Certified Tester Advanced Level Technical Test Analyst (CTAL-TTA) Syllabus

v4.0

International Software Testing Qualifications Board

Copyright Notice

Revision History

Version	Date	Remarks
v4.0	2021/06/30	GA release for v4.0 version
v4.0	2021/04/28	Draft updated based on feedback from Beta Review.
2021 v4.0 Beta	2021/03/01	Draft updated based on feedback from Alpha Review.
2021 v4.0 Alpha	2020/12/07	Draft for Alpha Review updated to: • Improve text throughout • Remove subsection associated with K3 TTA-2.6.1 (2.6 Basis Path Testing) and remove LO • Remove subsection associated with K2 TTA-3.2.4 (3.2.4 Call Graphs) and remove LO • Rewrite subsection associated with TTA-3.2.2 (3.2.2 Data Flow Analysis) and make it a K3 • Rewrite section associated with TTA-4.4.1 and TTA-4.4.2 (4.4. Reliability Testing) • Rewrite section associated with TTA-4.5.1 and TTA-4.5.2 (4.5 Performance Testing) • Add section 4.9 on Operational Profiles. • Rewrite section associated with TTA-2.8.1 (2.7 Selecting White-Box Test Techniques section) • Rewrite TTA-3.2.1 to include cyclomatic complexity (no impact on exam Qs) • Rewrite TTA-2.4.1 (MC/DC) to make it consistent with other white-box LOs (no impact on exam Qs)
2019 v1.0	2019/10/18	GA release for 2019 version
2012	2012/10/19	GA release for 2012 version

Table of Contents

Copyright Notice 2
Revision History 3
Table of Contents 4
0. Introduction to this Syllabus 7
0.1 Purpose of this Syllabus 7
0.2 The Certified Tester Advanced Level in Software Testing 7
0.3 Examinable Learning Objectives and Cognitive Levels of Knowledge 7
0.4 Expectations of Experience 7
0.5 The Advanced Level Technical Test Analyst Exam 8
0.6 Entry Requirements for the Exam 8
0.7 Accreditation of Courses 8
0.8 Level of Syllabus Detail 8
0.9 How this Syllabus is Organized 9
1. The Technical Test Analyst's Tasks in Risk-Based Testing - 30 mins. 10
1.1 Introduction 11
1.2 Risk-based Testing Tasks 11
1.2.1 Risk Identification 11
1.2.2 Risk Assessment 11
1.2.3 Risk Mitigation 12
2. White-Box Test Techniques - 300 mins. 13
2.1 Introduction 14
2.2 Statement Testing 14
2.3 Decision Testing 15
2.4 Modified Condition/Decision Testing 15
2.5 Multiple Condition Testing 16
2.6 Basis Path Testing 17
2.7 API Testing 17
2.8 Selecting a White-Box Test Technique 18
2.8.1 Non-Safety-Related Systems 19
2.8.2 Safety-related systems 19
3. Static and Dynamic Analysis - 180 mins. 21
3.1 Introduction 22
3.2 Static Analysis 22
3.2.1 Control Flow Analysis 22
3.2.2 Data Flow Analysis 22
3.2.3 Using Static Analysis for Improving Maintainability 23
3.3 Dynamic Analysis 24
3.3.1 Overview 24
3.3.2 Detecting Memory Leaks 24
3.3.3 Detecting Wild Pointers 25
3.3.4 Analysis of Performance Efficiency 25
4. Quality Characteristics for Technical Testing - 345 mins. 27
4.1 Introduction 28
4.2 General Planning Issues 29
4.2.1 Stakeholder Requirements 29
4.2.2 Test Environment Requirements 29
4.2.3 Required Tool Acquisition and Training 30
4.2.4 Organizational Considerations 30
4.2.5 Data Security and Data Protection 30
4.3 Security Testing 30
4.3.1 Reasons for Considering Security Testing 30
4.3.2 Security Test Planning 31

4.3.3 Security Test Specification ... 32
4.4 Reliability Testing ... 32
4.4.1 Introduction ... 32
4.4.2 Testing for Maturity ... 32
4.4.3 Testing for Availability ... 33
4.4.4 Testing for Fault Tolerance ... 33
4.4.5 Testing for Recoverability ... 34
4.4.6 Reliability Test Planning ... 34
4.4.7 Reliability Test Specification ... 35
4.5 Performance Testing ... 35
4.5.1 Introduction ... 35
4.5.2 Testing for Time Behavior ... 35
4.5.3 Testing for Resource Utilization ... 35
4.5.4 Testing for Capacity ... 36
4.5.5 Common Aspects of Performance Testing ... 36
4.5.6 Types of Performance Testing ... 36
4.5.7 Performance Test Planning ... 37
4.5.8 Performance Test Specification ... 38
4.6 Maintainability Testing ... 38
4.6.1 Static and Dynamic Maintainability Testing ... 38
4.6.2 Maintainability Sub-characteristics ... 39
4.7 Portability Testing ... 39
4.7.1 Introduction ... 39
4.7.2 Installability Testing ... 39
4.7.3 Adaptability Testing ... 40
4.7.4 Replaceability Testing ... 40
4.8 Compatibility Testing ... 40
4.8.1 Introduction ... 40
4.8.2 Coexistence Testing ... 40
4.9 Operational Profiles ... 41
5. Reviews - 165 mins ... 42
5.1 Technical Test Analyst Tasks in Reviews ... 43
5.2 Using Checklists in Reviews ... 43
5.2.1 Architectural Reviews ... 43
5.2.2 Code Reviews ... 44
6. Test Tools and Automation - 180 mins. ... 46
6.1 Defining the Test Automation Project ... 47
6.1.1 Selecting the Automation Approach ... 47
6.1.2 Modeling Business Processes for Automation ... 49
6.2 Specific Test Tools ... 50
6.2.1 Fault Seeding Tools ... 50
6.2.2 Fault Injection Tools ... 50
6.2.3 Performance Testing Tools ... 50
6.2.4 Tools for Testing Websites ... 51
6.2.5 Tools to Support Model-Based Testing ... 52
6.2.6 Component Testing and Build Tools ... 52
6.2.7 Tools to Support Mobile Application Testing ... 52
7. References ... 54
7.1 Standards ... 54
7.2 ISTQB® Documents ... 54
7.3 Books and articles ... 55
7.4 Other References ... 55
8. Appendix A: Quality Characteristics Overview ... 56
9. Index ... 58

Acknowledgements

The 2019 version of this document was produced by a core team from the International Software Testing Qualifications Board Advanced Level Working Group: Graham Bath (vice-chair), Rex Black, Judy McKay, Kenji Onoshi, Mike Smith (chair), Erik van Veenendaal.

The following persons participated in the reviewing, commenting, and balloting of the 2019 version of this syllabus:

Dani Almog	Andrew Archer	Rex Black
Armin Born	Sudeep Chatterjee	Tibor Csöndes
Wim Decoutere	Klaudia Dusser-Zieger	Melinda Eckrich-Brájer
Peter Foldhazi	David Frei	Karol Frühauf
Jan Giesen	Attila Gyuri	Matthias Hamburg
Tamás Horváth	N. Khimanand	Jan te Kock
Attila Kovács	Claire Lohr	Rik Marselis
Marton Matyas	Judy McKay	Dénes Medzihradszky
Petr Neugebauer	Ingvar Nordström	Pálma Polyák
Meile Posthuma	Stuart Reid	Lloyd Roden
Adam Roman	Jan Sabak	Péter Sótér
Benjamin Timmermans	Stephanie van Dijck	Paul Weymouth

This document was produced by a core team from the International Software Testing Qualifications Board Advanced Level Working Group: Armin Born, Adam Roman, Stuart Reid.

The updated v4.0 version of this document was produced by a core team from the International Software Testing Qualifications Board Advanced Level Working Group: Armin Born, Adam Roman, Christian Graf, Stuart Reid.

The following persons participated in the reviewing, commenting, and balloting of the updated v4.0 version of this syllabus:

Adél Vécsey-Juhász	Jane Nash	Pálma Polyák
Ágota Horváth	Lloyd Roden	Paul Weymouth
Benjamin Timmermans	Matthias Hamburg	Péter Földházi Jr.
Erwin Engelsma	Meile Posthuma	Rik Marselis
Gary Mogyorodi	Nishan Portoyan	Sebastian Małyska
Geng Chen	Joan Killeen	Tal Pe'er
Gergely Ágnecz	Ole Chr. Hansen	Wang Lijuan
		Zuo Zhenlei

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

This document was formally released by the General Assembly of the ISTQB® on 30 June 2021.

0. Introduction to this Syllabus

0.1 Purpose of this Syllabus

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 the 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 The Certified Tester Advanced Level in Software Testing

The Advanced Level qualification is comprised of three separate syllabi relating to the following roles:

- Test Manager
- Test Analyst
- Technical Test Analyst

The ISTQB® Technical Test Analyst Advanced Level Overview is a separate document [CTAL_TTA_OVIEW] which includes the following information:

- Business Outcomes
- Matrix showing traceability between business outcomes and learning objectives
- Summary

0.3 Examinable Learning Objectives and Cognitive Levels of Knowledge

The Learning Objectives support the Business Outcomes and are used to create the examination for achieving the Advanced Technical Test Analyst Certification.

The knowledge levels of the specific learning objectives at K2, K3 and K4 levels are shown at the beginning of each chapter and are classified as follows:

- K2: Understand
- K3: Apply
- K4: Analyze

0.4 Expectations of Experience

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

0.5 The Advanced Level Technical Test Analyst Exam

The Advanced Level Technical Test Analyst exam will be based on this syllabus. Answers to exam questions may require the use of materials based on more than one section of this syllabus. All sections of the syllabus are examinable except for the introduction and the appendices. Standards, books and other ISTQB® syllabi are included as references, but their content is not examinable beyond what is summarized in this syllabus itself.

The format of the exam is multiple choice. There are 45 questions. To pass the exam, at least 65% of the total points must be earned.

Exams may be taken as part of an accredited training course or taken independently (e.g., at an exam center or in a public exam). Completion of an accredited training course is not a pre-requisite for the exam.

0.6 Entry Requirements for the Exam

The Certified Tester Foundation Level certification shall be obtained before taking the Advanced Level Technical Test Analyst certification exam.

0.7 Accreditation of Courses

An ISTQB® Member Board may accredit training providers whose course material follows this syllabus. Training providers should obtain accreditation guidelines from the Member Board or body that performs the accreditation. An accredited course is recognized as conforming to this syllabus and is allowed to have an ISTQB® exam as part of the course.

0.8 Level of Syllabus Detail

The level of detail in this syllabus allows internationally consistent courses and exams. To achieve this goal, the syllabus consists of:

- General instructional objectives describing the intention of the Advanced Level Technical Test Analyst
- A list of terms that students must be able to recall
- Learning objectives for each knowledge area, describing the cognitive learning outcome to be achieved
- A description of the key concepts, including references to sources such as accepted literature or standards

The syllabus content is not a description of the entire knowledge area; it reflects the level of detail to be covered in Advanced Level training courses. It focuses on material that can apply to any software projects, using any software development lifecycle. The syllabus does not contain any specific learning objectives relating to any particular software development models, but it does discuss how these concepts apply in Agile software development, other types of iterative and incremental software development models, and in sequential software development models.

0.9 How this Syllabus is Organized

There are six chapters with examinable content. The top-level heading for each chapter specifies the minimum time for the chapter; timing is not provided below chapter level. For accredited training courses, the syllabus requires a minimum of 20 hours of instruction, distributed across the six chapters as follows:

- Chapter 1: The Technical Test Analyst's Tasks in Risk-Based Testing (30 minutes)
- Chapter 2: White-Box Test Techniques (300 minutes)
- Chapter 3: Static and Dynamic Analysis (180 minutes)
- Chapter 4: Quality Characteristics for Technical Testing (345 minutes)
- Chapter 5: Reviews (165 minutes)
- Chapter 6: Test Tools and Automation (180 minutes)

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

Keywords
product risk, project risk, risk assessment, risk identification, risk mitigation, risk-based testing

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

1.2 Risk-based Testing Tasks

TTA-1.2.1 (K2) Summarize the generic risk factors that the Technical Test Analyst typically needs to consider

TTA-1.2.2 (K2) Summarize the activities of the Technical Test Analyst within a risk-based approach for testing activities

1.1 Introduction

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

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 product risks that are inherent in the project, such as risks related to security, system reliability and performance. They should also contribute to the identification and treatment of project risks associated with test environments, such as the acquisition and set-up of test environments for performance, reliability, and security testing.

1.2 Risk-based Testing Tasks

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 risks and changing priorities, and to regularly evaluate and communicate risk status.

1.2.1 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 analyzing their experiences to determine where the likely areas of product risk lie. In particular, Technical Test Analysts work closely with other stakeholders, such as developers, architects, operations engineers, product owners, local support offices, technical experts, and service desk technicians, to determine areas of technical risk impacting the product and project. Involving other stakeholders ensures that all views are considered and is typically facilitated by Test Managers.

Risks that might be identified by the Technical Test Analyst are typically based on the [ISO 25010] product quality characteristics listed in Chapter 4 of this syllabus.

1.2.2 Risk Assessment

While risk identification is about identifying as many pertinent risks as possible, risk assessment is the study of those identified risks to categorize each risk and determine the likelihood and impact associated with it.

The likelihood of a product risk is usually interpreted as the probability of the occurrence of the failure in the system under test. The Technical Test Analyst contributes to understanding the probability of each technical product risk whereas the Test Analyst contributes to understanding the potential business impact of the problem should it occur.

Project risks that become issues can impact the overall success of the project. Typically, the following generic project risk factors need to be considered:

- Conflict between stakeholders regarding technical requirements
- Communication problems resulting from the geographical distribution of the development organization
- Tools and technology (including relevant skills)
- Time, resource, and management pressure
- Lack of earlier quality assurance

- High change rates of technical requirements

Product risks that become issues may result in higher numbers of defects. Typically, the following generic product risk factors need to be considered:

- Complexity of technology
- Code complexity
- Amount of change in the source code (insertions, deletions, modifications)
- Large number of defects found relating to technical quality characteristics (defect history)
- Technical interface and integration issues

Given the available risk information, the Technical Test Analyst proposes an initial risk likelihood according to the guidelines established by the Test Manager. The initial value may be modified by the Test Manager when all stakeholder views have been considered. The risk impact is normally determined by the Test Analyst.

1.2.3 Risk Mitigation

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

- Designing test cases for those risks addressing high risk areas and helping to evaluate the residual risk
- Reducing risk by executing the designed test cases and by putting into action appropriate mitigation and contingency measures as stated in the test plan
- Evaluating risks based on additional information gathered as the project unfolds, and using that information to implement mitigation measures aimed at decreasing the likelihood of those risks

The Technical Test Analyst will often cooperate with specialists in areas such as security and performance to define risk mitigation measures and elements of the test strategy. Additional information can be obtained from ISTQB® Specialist syllabi, such as the Security Testing syllabus [CT_SEC_SYL] and the Performance Testing syllabus [CT_PT_SYL].

2. White-Box Test Techniques - 300 mins.

Keywords
API testing, atomic condition, control flow, decision testing, modified condition/decision testing, multiple condition testing, safety integrity level, statement testing, white-box test technique

Learning Objectives for White-Box Test Techniques

2.2 Statement Testing
TTA-2.2.1 (K3) Design test cases for a given test object by applying statement testing to achieve a defined level of coverage

2.3 Decision Testing
TTA-2.3.1 (K3) Design test cases for a given test object by applying the Decision test technique to achieve a defined level of coverage

2.4 Modified Condition/Decision Testing
TTA-2.4.1 (K3) Design test cases for a given test object by applying the modified condition/decision test technique to achieve full modified condition/decision coverage (MC/DC)

2.5 Multiple Condition Testing
TTA-2.5.1 (K3) Design test cases for a given test object by applying the multiple condition test technique to achieve a defined level of coverage

2.6 Basis Path Testing (*has been removed from version v4.0 of this syllabus*)
TTA-2.6.1 has been removed from version v4.0 of this syllabus.

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 White-box Test Technique
TTA-2.8.1 (K4) Select an appropriate white-box test technique according to a given project situation

2.1 Introduction

This chapter describes white-box test techniques. These techniques apply to code and other structures with a control flow, such as business process flow charts.

Each specific technique enables test cases to be derived systematically and focuses on a particular aspect of the structure. The test cases generated by the techniques satisfy coverage criteria which are set as an objective and are measured against. Achieving full coverage (i.e. 100%) does not mean that the entire set of tests is complete, but rather that the technique being used no longer suggests any useful further tests for the structure under consideration.

Test inputs are generated to ensure a test case exercises a particular part of the code (e.g., a statement or decision outcome). Determining the test inputs that will cause a particular part of the code to be exercised can be challenging, especially if the part of the code to be exercised is at the end of a long control flow sub-path with several decisions on it. The expected results are identified based on a source external to this structure, such as a requirements or design specification or another test basis.

The following techniques are considered in this syllabus:

- Statement testing
- Decision testing
- Modified condition/decision testing
- Multiple condition testing
- API testing

The Foundation Syllabus [CTFL_SYL] introduces statement testing and decision testing. Statement testing focuses on exercising the executable statements in the code, whereas decision testing exercises the decision outcomes.

The modified condition/decision and multiple condition techniques listed above are based on decision predicates containing multiple conditions and find similar types of defects. No matter how complex a decision predicate may be, it will evaluate to either TRUE or FALSE, which will determine the path taken through the code. A defect is detected when the intended path is not taken because a decision predicate does not evaluate as expected.

Refer to [ISO 29119] for more details on the specification, and examples, of these techniques.

2.2 Statement Testing

Statement testing exercises the executable statements in the code. Coverage is measured as the number of statements executed by the tests divided by the total number of executable statements in the test object, normally expressed as a percentage.

Applicability
Achieving full statement coverage should be considered as a minimum for all code being tested, although this is not always possible in practice.

Limitations/Difficulties
Achieving full statement coverage should be considered as a minimum for all code being tested, although this is not always possible in practice due to constraints on the available time and/or effort. Even high percentages of statement coverage may not detect certain defects in the code's logic. In many cases achieving 100% statement coverage is not possible due to unreachable code. Although unreachable code is generally not considered good programming practice, it may occur, for instance, if a switch statement must have a default case, but all possible cases are handled explicitly.

2.3 Decision Testing

Decision testing exercises the decision outcomes in the code. To do this, the test cases follow the control flows from a decision point (e.g., for an IF statement, there is one control flow for the true outcome and one for the false outcome; for a CASE statement, there may be several possible outcomes; for a LOOP statement there is one control flow for the true outcome of the loop condition and one for the false outcome).

Coverage is measured as the number of decision outcomes exercised by the tests divided by the total number of decision outcomes in the test object, normally expressed as a percentage. Note that a single test case may exercise several decision outcomes.

Compared to the modified condition/decision and multiple condition techniques described below, decision testing considers the entire decision as a whole and evaluates only the TRUE and FALSE outcomes, regardless of the complexity of its internal structure.

Branch testing is often used interchangeably with decision testing, because covering all branches and covering all decision outcomes can be achieved with the same tests. Branch testing exercises the branches in the code, where a branch is normally considered to be an edge of the control flow graph. For programs with no decisions, the definition of decision coverage above results in a coverage of 0/0, which is undefined, no matter how many tests are run, while the single branch from entry to exit point (assuming one entry and exit point) will result in 100% branch coverage being achieved. To address this difference between the two measures, ISO 29119-4 requires at least one test to be run on code with no decisions to achieve 100% decision coverage, so making 100% decision coverage and 100% branch coverage equivalent for nearly all programs. Many test tools that provide coverage measures, including those used for testing safety-related systems, employ a similar approach.

Applicability
This level of coverage should be considered when the code being tested is important or even critical (see the tables in section 2.8.2 for safety-related systems). This technique can be used for code and for any model that involves decision points, like business process models.

Limitations/Difficulties
Decision testing does not consider the details of how a decision with multiple conditions is made and may fail to detect defects caused by combinations of the condition outcomes.

2.4 Modified Condition/Decision Testing

Compared to decision testing, which considers the entire decision as a whole and evaluates the TRUE and FALSE outcomes, modified condition/decision testing considers how a decision is structured when it includes multiple conditions (where a decision is composed of only one atomic condition, it is simply decision testing).

Each decision predicate is made up of one or more atomic conditions, each of which evaluates to a Boolean value. These are logically combined to determine the outcome of the decision. This technique checks that each of the atomic conditions independently and correctly affects the outcome of the overall decision.

This technique provides a stronger level of coverage than statement and decision coverage when there are decisions containing multiple conditions. Assuming N unique, mutually independent atomic conditions, MC/DC for a decision can usually be achieved by exercising the decision N+1 times. Modified condition/decision testing requires pairs of tests that show a change of a single atomic condition outcome can independently affect the result of a decision. Note that a single test case may exercise several condition combinations and therefore it is not always necessary to run N+1 separate test cases to achieve MC/DC.

Applicability
This technique is used in the aerospace and automotive industries, and other industry sectors for safety-critical systems. It is used when testing software where a failure may cause a catastrophe. Modified condition/decision testing can be a reasonable middle ground between decision testing and multiple condition testing (due to the large number of combinations to test). It is more rigorous than decision testing but requires far fewer test conditions to be exercised than multiple condition testing when there are several atomic conditions in the decision.

Limitations/Difficulties
Achieving MC/DC may be complicated when there are multiple occurrences of the same variable in a decision with multiple conditions; when this occurs, the conditions may be "coupled". Depending on the decision, it may not be possible to vary the value of one condition such that it alone causes the decision outcome to change. One approach to addressing this issue is to specify that only uncoupled atomic conditions are tested using modified condition/decision testing . The other approach is to analyze each decision in which coupling occurs.

Some compilers 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 has already been evaluated as FALSE. No value of B can change the final result, so the code may save execution time by not evaluating B. Short-circuiting may affect the ability to attain MC/DC since some required tests may not be achievable. Usually, it is possible to configure the compiler to switch off the short-circuiting optimization for the testing, but this may not be allowed for safety-critical applications, where the tested code and the delivered code must be identical.

2.5 Multiple Condition Testing

In rare instances, it might be required to test all possible combinations of atomic conditions that a decision may contain. This level of testing is called multiple condition testing. Assuming N unique, mutually independent atomic conditions, full multiple condition coverage for a decision can be achieved by exercising it 2^N times. Note that a single test case may exercise several condition combinations and therefore it is not always necessary to run 2^N separate test cases to achieve 100% multiple condition coverage.

Coverage is measured as the number of exercised atomic condition combinations over all decisions in the test object, normally expressed as a percentage.

Applicability
This technique is used to test high risk software and embedded software which are expected to run reliably without crashing for long periods of time.

Limitations/Difficulties
Because the number of test cases can be derived directly from a truth table containing all the atomic conditions, this level of coverage can easily be determined. However, the sheer number of test cases required for multiple condition testing makes modified condition/decision testing more feasible for situations where there are several atomic conditions in a decision.

If the compiler uses short-circuiting, the number of condition combinations that can be exercised will often be reduced, depending on the order and grouping of logical operations that are performed on the atomic conditions.

2.6 Basis Path Testing

This chapter has been removed from version v4.0 of this syllabus.

2.7 API Testing

An application programming interface (API) is a defined interface that enables a program to call another software system, which provides it with a service, such as access to a remote resource. Typical services include web services, enterprise service buses, databases, mainframes, and Web UIs.

API Testing is a type of testing rather than a technique. 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 that 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 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 for testing systems of systems as the individual 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

Software containerization results in the division of a software program into several containers which communicate with each other using mechanisms such as those listed above. API testing should also target these interfaces.

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 making calls to the API with both realistic input values and unexpected inputs for checking exception handling. More thorough API tests may ensure that callable entities are exercised at least once, or all possible functions are called at least once.

Representational State Transfer is an architectural style. RESTful Web services allow requesting systems to access Web resources using a uniform set of stateless operations. Several coverage criteria exist for RESTful web APIs, the de-facto standard for software integration [Web-7]. They can be divided into two groups: input coverage criteria and output coverage criteria. Among others, input criteria may

require the execution of all possible API operations, the use of all possible API parameters, and the coverage of sequences of API operations. Among others, output criteria may require the generation of all correct and erroneous status codes, and the generation of responses containing resources exhibiting all properties (or all property types).

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, duplication of transactions and issues in exception handling.

2.8 Selecting a White-Box Test Technique

The selected white-box test technique is normally specified in terms of a required coverage level, which is achieved by applying the test technique. For instance, a requirement to achieve 100% statement coverage would typically lead to the use of statement testing. However black-box test techniques are normally applied first, coverage is then measured, and the white-box test technique only used if the required white-box coverage level has not been achieved. In some situations, white-box testing may be used less formally to provide an indication of where coverage may need to be increased (e.g., creating additional tests where white-box coverage levels are particularly low). Statement testing would normally be sufficient for such informal coverage measurement.

When specifying a required white-box coverage level it is good practice to only specify it at 100%. The reason for this is that if lower levels of coverage are required, then this typically means that the parts of the code that are not exercised by testing are the parts that are the most difficult to test, and these parts are normally also the most complex and error-prone. So, by requesting and achieving for example 80% coverage, it may mean that the code that includes the majority of the detectable defects is left untested. For this reason, when white-box coverage criteria are specified in standards, they are nearly always specified at 100%. Strict definitions of coverage levels have sometimes made this level of coverage impracticable. However, those given in ISO 29119-4 allow infeasible coverage items to be discounted from the calculations, thus making 100% coverage an achievable goal.

When specifying the required white-box coverage for a test object, it is also only necessary to specify this for a single coverage criterion (e.g., it is not necessary to require both 100% statement coverage and 100% MC/DC). With exit criteria at 100% it is possible to relate some exit criteria in a subsumes hierarchy, where coverage criteria are shown to subsume other coverage criteria. One coverage criterion is said to subsume another if, for all components and their specifications, every set of test cases that satisfies the first criterion also satisfies the second. For example, branch coverage subsumes statement coverage because if branch coverage is achieved (to 100%), then 100% statement coverage is guaranteed. For the white-box test techniques covered in this syllabus, we can say that branch and decision coverage subsume statement coverage, MC/DC subsumes decision and branch coverage, and multiple condition coverage subsumes MC/DC (if we consider branch and decision coverage to be the same at 100%, then we can say that they subsume each other).

Note that when determining the white-box coverage levels to be achieved for a system, it is quite normal to define different levels for different parts of the system. This is because different parts of a system contribute differently to risk. For instance, in an avionics system, the subsystems associated with in-flight entertainment would be assigned a lower level of risk than those associated with flight control. Testing interfaces is common for all types of systems and is normally required for all integrity levels for safety-related systems (see section 2.8.2 for more on integrity levels). The level of required coverage for API testing will normally increase based on the associated risk (e.g., the higher level of risk associated with a public interface may require more rigorous API testing).

The selection of which white-box test technique to use is generally based on the nature of the test object and its perceived risks. If the test object is considered to be safety-related (i.e. failure could cause harm

to people or the environment) then regulatory standards are applicable and will define required white-box coverage levels (see section 2.8.2). If the test object is not safety-related then the choice of which white-box coverage levels to achieve is more subjective, but should still be largely based on perceived risks, as described in section 2.8.1.

2.8.1 Non-Safety-Related Systems

The following factors (in no particular order of priority) are typically considered when selecting white-box test techniques for non-safety-related systems:

- Contract – If the contract requires a particular level of coverage to be achieved, then not achieving this coverage level will potentially result in a breach of contract.
- Customer – If the customer requests a particular level of coverage, for instance as part of the test planning, then not achieving this coverage level may cause problems with the customer.
- Regulatory standard – For some industry sectors (e.g., financial) a regulatory standard that defines required white-box coverage criteria applies for mission-critical systems. See section 2.8.2 for coverage of regulatory standards for safety-related systems.
- Test strategy – If the organization's test strategy specifies requirements for white-box code coverage, then not aligning with the organizational strategy may risk censure from higher management.
- Coding style – If the code is written with no multiple conditions within decisions, then it would be wasteful to require white-box coverage levels such as MC/DC and multiple condition coverage.
- Historical defect information – If historical data on the effectiveness of achieving a particular coverage level suggests that it would be appropriate to use for this test object, it would be risky to ignore the available data. Note that such data may be available within the project, organization or industry.
- Skills and experience – If the testers available to perform the testing are not sufficiently experienced and skilled in a particular white-box technique, it may be misunderstood and may introduce unnecessary risk if that technique was selected.
- Tools – White-box coverage can only be measured in practice by using coverage tools. If such tools are not available that support a given coverage measure, then selecting that measure to be achieved would introduce a high risk level.

When selecting white-box testing for non-safety-related systems, the Technical Test Analyst has more freedom to recommend the appropriate white-box coverage for non-safety related systems than for safety related systems. Such choices are typically a compromise between the perceived risks and the cost, resources and time required to treat these risks through white-box testing. In some situations, other treatments, which might be implemented by other software testing approaches or otherwise (e.g., different development approaches) may be more appropriate.

2.8.2 Safety-related systems

Where the software being tested is part of a safety-related system, then a regulatory standard which defines the required coverage levels to be achieved will normally have to be used. Such standards typically require a hazard analysis to be performed for the system and the resultant risks are used to assign integrity levels to different parts of the system. Levels of required coverage are defined for each of the integrity levels.

IEC 61508 (functional safety of programmable, electronic, safety-related systems [IEC 61508]) is an umbrella international standard used for such purposes. In theory, it could be used for any safety-related system, however some industries have created specific variants (e.g., ISO 26262 [ISO 26262] applies to automotive systems) and some industries have created their own standards (e.g., DO-178C [DO-178C] for airborne software). Additional information regarding ISO 26262 is provided in the ISTQB® Automotive Software Tester syllabus [CT_AuT_SYL].

IEC 61508 defines four safety integrity levels (SILs), each of which is defined as a relative level of risk-reduction provided by a safety function, correlated to the frequency and severity of perceived hazards. When a test object performs a function related to safety, the higher the risk of failure means that the test object should have higher reliability. The following table shows the reliability levels associated with the SILs. Note that the reliability level for SIL 4 for the continuous operation case is extremely high, since it corresponds to a mean time between failures (MTBF) greater than 10,000 years.

IEC 61508 SIL	Continuous Operation (probability of a dangerous failure per hour)	On-Demand (probability of failure on demand)
1	$\geq 10^{-6}$ to $<10^{-5}$	$\geq 10^{-2}$ to $<10^{-1}$
2	$\geq 10^{-7}$ to $<10^{-6}$	$\geq 10^{-3}$ to $<10^{-2}$
3	$\geq 10^{-8}$ to $<10^{-7}$	$\geq 10^{-4}$ to $<10^{-3}$
4	$\geq 10^{-9}$ to $<10^{-8}$	$\geq 10^{-5}$ to $<10^{-4}$

The recommendations for white-box coverage levels associated with each SIL are shown in the following table. Where an entry is shown as 'Highly recommended', in practice achieving that level of coverage is normally considered mandatory. In contrast, where an entry is only shown as 'Recommended', many practitioners consider this to be optional and avoid achieving it by providing a suitable rationale. Thus, a test object assigned to SIL 3 is normally tested to achieve 100% branch coverage (it achieves 100% statement coverage automatically, as shown by the subsumes ordering).

IEC 61508 SIL	100% Statement Coverage	100% Branch Coverage	100% MC/DC
1	Recommended	Recommended	Recommended
2	Highly recommended	Recommended	Recommended
3	Highly recommended	Highly recommended	Recommended
4	Highly recommended	Highly recommended	Highly recommended

Note that the above SILs and requirements on coverage levels from IEC 61508 are different in ISO 26262, and different again in DO-178C.

3. Static and Dynamic Analysis - 180 mins.

Keywords

anomaly, control flow analysis, cyclomatic complexity, data flow analysis, definition-use pair, dynamic analysis, memory leak, static analysis, wild pointer

Learning Objectives for Static and Dynamic Analysis

3.2 Static Analysis

TTA-3.2.1 (K3) Use control flow analysis to detect if code has any control flow anomalies and to measure cyclomatic complexity
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

Note: TTA-3.2.4 has been removed from version v4.0 of this syllabus.

3.3 Dynamic Analysis

TTA-3.3.1 (K3) Apply dynamic analysis to achieve a specified goal

3.1 Introduction

Static analysis (see section 3.2) is a form of testing that is performed without executing the software. The quality of the software is evaluated by a tool or by a person based on its form, structure, content, or documentation. This static view of the software allows detailed analysis without having to create the data and preconditions necessary to execute test cases.

Apart from static analysis, static testing techniques also include different forms of review. The ones that are relevant for the Technical Test Analyst are covered in Chapter 5.

Dynamic analysis (see section 3.3) requires the actual execution of the code and is used to find defects 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 a person monitoring the executing system watching for such indicators as a rapid increase of memory use.

3.2 Static Analysis

The objective of static analysis is to detect actual or potential defects in code and system architecture and to improve their maintainability.

3.2.1 Control Flow Analysis

Control flow analysis is the static technique where the steps followed through a program are analyzed through the use of a control flow graph, usually with the use of 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 or that do not terminate), ambiguous targets of function calls in certain languages, incorrect sequencing of operations, code that cannot be reached, uncalled functions, etc.

Control flow analysis can be used to determine cyclomatic complexity. The cyclomatic complexity is a positive integer which represents the number of independent paths in a strongly connected graph.

The cyclomatic complexity is generally used as an indicator of the complexity of a component. Thomas McCabe's theory [McCabe76] was that the more complex the system, the harder it would be to maintain and the more defects it would contain. Many studies have noted this correlation between complexity and the number of contained defects. Any component that is measured with a higher complexity should be reviewed for possible refactoring, for example division into multiple components.

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. The lifecycle of each variable along a control flow path is investigated, (i.e., where it is declared, defined, used, and destroyed), since potential anomalies can be identified if these actions are used out of sequence [Beizer90].

One common technique classifies the use of a variable as one of three atomic actions:

- when the variable is defined, declared, or initialized (e.g., x:=3)
- when the variable is used or read (e.g., if x > temp)
- when the variable is killed, destroyed, or goes out of scope (e.g., text_file_1.close, loop control variable (i) on exit from loop)

Sequences of such actions that indicate potential anomalies include:

- definition followed by another definition or kill with no intervening use
- definition with no subsequent kill (e.g., leading to a possible memory leak for dynamically allocated variables)

- use or kill before definition
- use or kill after a kill

Depending on the programming language, some of these anomalies may be identified by the compiler, but a separate static analysis tool might be needed to identify the data flow anomalies. For instance, re-definition with no intervening use is allowed in most programming languages, and may be deliberately programmed, but it would be flagged by a data flow analysis tool as being a possible anomaly that should be checked.

The use of control flow paths to determine the sequence of actions for a variable can lead to the reporting of potential anomalies that cannot occur in practice. For instance, static analysis tools cannot always identify if a control flow path is feasible, as some paths are only determined based on values assigned to variables at run time. There is also a class of data flow analysis problems that are difficult for tools to identify, when the analyzed data are part of data structures with dynamically assigned variables, such as records and arrays. Static analysis tools also struggle with identifying potential data flow anomalies when variables are shared between concurrent threads of control in a program as the sequence of actions on data becomes difficult to predict.

In contrast to data flow analysis, which is static testing, data flow testing is dynamic testing in which test cases are generated to exercise ‘definition-use pairs’ in program code. Data flow testing uses some of the same concepts as data flow analysis as these definition-use pairs are control flow paths between a definition and a subsequent use of a variable in a program.

3.2.3 Using Static Analysis for Improving Maintainability

Static analysis can be applied in several ways to improve the maintainability of code, architecture and websites.

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 feature is likely to result in further defects being introduced.

Static analysis is used to verify compliance with coding standards and guidelines; where non-compliant code is identified, it can be updated to improve its maintainability. These standards and guidelines describe required coding and design practices such as conventions for naming, commenting, indentation and modularization. Note that static analysis tools generally raise warnings rather than detect defects. These warnings (e.g., on level of complexity) may be provided 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 components (although the runtime overhead imposed by component 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 has good maintainability is more likely to have a low measure of coupling (the degree to which components rely on each other during execution) and a high measure of cohesion (the degree to which a component 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.

The maintenance of a website 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

In addition to evaluating maintainability, static analysis tools can also be applied to the code used for implementing websites to check for possible exposure to security vulnerabilities such as code injection, cookie security, cross-site scripting, resource tampering, and SQL code injection. Further details are provided in section 4.3 and in the Security Testing syllabus [CT_SEC_SYL].

3.3 Dynamic Analysis

3.3.1 Overview

Dynamic analysis is used to detect failures where the symptoms are only visible when the code is executed. 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 (intermittent) 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 or resource 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 start dynamic analysis early in the project.

Dynamic analysis may be applied to accomplish the following:

- Prevent failures from occurring by detecting memory leaks (see section 3.3.2) and wild pointers (see section 3.3.3);
- Analyze system failures which cannot easily be reproduced;
- Evaluate network behavior;
- Improve system performance by using code profilers to provide information on runtime system behavior which can be used to make informed changes.

Dynamic analysis may be performed at any test level and requires technical and system skills to do the following:

- Specify the test objectives of dynamic analysis;
- Determine the proper time to start and stop the analysis;
- Analyze the results.

Dynamic analysis tools can be used even if the Technical Test Analyst has minimal technical skills; the tools used usually create comprehensive logs which can be analyzed by those with the needed technical and analytical skills.

3.3.2 Detecting Memory Leaks

A memory leak occurs when areas of memory (RAM) are allocated to a program but are not subsequently released when no longer needed. This memory area 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 to avoid a memory leak. Many modern programming environments include automatic or semi-automatic “garbage collection” where allocated

memory is freed after use without the programmer's direct intervention. Isolating memory leaks can be very difficult in cases where allocated memory should be freed by the automatic garbage collectors.

Memory leaks typically cause problems after some time – when a significant amount of memory has leaked and become unavailable. When software is newly installed, or when the system is restarted, memory is reallocated, and so memory leaks are not noticeable; testing is an example where frequent memory allocation can prevent the detection of memory leaks. For these reasons, the negative effects of memory leaks may first be noticed when the program is in production.

The primary symptom of a memory leak is a steadily worsening 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 for some systems.

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.

3.3.3 Detecting Wild Pointers

Wild pointers within a program are pointers that are no longer accurate and 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 including the following:

- 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 (this may also represent a security threat).

Note that changes 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. 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 runtime. 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 Efficiency

Dynamic analysis is not just useful for detecting failures and locating the associated defects. With the dynamic analysis of program performance, tools help identify performance efficiency 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 component is called during execution. Components which are frequently called would be likely candidates for performance enhancement. Often the Pareto rule holds here: a program spends a disproportionate part (80%) of its run time in a small number (20%) of components [Andrist20].

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. Further details are provided in the Performance Testing syllabus [CT_PT_SYL].

4. Quality Characteristics for Technical Testing - 345 mins.

Keywords
accountability, adaptability, analyzability, authenticity, availability, capacity, coexistence, compatibility, confidentiality, fault tolerance, installability, integrity, maintainability, maturity, modifiability, modularity, non-repudiation, operational profile, performance efficiency, portability, quality characteristic, recoverability, reliability, reliability growth model, replaceability, resource utilization, reusability, security, testability, time behavior

Learning Objectives for Quality Characteristics for Technical Testing

4.2 General Planning Issues

TTA-4.2.1 (K4) For a particular scenario, analyze the non-functional requirements and write the respective sections of the test plan
TTA-4.2.2 (K3) Given a particular product risk, define the particular non-functional test type(s) which are most appropriate
TTA-4.2.3 (K2) Understand and explain the stages in an application's software development lifecycle where non-functional testing should typically be applied
TTA-4.2.4 (K3) For a given scenario, define the types of defects you would expect to find by using the different non-functional test types

4.3 Security Testing

TTA-4.3.1 (K2) Explain the reasons for including security testing in a test approach
TTA-4.3.2 (K2) Explain the principal aspects to be considered in planning and specifying security tests

4.4 Reliability Testing

TTA-4.4.1 (K2) Explain the reasons for including reliability testing in a test approach
TTA-4.4.2 (K2) Explain the principal aspects to be considered in planning and specifying reliability tests

4.5 Performance Testing

TTA-4.5.1 (K2) Explain the reasons for including performance testing in a test approach
TTA-4.5.2 (K2) Explain the principal aspects to be considered in planning and specifying performance testing

4.6 Maintainability Testing

TTA-4.6.1 (K2) Explain the reasons for including maintainability testing in a test approach

4.7 Portability Testing

TTA-4.7.1 (K2) Explain the reasons for including portability testing in a test approach

4.8 Compatibility Testing

TTA-4.8.1 (K2) Explain the reasons for including coexistence testing in a test approach

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 efficiency benchmarking and testing resource usage is important. During operational acceptance testing and system testing, 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 performed according to the risk priorities and the available resources.

Both dynamic testing and static testing, including reviews (see Chapters 2, 3 and 5) may be applied to test the non-functional quality characteristics described in this chapter.

The description of product quality characteristics provided in ISO 25010 [ISO 25010] is used as a guide to the characteristics and their 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 the Technical Test Analyst syllabi.

Characteristic	Sub-Characteristics	Test Analyst	Technical Test Analyst
Functional suitability	Functional correctness, functional appropriateness, functional completeness	X	
Reliability	Maturity, fault tolerance, recoverability, availability		X
Usability	Appropriateness recognizability, learnability, operability, user interface aesthetics, user error protection, accessibility	X	
Performance efficiency	Time behavior, resource utilization, capacity		X
Maintainability	Analyzability, modifiability, testability, modularity, reusability		X
Portability	Adaptability, installability, replaceability	X	X
Security	Confidentiality, integrity, non-repudiation, accountability, authenticity		X
Compatibility	Coexistence		X
	Interoperability	X	

Note that a table is provided in Appendix A which compares the characteristics described in the now cancelled ISO 9126-1 standard (as used in version 2012 of this syllabus) with those in the later ISO 25010 standard.

For all the quality characteristics and sub-characteristics discussed in this section, the typical risks must be recognized so that an appropriate testing approach can be formed and documented. Quality characteristic testing requires particular attention to lifecycle timing, required tools, required standards, software and documentation availability and technical expertise. Without planning an approach 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.

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.

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 (SLAs) 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 performance testing 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 testing can put the success of a project 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. This information may be used in creating the master test plan.

The following general factors are considered when performing these tasks:

- Stakeholder requirements
- Test environment requirements
- Required tool acquisition and training
- 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 efficiency, 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, product owners, users, operations staff, and maintenance staff. If key stakeholders are excluded some requirements are likely to be missed. For more details about capturing requirements, refer to the Advanced Test Manager syllabus [CTAL_TM_SYL].

In agile software development non-functional requirements may be stated as user stories or added to the functionality specified in use cases as non-functional constraints.

4.2.2 Test Environment Requirements

Many technical tests (e.g., security tests, reliability tests, performance efficiency tests) require a production-like test environment 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 options may be considered:

- Using the production environment

- Using a scaled-down version of the system, taking care that the test results obtained are sufficiently representative of the production system
- Using cloud-based resources as an alternative to acquiring the resources directly
- Using virtualized environments

Test execution times must be planned carefully, and it is quite likely that some tests can only be executed at specific times (e.g., at low usage times).

4.2.3 Required Tool Acquisition and Training

Tools are part of the test environment. Commercial tools or simulators are particularly relevant for performance efficiency 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 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.

4.2.4 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 several 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 a particular time of day, 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 and Data Protection

Specific security measures implemented for a system should be considered at the test planning stage to ensure that all test 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 (e.g., personal data, credit card data). Making test data anonymous is a non-trivial task which must be planned as part of test implementation.

4.3 Security Testing

4.3.1 Reasons for Considering Security 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 component, integration and system testing levels. Due to the changing nature of security threats, security tests may also be scheduled regularly after the system has entered production. This is particularly true for dynamic open architectures such as the Internet of Things (IoT) where the production phase is characterized by many updates to the software and hardware elements used.
- The test approaches proposed by the Technical Test Analyst may include reviews of the architecture, design, and code, and the static analysis of code with security tools. These can be effective in finding security issues that are easily missed during dynamic testing.
- The Technical Test Analyst may be called upon to design and perform certain security “attacks” (see below) which require careful planning and coordination with stakeholders (including security testing specialists). Other security tests may be performed in cooperation with developers or with Test Analysts (e.g., testing user rights, access, and privileges).
- An essential aspect of security test planning is obtaining approvals. For the Technical Test Analyst, this means ensuring that explicit permission has been obtained 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.
- All security test planning should be coordinated with an organization’s Information Security Officer if the organization has such a role.
- It should be noted that improvements which may be made to the security of a system may affect its performance efficiency or reliability. After making security improvements it is advisable to consider the need for conducting performance efficiency or reliability tests (see sections 4.5 and 4.4).

Individual standards may apply when conducting security test planning, such as [IEC 62443-3-2], which applies to industrial automation and control systems.

The Security Testing syllabus [CT_SEC_SYL] includes further details of key security test plan elements.

4.3.3 Security Test Specification

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

- 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 ISO 25010 sub-characteristics of security [ISO 25010] also provide a basis from which security tests may be specified. These focus on the following aspects of security:

- Confidentiality
- Integrity
- Non-repudiation
- Accountability
- Authenticity

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 information and checklists such as those provided by the National Institute of Standards and Technology (NIST) [Web-1] and the Open Web Application Security Project™ (OWASP) [Web-4].
- 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 defects. The various “attacks” described in [Whittaker04] are a valuable source of techniques developed specifically for security testing.

Note that attack plans may be developed for penetration testing.

Section 3.2 (static analysis) and the Security Testing syllabus [CT_SEC_SYL] include further details of security testing.

4.4 Reliability Testing

4.4.1 Introduction

The ISO 25010 classification of product quality characteristics defines the following sub-characteristics of reliability: maturity, availability, fault tolerance, and recoverability. Testing for reliability is concerned with the ability of a system or software to perform specified functions under specified conditions for a specified period of time.

4.4.2 Testing for Maturity

Maturity is the degree to which the system (or software) meets reliability requirements under normal operation conditions, which are typically specified by using an operational profile (see section 4.9). Maturity measures, when used, often form one of the release criteria for a system.

Traditionally, maturity has been specified and measured for high-reliability systems, such as those associated with safety-critical functions (e.g., an aircraft flight control system), where the maturity goal is defined as part of a regulatory standard. A maturity requirement for such a high-reliability system may be a mean time between failures (MTBF) of up to 10^9 hours (although this is practically impossible to measure).

The usual approach to testing maturity for high-reliability systems is known as reliability growth modelling, and it normally takes place at the end of system testing, after the testing for other quality characteristics has been completed and any defects associated with detected failures have been fixed. It is a statistical approach usually performed in a test environment as close to the operational environment as possible. To measure a specified MTBF, test inputs are generated based on the operational profile and the system is run and failures are recorded (and subsequently fixed). A reducing frequency of failure allows the MTBF to be predicted using a reliability growth model.

Where maturity is used as a goal for lower reliability systems (e.g., non-safety-related), then the number of failures observed during a defined period of expected operational use (e.g., no more than 2 high-impact failures per week) may be used and may be recorded as part of the service level agreement (SLA) for the system.

4.4.3 Testing for Availability

Availability is typically specified in terms of the amount of time a system (or software) is available to users and other systems under normal operating conditions. Systems may have a low maturity, but still have a high availability. For instance, a phone network may fail to connect several calls (and thus have low maturity), but as long as the system recovers quickly and allows the next attempts to connect most users will be content. However, a single failure that caused a phone network outage for several hours would represent an unacceptable level of availability. Availability is often specified as part of an SLA and measured for operational systems, such as websites and software as a service (SaaS) applications. The availability of a system may be described as 99.999% ('five nines'), in which case it should be unavailable no more than 5 minutes per year, alternatively system availability may be specified in terms of unavailability (e.g., the system shall not be down for more than 60 minutes per month).

Measuring availability prior to operation (e.g., as part of making the release decision) is often performed using the same tests used for measuring maturity; tests are based on an operational profile of expected use over a prolonged period and performed in a test environment as close to the operational environment as possible. Availability can be measured as MTTF/(MTTF + MTTR), where MTTF is the mean time to failure and MTTR is the mean time to repair (MTTR), which is often measured as part of maintainability testing. Where a system is high-reliability and incorporates recoverability (see section 4.4.5) then we can substitute mean time to recover for MTTR in the equation when the system takes some time to recover from a failure.

4.4.4 Testing for Fault Tolerance

Systems (or software) with extremely high reliability requirements often incorporate a fault tolerant design, ideally allowing the system to continue operating with no noticeable downtime when failures occur. The main measure of fault tolerance for a system is the ability of the system to tolerate failures. Testing for fault tolerance, therefore, involves simulating failures to determine if the system can continue operating when such a failure occurs. Identifying potential failure conditions to be tested is an important part of testing for fault tolerance.

A fault tolerant design will typically involve one or more duplicate subsystems, so providing a level of redundancy in case of failure. In the case of software, such duplicate systems need to be developed independently, to avoid common mode failures; this approach is known as N-version programming. Aircraft flight control systems may include three or four levels of redundancy, with the most critical functions implemented in several variants. Where hardware reliability is a concern, an embedded system may run on multiple dissimilar processors, while a critical website may run with a mirror (failover) server

performing the same functions that is always available to take over in the event of the primary server failing. Whichever approach to fault tolerance is implemented, testing it typically requires both the detection of the failure and the subsequent response to the failure to be tested.

Fault injection testing tests the robustness of a system in the presence of defects in the system's environment (e.g., a faulty power supply, badly-formed input messages, a process or service not available, a file not found, or memory not available) and defects in the system itself (e.g., a flipped bit caused by cosmic radiation, a poor design, or bad coding). Fault injection testing is a form of negative testing - we deliberately inject defects into the system to assure ourselves that it reacts in the way we expected (i.e., safely for a safety-related system). Sometimes the defect scenarios we test should never occur (e.g., a software task should never 'die' or get stuck in an infinite loop) and cannot be simulated by traditional system testing, but with fault injection testing, we create the defect scenario and measure the subsequent system behavior to ensure it detects and handles the failure.

4.4.5 Testing for Recoverability

Recoverability is a measure of a system's (or software's) ability to recover from a failure, either in terms of the time required to recover (which may be to a diminished state of operation) or the amount of data lost. Approaches to recoverability testing include failover testing and backup and restore testing; both typically include testing procedures based on dry runs and only occasional, and ideally, unannounced, hands-on tests in operational environments.

Backup and restore testing focuses on testing the procedures in place to minimize the effects of a failure on the system data. Tests evaluate the procedures for both the backup and restoration of data. While testing for data backup is relatively easy, testing for restoring a system from backed up data can be more complex and often require careful planning to ensure disruption to the operational system is minimized. Measures include the time taken to perform different types of backup (e.g., full and incremental), the time taken to restore the data (the recovery time objective), and the level of data loss that is acceptable (recovery point objective).

Failover testing is performed when the system architecture comprises both a primary and a failover system that will take over if the primary system fails. Where a system must be able to recover from a catastrophic failure (e.g., a flood, terrorist attack, or a serious ransomware attack) then failover testing is often known as disaster recovery testing and the failover system (or systems) may often be in another geographical location. Performing a full disaster recovery test on an operational system needs extremely careful planning due to the risks and disruption (often to senior managers' free time, which is likely to be taken up in managing the recovery). If a full disaster recovery test fails then we would immediately revert to the primary system (as it hasn't really been destroyed!). Failover tests include testing the move to the failover system, once it has taken over, provides the required service level.

4.4.6 Reliability Test Planning

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

- Timing – Reliability testing typically requires the complete system to be tested and other test types to already be completed – and can take a long time to perform.
- Costs – High-reliability systems are notoriously expensive to test due to the long periods for which systems must be tested without failure to be able to predict a required high MTBF.
- Duration – Maturity testing using reliability growth models is based on detected failures and for high reliability levels, it will take a long time to get statistically significant results.
- Test environment – The test environment needs to be as similar to operational as possible, or the operational environment may be used. However, if using the operational environment, this can be disruptive to users and can be high risk if, for instance, a disaster recovery test adversely affects the operational system.
- Scope – Different subsystems and components may be tested for different types and levels of reliability.

- Exit criteria – Reliability requirements should be set by regulatory standards for safety-related applications.
- Failure – Reliability measures are very dependent on counting failures and so there must be up-front agreement on what constitutes a failure.
- Developers – For maturity testing using reliability growth models agreement must be reached with developers that identified defects are fixed as soon as possible.
- Measuring operational reliability is relatively simple compared to measuring reliability prior to release as we have to only measure failures; this this may need liaison with operations staff.
- Early testing - Achieving high reliability (as opposed to measuring reliability) requires testing to start as early as possible, with rigorous reviews of early baseline documents and the static analysis of code.

4.4.7 Reliability Test Specification

For testing maturity and availability, the testing is largely based on testing the system under normal operating conditions. For such tests, an operational profile that defines how the system is expected to be used is required. See section 4.9 for more detail on operational profiles.

For testing fault tolerance and recoverability, it is often necessary to generate tests that replicate failures in the environment and the system itself, to determine how the system reacts. Fault injection testing is often used for this. Various techniques and checklists are available for the identification of possible defects and corresponding failures (e.g., Fault Tree Analysis, Failure Mode and Effect Analysis).

4.5 Performance Testing

4.5.1 Introduction

The ISO 25010 classification of product quality characteristics defines the following sub-characteristics of performance efficiency: time behavior, resource utilization, and capacity. Performance testing (associated with the performance efficiency quality characteristic) is concerned with measuring the performance of a system or software under specified conditions relative to the amount of resources used. Typical resources include elapsed time, CPU time, memory, and bandwidth.

4.5.2 Testing for Time Behavior

Testing for time behavior measures the following aspects of a system (or software) under specified operating conditions:

- elapsed time from receiving a request until the first response (i.e. the time to start responding, not the time to complete the requested activity), also called response time;
- turnaround time from starting an activity until the activity is completed, also called processing time;
- number of activities completed per unit of time (e.g., number of database operations per second), also called throughput rate.

For many systems, maximum response times for different system functions are specified as requirements. In such cases the response time is the elapsed time plus the turnaround time. When a system has to perform a number of steps (e.g., a pipeline) to complete an activity, it may be useful to measure the time taken for each step and analyze the results to determine if one or more steps are causing a bottleneck.

4.5.3 Testing for Resource Utilization

Testing for resource utilization measures the following aspects of a system (or software) under specified operating conditions:

- CPU utilization, normally as a percentage of available CPU time;

- memory utilization, normally as a percentage of available memory;
- i/o device utilization, normally as a percentage of available i/o device time;
- bandwidth utilization, normally as a percentage of available bandwidth.

4.5.4 Testing for Capacity

Testing for capacity measures the maximum limits for the following aspects of a system (or software) under specified operating conditions:

- transactions processed per unit of time (e.g., maximum of 687 words translated per minute);
- users simultaneously accessing the system (e.g., maximum of 1223 users);
- new users added to have system access per unit of time (e.g., maximum of 400 users added per second).

4.5.5 Common Aspects of Performance Testing

When testing for time behavior, resource utilization, or capacity it is normal for several measurements to be taken and the mean used as the reported measure; this is because the time values measured can fluctuate depending on other background tasks the system may be performing. In some situations, the measurements will be handled in a more meticulous way (e.g. using variance or other statistical measures), or outliers may be investigated and discarded, if appropriate.

Dynamic analysis (see section 3.3.4) may be used to identify components causing a bottleneck, measure the resources used for resource utilization testing, and measure the maximum limits for capacity testing.

4.5.6 Types of Performance Testing

Performance testing differs from most other forms of testing in that there can be two distinct objectives. The first is to determine if the software under test meets the specified acceptance criteria. For instance, determining whether the system displays a requested web page within the maximum specified 4 seconds. The second objective is to provide information to the system developers to help them improve the efficiency of the system. For instance, detecting bottlenecks and identifying which parts of the system architecture are adversely affected when unexpectedly high numbers of users access the system simultaneously.

The performance testing described in sections 4.5.2, 4.5.3 and 4.5.4 can all be used to determine if the software under test meets the specified acceptance criteria. They are also used to measure baseline values that are used for later comparison when the system is changed. The following performance test types are more often used to provide information to developers on how the system responds under different operational conditions.

4.5.6.1 Load Testing

Load testing focuses on the ability of a system to handle different loads. These loads are typically defined in terms of the number of users accessing the system simultaneously or the number of concurrent processes running and can be defined as operational profiles (see section 4.9 for more details on operational profiles). The handling of these loads is typically measured in terms of the system's time behavior and resource utilization (e.g., determining the effect on the response time of doubling the number of users). When performing load testing, it is normal practice to start with a low load and gradually increase the load while measuring the system's time behavior and resource utilization. Typical information from load testing that could be useful to developers would include unexpected changes in response times or the use of system resources when the system handled a particular load.

4.5.6.2 Stress Testing

There are two types of stress testing; the first is similar to load testing and the second is a form of robustness testing.

In the first, load tests are normally performed with the load initially set to the maximum expected and then increased until the system fails (e.g., response times become unreasonably long or the system crashes). Sometimes, instead of forcing the system to fail, a high load is used to stress the system and then the load is reduced to a normal level and the system is checked to ensure its performance levels have recovered to their pre-stress levels.

In the second, performance tests are run with the system deliberately compromised by reducing its access to expected resources (e.g., reducing available memory or bandwidth). The results of stress testing can provide developers with an insight into which aspects of the system are most critical (i.e. weak links) and so may need upgrading.

4.5.6.3 Scalability Testing

A scalable system can adapt to different loads. For instance, a scalable website could scale to use more back-end servers as demand increases and use fewer when demand decreases. Scalability testing is similar to load testing but tests the ability of a system to scale up and down when faced with changing loads (e.g., more users than the current hardware can handle).

The Performance Testing syllabus [CT_PT_SYL] includes further performance test types.

4.5.7 Performance Test Planning

In general, the following are of particular relevance when planning performance efficiency tests:

- Timing – Performance tests often require the entire system to be implemented and run on a representative test environment, which means it is typically performed as part of system testing.
- Reviews - Code reviews, in particular those which focus on database interaction, component interaction and error handling, can identify performance efficiency issues (particularly regarding “wait and retry” logic and inefficient queries) and should be scheduled to take place as soon as code is available (i.e. prior to dynamic testing).
- Early testing – Some performance tests (e.g., determining CPU utilization for a critical component) may be scheduled as part of component testing. Components identified as being a bottleneck by performance testing may be updated and re-tested in isolation as part of component testing.
- Architecture changes - Adverse results from performance
- testing can sometimes result in a change to the system architecture. Where such major changes to the system could be suggested by performance test results, performance testing should start as early as possible, to maximize the time available to address such issues.
- Costs – Tools and test environments can be expensive, which means that temporary cloud-based test environments may be rented, and “top-up” tool licenses may be used. In such cases, test planning typically needs to optimize the time spent running tests to minimize costs.
- Test environment - The test environment needs to be as representative of the operational environment as possible, otherwise the challenge of scaling the performance test results from the test environment to the expected operational environment is increased.
- Exit criteria – Performance efficiency requirements can sometimes be difficult to obtain from the customer and so are often derived from baselines from previous or similar systems. In the case of embedded safety-related systems, some requirements, such as the maximum amount of CPU and memory used, may be specified by regulatory standards.
- Tools – Load generation tools are often required to support performance testing. For instance, verifying the scalability of a popular website may require the simulation of hundreds of thousands of virtual users. Tools that simulate resource restrictions are also especially useful

for stress testing. Care should be taken to ensure that any tools acquired to support the testing are compatible with the communications protocols used by the system under test.

The Performance Testing syllabus [CT_PT_SYL] includes further details of performance test planning.

4.5.8 Performance Test Specification

Performance testing is largely based on testing the system under specified operating conditions. For such tests, an operational profile that defines how the system is expected to be used is required. See section 4.9 for more details on operational profiles.

For performance testing, it is often necessary to change the load on the system by modifying parts of the operational profile to simulate a change from the expected operational use of the system. For instance, in the case of capacity testing, it will normally be necessary to override the operational profile in the area of the variable being capacity tested (e.g., increase the number of users accessing the system until the system stops responding to determine the user access capacity). Similarly, the volume of transactions may be gradually increased when load testing.

The Performance Testing syllabus [CTSL_PT_SYL] includes further details of performance efficiency test design.

4.6 Maintainability Testing

Software often spends substantially more of its lifetime being maintained than being developed. 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, tested, modularized, and reused. Maintainability testing should not be confused with maintenance testing, which is performed to test the changes made to operational software.

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

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

Maintainability tests should be included in a 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 over the software development 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

4.6.1 Static and Dynamic Maintainability Testing

Appropriate techniques for static 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 documentation is available and should continue throughout the code implementation effort. Since maintainability is built into the code and the documentation for each code component, maintainability can be evaluated early in the software development 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). 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.

4.6.2 Maintainability Sub-characteristics

The maintainability of a system can be measured in terms of:

- Analyzability
- Modifiability
- Testability

Factors which influence these characteristics include the application of good programming practices (e.g., commenting, naming of variables, indentation), and the availability of technical documentation (e.g., system design specifications, interface specifications).

Other relevant quality sub-characteristics for maintainability [ISO 25010] are:

- Modularity
- Reusability

Modularity can be tested by means of static analysis (see section 3.2.3). Reusability testing may take the form of architectural reviews (see Chapter 5).

4.7 Portability Testing

4.7.1 Introduction

In general, portability testing relates to the degree to which a software component or system can be transferred into its intended environment (either initially or from an existing environment), can be adapted to a new environment, or can replace another entity.

ISO 25010 [ISO 25010] includes the following sub-characteristics of portability:

- Installability
- Adaptability
- 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 then 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).

4.7.2 Installability Testing

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

Typical installability testing objectives include the following:

- Validating that the software can be 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 identify invalid hardware platforms or operating system configurations
- Measuring whether the installation process can be completed within a specified number of minutes or within a specified number of steps
- Validating that the software can be downgraded or uninstalled

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

4.7.3 Adaptability Testing

Adaptability testing checks whether a given application can function correctly in all intended target environments (hardware, software, middleware, operating system, etc.). Specifying tests for adaptability requires that 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.

4.7.4 Replaceability Testing

Replaceability testing focuses on the ability of a software component to replace an existing software component in a system. This may be particularly relevant for systems which use commercial off-the-shelf (COTS) software for specific system components or for IoT applications.

Replaceability tests may be executed in parallel with functional integration tests where more than one alternative component is available for integration into the complete system. Replaceability may also be evaluated by technical review or inspection at the architecture and design levels, where the emphasis is placed on the clear definition of the interfaces of the component that may be used as a replacement.

4.8 Compatibility Testing

4.8.1 Introduction

Compatibility testing considers the following aspects [ISO 25010]:

- Coexistence
- Interoperability

4.8.2 Coexistence Testing

Computer systems which are not related to each other are said to coexist when they can run in the same environment (e.g., on the same hardware) without affecting each other's behavior (e.g., resource

conflicts). Coexistence testing should be performed when new or upgraded software will be rolled out into environments which already contain installed applications.

Coexistence 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 to another environment (e.g., production) which also runs other applications.

Typical objectives of coexistence testing include:

- Evaluation of possible adverse impact on functional suitability 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

Coexistence issues should be analyzed when planning the targeted production environment, but the actual tests are normally performed after system testing has been completed.

4.9 Operational Profiles

Operational profiles are used as part of the test specification for several non-functional test types, including reliability testing and performance testing. They are especially useful when the requirement being tested includes the constraint of ‘under specified conditions’ as they can be used to define these conditions.

The operational profile defines a pattern of use for the system, typically in terms of the users of the system and the operations performed by the system. Users are typically specified in terms of how many are expected to be using the system (and at which times), and perhaps their type (e.g., primary user, secondary user). The different operations expected to be performed by the system are typically specified, with their frequency (and probability of occurrence). This information may be obtained by using monitoring tools (where the actual or a similar application is already available) or by predicting usage based on algorithms or estimates provided by the business organization.

Tools can be used to generate test inputs based on the operational profile, often using an approach that generates the test inputs pseudo-randomly. Such tools can be used to create "virtual" or simulated users in quantities that match the operational profile (e.g., for reliability and availability testing) or exceed it (e.g., for stress or capacity testing). See section 6.2.3 for more details on these tools.

5. Reviews - 165 mins.

Keywords
review, technical review

Learning Objectives for Reviews

5.1 Technical Test Analyst Tasks in Reviews

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 Technical Test Analyst Tasks in Reviews

Technical Test Analysts must be active participants in the technical review process, providing their unique views. All review participants should have formal review training to better understand their respective roles and must be committed to the benefits of a well-conducted technical review. This includes maintaining a constructive working relationship with the authors when describing and discussing review comments. For a detailed 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 in providing defect severity information.

Regardless of the type of review being performed, the Technical Test Analyst must be allowed adequate time to prepare. This includes time to review the work product, time to check cross-referenced documentation 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, verifying correctness with respect to technical aspects, 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 other items in the system.

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 checklist might concentrate on security issues or performance efficiency 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 efficiency, security)

Checklists should be customized for the organization and perhaps for the particular project. The checklists in this chapter are examples.

5.2.1 Architectural Reviews

Software architecture consists of the fundamental concepts or properties of a system, embodied in its elements, relationships, and in the principles of its design and evolution [ISO 42010].

Checklists[1] used for architecture reviews of the time behavior of websites could, for example, include verification of the proper implementation of the following items, which are quoted from [Web-2]:

- "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"

5.2.2 Code Reviews

Checklists for code reviews are necessarily low-level and are most useful when they are language-specific. The inclusion of code-level anti-patterns is helpful, particularly for less experienced software developers.

Checklists[1] used for code reviews could include the following items:

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?

[1] The exam question will provide a subset of the checklist with which to answer the question

- 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?

6. Test Tools and Automation - 180 mins.

Keywords
capture/playback, data-driven testing, emulator, fault injection, fault seeding, keyword-driven testing, model-based testing (MBT), simulator, test execution

Learning Objectives for Test Tools and Automation

6.1 Defining the Test Automation Project

TTA-6.1.1 (K2) Summarize the activities that the Technical Test Analyst performs when setting up a test automation project
TTA-6.1.2 (K2) Summarize the differences between data-driven and keyword-driven automation
TTA-6.1.3 (K2) Summarize common technical issues that cause automation projects to fail to achieve the planned return on investment
TTA-6.1.4 (K3) Construct keywords based on a given business process

6.2 Specific Test Tools

TTA-6.2.1 (K2) Summarize the purpose of tools for fault seeding and fault injection
TTA-6.2.2 (K2) Summarize the main characteristics and implementation issues for performance testing tools
TTA-6.2.3 (K2) Explain the general purpose of tools used for web-based testing
TTA-6.2.4 (K2) Explain how tools support the practice of model-based testing
TTA-6.2.5 (K2) Outline the purpose of tools used to support component testing and the build process
TTA-6.2.6 (K2) Outline the purpose of tools used to support mobile application testing

6.1 Defining the Test Automation Project

To be cost-effective, test tools (and particularly those which support test execution), must be carefully architected and designed. Implementing a test execution 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 documentation, detailed design documentation, 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 tasks that the Technical Test Analyst can perform regarding test execution automation. These include:

- Determining who will be responsible for the test execution (possibly in coordination with a Test Manager)
- Selecting the appropriate tool for the organization, timeline, skills of the team, and 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, defect management and tools used for continuous integration
- Developing adapters which may be required to create an interface between the test execution tool and the software under test
- Selecting the automation approach, i.e., keyword-driven or data-driven (see section 6.1.1)
- Working with the Test Manager to estimate the cost of the implementation, including training. In agile software development this aspect would typically be discussed and agreed in project/sprint planning meetings with the whole team.
- 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 and when the automated tests will be executed
- Determining how the automated test results will be combined with the manual test results

In projects with a strong emphasis on test automation, a Test Automation Engineer may be tasked with many of these activities (see the Test Automation Engineer syllabus [CT_TAE_SYL] for details). Certain organizational tasks may be taken on by a Test Manager according to project needs and preferences. In agile software development the assignment of these tasks to roles is typically more flexible and less formal.

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.

6.1.1 Selecting the Automation Approach

This section considers the following factors which impact the test automation approach:

- Automating through the GUI, API and CLI
- Applying a data-driven approach
- Applying a keyword-driven approach
- Handling software failures
- Considering system state

The Test Automation Engineer syllabus [CT_TAE_SYL] includes further details on selecting an automation approach.

6.1.1.1 Automating through the GUI, API and CLI

Test automation is not limited to testing through the GUI. Tools also 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 determining the most effective interface to be accessed to automate the testing. General test execution tools require the development of adapters to these interfaces. Planning shall consider the effort for the adapter development.

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 capture/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/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).

6.1.1.2 Applying a Data-driven Approach

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, in addition to the test scripts that process the supplied data, a harness and infrastructure are 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. In agile software development the business representative (e.g., Product Owner) may also be involved in defining data, in particular for acceptance tests. This division of labor allows those responsible for developing test scripts (e.g., the Technical Test Analyst) to focus on the implementation of 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.

6.1.1.3 Applying a Keyword-driven Approach

Another approach, called keyword- or action word-driven, goes a step further by also separating the action to be performed on the supplied test data from the test script [Buwalda01]. To accomplish this further separation, a high-level language is created which is descriptive rather than directly executable. Keywords can be defined for both high and low level actions. For example, business process keywords could include "Login", "CreateUser", and "DeleteUser". Such keywords describe the high-level actions that will be performed in the application domain. Lower level actions denote interaction with the software interface itself. Keywords like: “ClickButton”, “SelectFromList”, or “TraverseTree” may be used to test GUI capabilities that do not neatly fit into business process keywords. Keywords may contain parameters, for example the keyword “LogIn” could have two parameters: user_name and password.

Once the keywords and data to be used have been defined, the test automator (e.g., Technical Test Analyst or Test Automation Engineer) 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.

6.1.1.4 Handling Software Failures

In test automation design, it is important to anticipate and handle software failures. If a failure occurs, the test automator must determine what the test execution 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.

6.1.1.5 Considering System State

It is also important to consider the state of the system at the start and end of each test. 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).

6.1.2 Modeling Business Processes for Automation

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 or, in the case of agile software development, the business representative (e.g., Product Owner).

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.

Issues to consider include the following:

- The more detailed 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 keywords that collectively create a record. However, creating a high-level keyword that calls all six keywords may not be the most efficient approach overall.

- No matter how much analysis goes into the keyword language, there will often be times when new and changed 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. It may be more costly to set up initially, but it is likely to be cheaper overall if the project lasts long enough.

The Test Automation Engineer syllabus [CT_TAE_SYL] includes further details of modelling business processes for automation.

6.2 Specific Test Tools

This section contains overview information on tools that are likely to be used by a Technical Test Analyst beyond what is discussed in the Foundation Level syllabus [CTFL_SYL].

Note that detailed information about tools is provided by the following ISTQB® syllabi:

- Mobile Application Testing [CT_MAT_SYL]
- Performance Testing [CT_PT_SYL]
- Model-Based Testing [CT_MBT_SYL]
- Test Automation Engineer [CT_TAE_SYL]

6.2.1 Fault Seeding Tools

Fault seeding tools modify the code under test (possibly using predefined algorithms) to check the coverage achieved by specified tests. When applied in a systematic way this enables the quality of the tests (i.e., their ability to detect the inserted defects) to be evaluated and, where necessary, improved.

Fault seeding tools are generally used by the Technical Test Analyst but may also be used by the developer when testing newly developed code.

6.2.2 Fault Injection Tools

Fault injection tools deliberately supply incorrect inputs to the software to ensure the software can cope with the defect. The injected inputs cause negative conditions, which should cause error handling to run (and be tested). This disruption of the normal execution flow of the code also increases code coverage.

Fault injection tools are generally used by the Technical Test Analyst but may also be used by the developer when testing newly developed code.

6.2.3 Performance Testing Tools

Performance testing tools have the following main functions:

- Generating load
- Providing measurement, monitoring, visualization, and analysis of the system response to a given load, giving insights into the resource behavior of system and network components

Load generation is performed by implementing a pre-defined operational profile (see section 4.9) as a script. The script may initially be captured for a single user (possibly using a capture/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 accomplish tasks including generating 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 times 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 simple compared to the load profile and facilities provided by commercial tools. Further details of performance testing tools are provided in the Performance Testing syllabus [CT_PT_SYL].

6.2.4 Tools for Testing Websites

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 website
- HTML and XML checkers are tools which check compliance to the HTML and XML standards of the pages that are created by a website
- Performance testing tools 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 code, checking for orphaned (unlinked) files previously accessed by the website
- 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

The following are good sources of open-source web testing tools:

- The World Wide Web Consortium (W3C) [Web-3] This organization sets standards for the Internet and supplies a variety of tools to check for errors against those standards.
- The Web Hypertext Application Technology Working Group (WHATWG) [Web-5]. This organization sets HTML standards. They have a tool which performs HTML validation [Web-6].

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.

6.2.5 Tools to Support Model-Based Testing

Model-Based Testing (MBT) is a technique whereby a 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 [Utting07]) 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 also help to 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.

Further details of model-based testing tools are provided in the Model-Based Testing syllabus [CT_MBT_SYL].

6.2.6 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 component. For example, if Java was used as the programming language, JUnit might be used to automate the component 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.

Some build automation tools allow a new build to be automatically triggered when 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 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.

6.2.7 Tools to Support Mobile Application Testing

Emulators and simulators are frequently used tools to support the testing of mobile applications.

6.2.7.1 Simulators

A mobile simulator models the mobile platform’s runtime environment. Applications tested on a simulator are compiled into a dedicated version, which works in the simulator but not on a real device. Simulators are used as replacements for real devices in testing, but are typically limited to initial functional testing and simulating many virtual users in load testing. Simulators are relatively simple (compared to emulators) and can run tests faster than an emulator. However, the application tested on a simulator differs from the application that will be distributed.

6.2.7.2 Emulators

A mobile emulator models the hardware and utilizes the same runtime environment as the physical hardware. Applications compiled to be deployed and tested on an emulator could also be used by the real device.

However, an emulator cannot fully replace a device because the emulator may behave in a different manner than the mobile device it tries to mimic. In addition, some features may not be supported such as (multi)touch, and accelerometer. This is partly caused by limitations of the platform used to run the emulator.

6.2.7.3 Common Aspects

Simulators and emulators are often used to reduce the cost of test environments by replacing real devices. Simulators and emulators are useful in the early stage of development as these typically integrate with development environments and allow quick deployment, testing, and monitoring of applications. Using an emulator or simulator requires launching it, installing the necessary app on it, and then testing the app as if it were on the actual device. Each mobile operating system development environment typically comes with its own bundled emulator or simulator. Third party emulators and simulators are also available.

Usually, emulators and simulators allow the setting of various usage parameters. These settings might include network emulation at different speeds, signal strengths and packet losses, changing the orientation, generating interrupts, and GPS location data. Some of these settings can be very useful because they can be difficult or costly to replicate with real devices, such as global GPS positions or signal strengths.

The Mobile Application Testing syllabus [CT_MAT_SYL] includes further details.

7. References

7.1 Standards

The following standards are mentioned in these respective chapters.

[DO-178C]	DO-178C - Software Considerations in Airborne Systems and Equipment Certification, RTCA, 2011 Chapter 2.
[ISO 9126]	ISO/IEC 9126-1:2001, Software Engineering – Software Product Quality Chapter 4 and Appendix A.
[ISO 25010]	ISO/IEC 25010: 2011, Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models Chapters 1, 4 and Appendix A.
[ISO 29119]	ISO/IEC/IEEE 29119-4:2015, Software and systems engineering - Software testing - Part 4: Test techniques Chapter 2.
[ISO 42010]	ISO/IEC/IEEE 42010:2011, Systems and software engineering - Architecture description Chapter 5.
[IEC 61508]	IEC 61508-5:2010, Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems, Part 5: Examples of methods for the determination of safety integrity levels Chapter 2.
[ISO 26262]	ISO 26262-1:2018, Road vehicles — Functional safety, Parts 1 to 12. Chapter 2.
[IEC 62443-3-2]	IEC 62443-3-2:2020, Security for industrial automation and control systems - Part 3-2: Security risk assessment for system design Chapter 4.

7.2 ISTQB® Documents

[CTAL_TTA_OVIEW]	ISTQB® Technical Test Analyst Advanced Level Overview v4.0
[CT_SEC_SYL]	Security Testing Syllabus, Version 2016
[CT_TAE_SYL]	Test Automation Engineer Syllabus, Version 2017
[CTFL_SYL]	Foundation Level Syllabus, Version 2018
[CT_PT_SYL]	Performance Testing Syllabus, Version 2018
[CT_MBT_SYL]	Model-Based Testing Syllabus, Version 2015
[CTAL_TM_SYL]	Test Manager Syllabus, Version 2012
[CT_MAT_SYL]	Mobile Application Testing Syllabus, 2019
[ISTQB_GLOSSARY]	Glossary of Terms used in Software Testing, Version 3.5, 2020
[CT_AuT_SYL]	Automotive Software Tester, Version 2018

7.3 Books and articles

[Andrist20] Björn Andrist and Viktor Sehr, C++ High Performance: Master the art of optimizing the functioning of your C++ code, 2nd Edition, Packt Publishing, 2020

[Beizer90] Boris Beizer, "Software Testing Techniques Second Edition", International Thomson Computer Press, 1990, ISBN 1-8503-2880-3

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

[Kaner02] Cem Kaner, James Bach, Bret Pettichord; “Lessons Learned in Software Testing”; Wiley, 2002, ISBN: 0-471-08112-4

[McCabe76] Thomas J. McCabe, "A Complexity Measure", IEEE Transactions on Software Engineering, Vol. SE-2, No. 4, December 1976, pp. 308-320

[Utting07] 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® cannot be held responsible if the references are not available anymore.

[Web-1] http://www.nist.gov (NIST National Institute of Standards and Technology)

[Web-2] http://www.codeproject.com/KB/architecture/SWArchitectureReview.aspx

[Web-3] http://www.W3C.org

[Web-4] https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project

[Web-5] https://whatwg.org

[Web-6] https://validator.w3.org/

[Web-7] https://dl.acm.org/doi/abs/10.1145/3340433.3342822

Chapter 2: [Web-7]

Chapter 4: [Web-1] [Web-4]

Chapter 5: [Web-2]

Chapter 6: [Web-3] [Web-5] [Web-6]

8. Appendix A: Quality Characteristics Overview

The following table compares the quality characteristics described in the now replaced ISO 9126-1 standard (as used in the 2012 version of the Technical Test Analyst syllabus) with those in the newer ISO 25010 standard (as used in the latest version of the syllabus).
Note that functional suitability and usability are covered as part of the Test Analyst syllabus.

ISO/IEC 25010	ISO/IEC 9126-1	Notes
Functional Suitability	**Functionality**	New name is more accurate, and avoids confusion with other meanings of "functionality"
Functional completeness		Coverage of the stated needs
Functional correctness	Accuracy	More general than accuracy
Functional appropriateness	Suitability	Coverage of the implied needs
	Interoperability	Moved to Compatibility
	Security	Now a characteristic
Performance efficiency	**Efficiency**	Renamed to avoid conflicting with the definition of efficiency in ISO/IEC 25062
Time behaviour	Time behaviour	
Resource utilization	Resource utilization	
Capacity		New sub-characteristic
Compatibility		New characteristic
Coexistence	Coexistence	Moved from Portability
Interoperability		Moved from Functionality (Test Analyst)
Usability		Implicit quality issue made explicit
Appropriateness recognizability	Understandability	New name is more accurate
Learnability	Learnability	
Operability	Operability	
User error protection		New sub-characteristic
User interface aesthetics	Attractiveness	New name is more accurate
Accessibility		New sub-characteristic
Reliability	**Reliability**	
Maturity	Maturity	
Availability		New sub-characteristic
Fault tolerance	Fault tolerance	
Recoverability	Recoverability	
Security	**Security**	No previous sub-characteristics
Confidentiality		No previous sub-characteristics
Integrity		No previous sub-characteristics
Non-repudiation		No previous sub-characteristics
Accountability		No previous sub-characteristics
Authenticity		No previous sub-characteristics

Maintainability	**Maintainability**	
Modularity		New sub-characteristic
Reusability		New sub-characteristic
Analysability	Analysability	
Modifiability	Stability	More accurate name combining changeability and stability
Testability	Testability	
Portability	**Portability**	
Adaptability	Adaptability	
Installability	Installability	
	Coexistence	Moved to Compatibility
Replaceability	Replaceability	

9. Index

accountability, 27
action word-driven, 48
adaptability, 27
adaptability testing, 40
analyzability, 27
anti-pattern, 43
API testing, 13, 17
Application Programming Interface (API), 17
architectural reviews, 43
atomic condition, 13, 15
attack, 32
authenticity, 27
availability, 27

benchmark, 29

capacity, 27
capture/playback, 46
capture/playback tool, 48
code reviews, 44
coexistence, 27
coexistence/compatibility testing, 40
cohesion, 23
compatibility, 27
confidentiality, 27
control flow, 13
control flow analysis, 21, 22
control flow coverage, 15
coupling, 23
cyclomatic complexity, 21, 22

data flow analysis, 21, 22
data security considerations, 30
data-driven, 48
decision predicates, 14
decision testing, 13, 15
definition-use pair, 21
dynamic analysis, 21, 24
  memory leaks, 24
  overview, 24
  performance efficiency, 25
  wild pointers, 25
dynamic maintainability testing, 38

emulator, 46, 52

fault injection, 34, 50
fault seeding, 46, 50
fault tolerance, 27

installability, 27
installability testing, 39
integrity, 27

keyword-driven, 48

load testing, 36

maintainability, 23, 27
maintainability testing, 38
master test plan, 29
maturity, 27
MC/DC, 15
memory leak, 21
metrics
  performance, 25
model-based testing, 46
modifiability, 27
modified condition/decision testing (MC/DC), 15
modularity, 27
multiple condition coverage, 16
multiple condition testing, 13, 16

non-repudiation, 27

operational profile, 27, 41
organizational considerations, 30

performance efficiency, 27
performance efficiency test planning, 37
performance efficiency test specification, 38
portability, 27
portability testing, 39, 40
product quality characteristics, 28
product risk, 10

quality attributes for technical testing, 27
quality characteristic, 27

recoverability, 27
reliability, 27
reliability growth model, 27
reliability testing, 32
remote procedure calls (RPC), 17
replaceability, 27
replaceability testing, 40
resource utilization, 27
reusability, 27
reviews, 42
  checklists, 43
risk assessment, 10, 11
risk identification, 10, 11

risk mitigation, 10, 12
risk-based testing, 10

scalability testing, 37
security, 27
  buffer overflow, 31
  cross-site scripting, 31
  denial of service, 31
  logic bombs, 31
  man in the middle, 31
security test planning, 31
security testing, 30
service-oriented architectures (SOA), 17
short-circuiting, 16
simulator, 30, 46, 52
stakeholder requirements, 29
standards
  DO-178C, 19, 20
  IEC 61508, 19
  IEC 62443, 31
  ISO 25010, 11, 28, 32, 35, 39, 40
  ISO 26262, 19, 20
  ISO 29119, 14, 15, 18
  ISO 42010, 43
  ISO 9126-1, 28
  M/376, 51
  Section 508, 51
statement testing, 13, 14
static analysis, 21, 22, 23
stress testing, 37

test automation project, 47
test environment, 29
test execution, 46
test tools, 50
  build automation, 52
  component testing, 52
  fault injection, 50
  fault seeding, 50
  hyperlink verification, 51
  mobile testing, 52
  model-based testing, 52
  performance, 50
  web tools, 51
testability, 27
time behavior, 27

virtual users, 50

white-box technique, 13
wild pointer, 21