source of open source web testing tools is [Web-7]. The organization behind this web site sets the standards for the Internet and it supplies a variety of tools to check for errors against those standards.

Some tools that include a web spider engine can also provide information on the size of the pages and on the time necessary to download them, and on whether a page is present or not (e.g., HTTP error 404). This provides useful information for the developer, the webmaster and the tester.

Test Analysts and Technical Test Analysts use these tools primarily during system testing.

6.3.4 Tools to Support Model-Based Testing

Model-Based Testing (MBT) is a technique whereby a formal model such as a finite state machine is used to describe the intended execution-time behavior of a software-controlled system. Commercial MBT tools (see [Utting 07]) often provide an engine that allows a user to "execute" the model. Interesting threads of execution can be saved and used as test cases. Other executable models such as Petri Nets and Statecharts also support MBT. MBT models (and tools) can be used to generate large sets of distinct execution threads.

MBT tools can help reduce the very large number of possible paths that can be generated in a model.

Testing using these tools can provide a different view of the software to be tested. This can result in the discovery of defects that might have been missed by functional testing.

6.3.5 Component Testing and Build Tools

While component testing and build automation tools are developer tools, in many instances, they are used and maintained by Technical Test Analysts, especially in the context of Agile development.

Component testing tools are often specific to the language that is used for programming a module. For example, if Java was used as the programming language, JUnit might be used to automate the unit testing. Many other languages have their own special test tools; these are collectively called xUnit frameworks. Such a framework generates test objects for each class that is created, thus simplifying the tasks that the programmer needs to do when automating the component testing.

Debugging tools facilitate manual component testing at a very low level, allowing developers and Technical Test Analysts to change variable values during execution and step through the code line by line while testing. Debugging tools are also used to help the developer isolate and identify problems in the code when a failure is reported by the test team.

Build automation tools often allow a new build to be automatically triggered any time a component is changed. After the build is completed, other tools automatically execute the component tests. This level of automation around the build process is usually seen in a continuous integration environment.

When set up correctly, this set of tools can have a very positive effect on the quality of builds being released into testing. Should a change made by a programmer introduce regression defects into the build, it will usually cause some of the automated tests to fail, triggering immediate investigation into the cause of the failures before the build is released into the test environment.

7. References

7.1 Standards

The following standards are mentioned in these respective chapters.

- ANSI/IEEE Std 1471-2000, Recommended Practice for Architectural Description of Software-Intensive Systems
  Chapter 5

- IEC-61508
  Chapter 2

- [ISO25000] ISO/IEC 25000:2005, Software Engineering - Software Product Quality Requirements and Evaluation (SQuaRE)
  Chapter 4

- [ISO9126] ISO/IEC 9126-1:2001, Software Engineering – Software Product Quality
  Chapter 4

- [RTCA DO-178B/ED-12B]: Software Considerations in Airborne Systems and Equipment Certification, RTCA/EUROCAE ED12B.1992.
  Chapter 2

7.2 ISTQB Documents

- [ISTQB_AL_OVIEW] ISTQB Advanced Level Overview, Version 2012
- [ISTQB_ALTA_SYL] ISTQB Advanced Level Test Analyst Syllabus, Version 2012
- [ISTQB_FL_SYL] ISTQB Foundation Level Syllabus, Version 2011
- [ISTQB_GLOSSARY] ISTQB Glossary of Terms used in Software Testing, Version 2.2, 2012

7.3 Books

[Bass03] Len Bass, Paul Clements, Rick Kazman “Software Architecture in Practice (2nd edition)”, Addison-Wesley 2003] ISBN 0-321-15495-9
[Bath08] Graham Bath, Judy McKay, “The Software Test Engineer’s Handbook”, Rocky Nook, 2008, ISBN 978-1-933952-24-6
[Beizer90] Boris Beizer, "Software Testing Techniques Second Edition", International Thomson Computer Press, 1990, ISBN 1-8503-2880-3
[Beizer95] Boris Beizer, “Black-box Testing”, John Wiley & Sons, 1995, ISBN 0-471-12094-4
[Buwalda01]: Hans Buwalda, “Integrated Test Design and Automation” Addison-Wesley Longman, 2001, ISBN 0-201-73725-6
[Copeland03]: Lee Copeland, “A Practitioner's Guide to Software Test Design”, Artech House, 2003, ISBN 1-58053-791-X
[Gamma94] Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1994, ISBN 0-201-63361-2
[Jorgensen07]: Paul C. Jorgensen, “Software Testing, a Craftsman’s Approach third edition”, CRC press, 2007, ISBN-13:978-0-8493-7475-3

[Kaner02]: Cem Kaner, James Bach, Bret Pettichord; “Lessons Learned in Software Testing”; Wiley, 2002, ISBN: 0-471-08112-4
[Koomen06]: Tim Koomen, Leo van der Aalst, Bart Broekman, Michael Vroon, "TMap Next for result-driven testing"; UTN Publishers, 2006, ISBN: 90-72194-79-9
[McCabe76] Thomas J. McCabe, "A Complexity Measure", IEEE Transactions on Software Engineering, Vol. SE-2, No. 4, December 1976. PP 308-320
[NIST96] Arthur H. Watson and Thomas J. McCabe, "Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric", NIST Special Publication 500-235, Prepared under NIST Contract 43NANB517266, September 1996.
[Splaine01]: Steven Splaine, Stefan P. Jaskiel, “The Web-Testing Handbook”, STQE Publishing, 2001, ISBN 0-970-43630-0
[Utting 07] Mark Utting, Bruno Legeard, “Practical Model-Based Testing: A Tools Approach”, Morgan-Kaufmann, 2007, ISBN: 978-0-12-372501-1
[Whittaker04]: James Whittaker and Herbert Thompson, “How to Break Software Security”, Pearson / Addison-Wesley, 2004, ISBN 0-321-19433-0
[Wiegers02] Karl Wiegers, "Peer Reviews in Software: A Practical Guide", Addison-Wesley, 2002, ISBN 0-201-73485-0

7.4 Other References

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

[Web-1] www.testingstandards.co.uk
[Web-2] http://www.nist.gov NIST National Institute of Standards and Technology,
[Web-3] http://www.codeproject.com/KB/architecture/SWArchitectureReview.aspx
[Web-4] http://portal.acm.org/citation.cfm?id=308798
[Web-5] http://www.processimpact.com/pr_goodies.shtml
[Web-6]http://www.ifsq.org
[Web-7] http://www.W3C.org

Chapter 4: [Web-1], [Web-2]
Chapter 5: [Web-3], [Web-4], [Web-5], [Web-6]
Chapter 6: [Web-7]

8. Index