

TMAP[®]: High-performance quality engineering Syllabus

Version 1.3
Released 23 December 2022



Copyright notice

Copyright © Sogeti Nederland B.V. 2022. All rights reserved.

This document may be copied in its entirety, or extracts made, if the source is acknowledged.

- Any individual or training provider may use this syllabus as the basis for a training course if Sogeti is acknowledged as the copyright owner and the source of the syllabus.
- Any individual or group of individuals may use this syllabus as the basis for articles, books, or other derivative writings if Sogeti is acknowledged as the copyright owner and the source of the syllabus.

TMAP® is a registered trademark of Sogeti Nederland B.V.

Revision history

Version	Date	Author	Remarks
0.1	20 July 2020	Bert Linker	Initial version
0.3	03 August 2020	Rik Marselis	Worked out Learning Objectives
0.4	12 August 2020	Bert Linker	Internal review
0.5	28 Augustus 2020	Bert Linker	Second internal review
0.6	01 September 2020	Bert Linker	Change requests by exercises-team
0.7	9 October 2020	Bert Linker & Rik Marselis	Integrated subjects of chapter 7
0.8	16 October 2020	Bert Linker & Rik Marselis	Version for review by SIG members
0.9	18-December-2020	Bert Linker & Rik Marselis	Basis for pilot-training-course
0.95	12 May 2021	Wouter Ruigrok, Guido Nelissen & Rik Marselis	Changed order of some LO's in the course based on feedback from pilot course
1.0	8 July 2021	Rik Marselis	Final version
1.3	23 December 2022	Rik Marselis	Yearly update (note: 1.1 and 1.2 were skipped)

Table of Contents

- Table of Contents 3
- 0. Introduction to this syllabus..... 5
 - 0.1. TMAP®: Quality engineering certification scheme 5
 - 0.2. Purpose of this syllabus 5
 - 0.3. Brief introduction to the other TMAP certifications 6
 - 0.4. Format of this training course and syllabus 6
 - 0.5. Learning objectives and K-levels explained 6
 - 0.6. Learning objectives and K-levels for this certification 7
 - 0.7. The TMAP®: High-performance quality engineering - exam 10
 - 0.8. Target audience and prerequisites for candidates..... 10
 - 0.9. Accreditation of training providers 10
 - 0.10. Literature 11
 - 0.11. Acknowledgements 11
- 1. Session 1 12
 - 1.1. The VOICE model of business delivery and IT delivery (LO01; K2)..... 12
 - 1.2. Terms relevant to quality and testing (LO46; K1) 12
 - 1.3. Introduction to Performing QA & testing topics (LO08; K2) 12
 - 1.4. IT delivery models (LO02; K1) 13
 - 1.5. Scrum (LO03; K1) 13
 - 1.6. DevOps (LO04; K1) 13
 - 1.7. Quality measures (LO22; K2) 13
 - 1.8. Quality Risk Analysis & Test Strategy (LO14; K3) 14
 - 1.9. Specification and Example (LO23; K3) 14
 - 1.10. Acceptance criteria (LO15; K3) 14
- 2. Session 2 15
 - 2.1. CI/CD pipeline (LO06; K2)..... 15
 - 2.2. Capabilities (LO07; K2) 15
 - 2.3. Infrastructure (LO11; K3) 15
 - 2.4. Experience-based testing overview (LO43; K1) 16
 - 2.5. Checklist (LO44; K3) 16
 - 2.6. Monitoring & control (LO09; K3) 16
 - 2.7. Reporting & alerting (LO10; K3)..... 17
- 3. Session 3 18
 - 3.1. Clean architecture (quality aspects) (LO48; K2) 18
 - 3.2. Test-driven development & Specification and Example (LO24; K2) 18
 - 3.3. Unit testing principles (LO49; K2) 18
 - 3.4. Static Code Analysis with tooling (LO47; K2) 19
 - 3.5. Automation (LO19; K2)..... 19

TMAP: High-performance quality engineering – syllabus

- 3.6. Code coverage (LO33; K2) 19
- 3.7. Mutation testing tests the tests (LO29; K3) 19
- 3.8. Process-oriented test design overview (LO31; K2)..... 20
- 3.9. State transition testing (LO32; K3)..... 20
- 4. Session 4 21
 - 4.1. Condition-oriented test design overview (LO34; K2) 21
 - 4.2. Condition -, Decision - & Condition Decision - & Multiple Condition Coverage (LO35; K1) 21
 - 4.3. Modified Condition Decision Coverage (LO36; K3) 21
 - 4.4. Semantic Test (LO37; K3) 22
 - 4.5. Elementary Comparison Test (LO38; K3) 22
 - 4.6. Quality characteristics and non-functional testing (LO45; K2) 22
 - 4.7. Quality characteristic Maintainability (LO28; K2) 23
- 5. Session 5 24
 - 5.1. Test varieties (LO27; K2) 24
 - 5.2. Reviewing (LO16; K3)..... 24
 - 5.3. Pull requests (LO17; K3) 24
 - 5.4. Cross-functional teams (LO05; K2)..... 25
 - 5.5. Data-oriented test design overview (LO39; K2)..... 25
 - 5.6. Equivalence partitioning (LO40; K1) 25
 - 5.7. Boundary Value Analysis (LO41; K1) 25
 - 5.8. Data Combination Test (LO42; K3) 26
- 6. Session 6 27
 - 6.1. Selecting and combining approaches and techniques (LO30; K2) 27
 - 6.2. Test execution (LO20; K3) 27
 - 6.3. Investigate & assess outcome (LO21; K3) 27
 - 6.4. Feature toggles (LO25; K2) 28
 - 6.5. Metrics (LO12; K3)..... 28
 - 6.6. Continuous improvement (LO13; K3)..... 28
 - 6.7. Test data management (LO18; K3) 28
- 7. Description of additional subjects 29
 - 7.1. Infrastructure-as-code and infrastructure verification 29
 - 7.2. Clean Architecture 33
 - 7.3. Test-Driven Development & Specification and Example approach..... 35
 - 7.4. Static code analysis with SonarQube 38
 - 7.5. Unit testing principles..... 40
 - 7.6. State Transition Testing..... 45
 - 7.7. Condition-oriented testing with MCDC 51

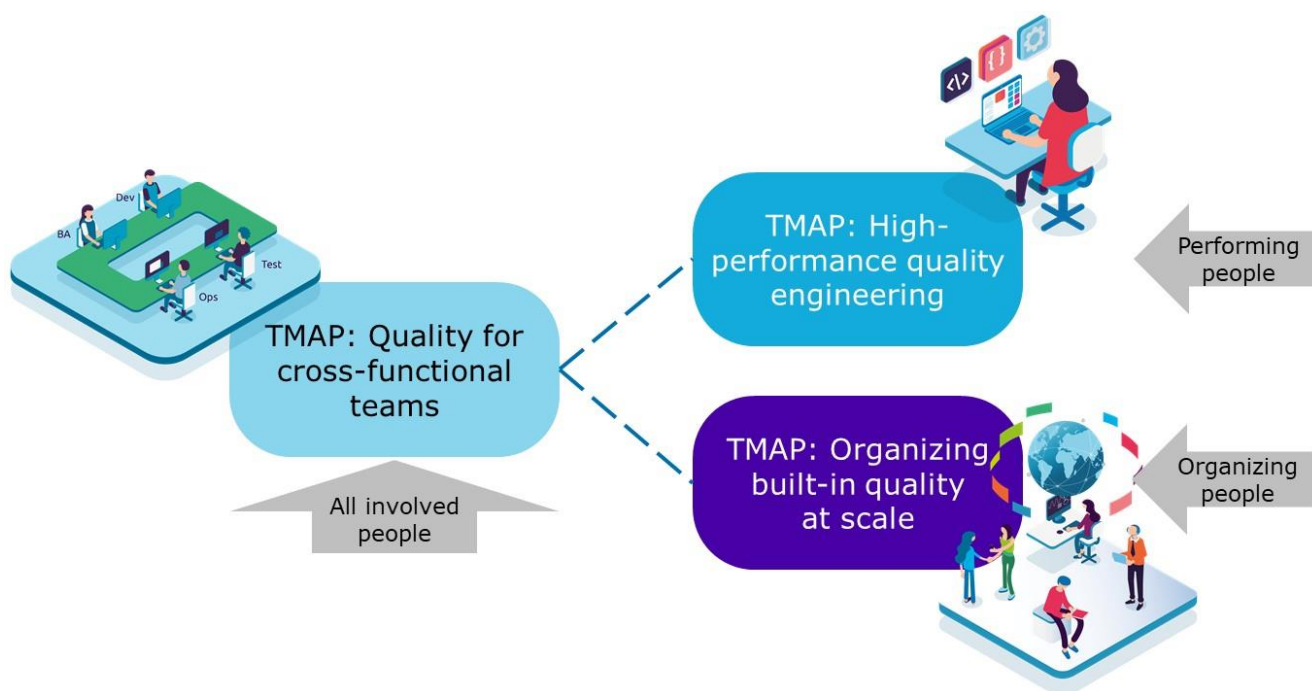
0. Introduction to this syllabus

0.1. TMAP®: Quality engineering certification scheme

In today’s IT world cross-functional teams are expected to deliver business value with the right quality at speed. This requires high-performance IT delivery models such as DevOps and Scrum, which may be extended to a hybrid IT delivery model such as the Scaled Agile framework (SAFe®).

The TMAP® body of knowledge for quality engineering & testing supports working towards built-in quality and takes the need for quality in products, processes and people far beyond just testing.

The new TMAP® certification scheme tailors to the needs of three target audiences. The figure below shows the certifications and indicates that the first certification “TMAP®: Quality for cross-functional teams” provides knowledge necessary for the other two certifications.



0.2. Purpose of this syllabus

The training course “**TMAP®: High-performance quality engineering**” focuses on the knowledge and skills that professionals need to perform operational QA & testing activities. It is all about delivering software – which will generate business value – at the right time with the desired quality!

This syllabus is the basis for the training course “**TMAP®: High-performance quality engineering**” and provides directions for the associated examination and certification. This is a 3-day training course, consisting of six sessions. Every session takes 3 hours (excluding breaks). There is a separate exam of 1.5-hours, for which an optional separate 1-day exam training can be followed.

0.3. Brief introduction to the other TMAP certifications

There are two other certifications in the TMAP certification scheme:

Many people are working in, or are related to, a high-performance IT delivery team, such as in DevOps or Scrum. In the training course “**TMAP®: Quality for cross-functional teams**” these people will acquire the required knowledge and skills that are important for building quality in their IT system and gathering information necessary to establish confidence that the pursued business value can be achieved. It is a 3-day training course with a 1-hour exam.

Organizing QA & testing requires orchestrating, arranging, planning, preparing and controlling the activities. The training course “**TMAP®: Organizing built-in quality at scale**” enables professionals that are responsible for organizing QA & testing to acquire necessary knowledge and skills to enable teams to achieve this. It is a 3-day training course with a separate exam of 1.5 hours.

0.4. Format of this training course and syllabus

The 3-day training course consists of 6 sessions with a minimum of 3 hours (that is 18 contact hours in total). The number of hours mentioned is excluding homework (such as self-study), logistical preparation of the exam and breaks. Candidates must prepare to spend about 5 to 15 hours of individual study and preparation for the exam.

The order of chapters and sections in this syllabus is according to the sequence of the training course, which gives a mix of theoretical and practical subjects. Every training session is a separate chapter in this syllabus and the sections each cover a learning objective.

Chapter 7 of this syllabus contains supporting knowledge from the TMAP body of knowledge website, which is additional to what is described in the book “Quality for DevOps teams”.

0.5. Learning objectives and K-levels explained

Learning objectives (LOs) are brief statements that describe what you are expected to know after studying each subject. The relevant information for the learning objectives can be found in the book “Quality for DevOps teams” and in chapter 7 of this syllabus. With each LO there is a reference to the relevant chapter(s) or section(s). The LOs are used to create the examination for achieving the “TMAP®: High-performance quality engineering” certification.

Each learning objective has a corresponding cognitive level of knowledge (K-level).

These K-levels, based on Bloom’s modified taxonomy, are as follows:

- K1: Remember (knowledge). The candidate should remember or recognize a term or a concept.
- K2: Understand (comprehension). The candidate should select an explanation for a statement related to the question subject.
Examples are: The candidate... can explain, recognizes examples related to the subject, understands, is able to recite, is aware of, can indicate, can distinguish.
- K3: Apply (application). The candidate should select the correct application of a concept or technique and apply it to a given context.
Examples are: The candidate... can relate, can enumerate, can select, can compose, can identify, is able to apply, can assign, can propose.

An overview of the learning objectives for this certification and their corresponding K-levels is given in the next section.

0.6. Learning objectives and K-levels for this certification

Learning objectives in the order in which the subjects appear in the book Quality for DevOps teams.		K-level	Section	
			in this syllabus	Literature in the book or syllabus
The VOICE model				
L001	The VOICE model of business delivery and IT delivery	K2	§ 1.1	Ch 3
IT delivery models				
L002	IT delivery models	K1	§ 1.4	Ch 7, Ch 9 intro;
L003	Scrum	K1	§ 1.5	§ 9.1
L004	DevOps	K1	§ 1.6	§ 1.1, § 9.2 intro, § 9.2.1, § 9.2.2
Continuous quality engineering				
L005	Cross-functional teams	K2	§ 5.4	Ch 2 introduction; § 2.2 introduction, § 2.4, § 16.1
CI/CD pipelines and tooling				
L006	CI/CD pipeline	K2	§ 2.1	§ 6.1; § 6.2; § 6.3 intro, § 9.2.4
L007	Capabilities	K2	§ 2.2	§ 6.3; § 6.4
QA & testing topics				
L008	Introduction to Performing QA & testing topics	K2	§ 1.3	Ch 11; Ch 13
L009	Monitoring & control	K3	§ 2.6	§ 4.1, Ch 17; § 35.9
L010	Reporting & alerting	K3	§ 2.7	§ 5.4, § 17.1.5 Ch 19
L011	Infrastructure	K3	§ 2.3	Ch 22, syllabus § 7.1
L012	Metrics	K3	§ 6.5	Ch 24 through § 24.4
L013	Continuous improvement	K3	§ 6.6	Ch 25 intro, § 25.2.4
L014	Quality Risk Analysis & Test Strategy (and link this to the voice model)	K3	§ 1.8	§ 5.2.1, § 5.2.2, Ch 26; Ch 35 introduction

Learning objectives in the order in which the subjects appear in the book Quality for DevOps teams.		K-level	Section	
			in this syllabus	Literature in the book or syllabus
LO15	Acceptance criteria	K3	§ 1.10	§ 5.6; Ch 27; § 35.2.2
LO16	Reviewing	K3	§ 5.2	Ch 29; § 35.2.1, § 35.6
LO17	Pull requests	K3	§ 5.3	§ 29.1.1.1
LO18	Test data management	K3	§ 6.7	Ch 31
LO19	Automation	K2	§ 3.5	Ch 32 intro, § 32.2, § 32.3, § 32.6
LO20	Test execution	K3	§ 6.2	Ch 33
LO21	Investigate & assess outcome	K3	§ 6.3	Ch 34
Quality measures and skills				
LO22	Quality measures	K2	§ 1.7	Ch 28
LO23	Specification and Example	K3	§ 1.9	§ 35.2
LO24	Test-driven development & Spec. and example	K2	§ 3.2	§ 35.3, syllabus § 7.3
LO25	Feature toggles	K2	§ 6.4	§ 35.8
Test varieties				
LO27	Test varieties	K2	§ 5.1	Ch 37
LO28	Quality characteristic Maintainability	K2	§ 4.7	Ch 41
LO29	Mutation testing tests the tests	K3	§ 3.7	Ch 42
Test design				
LO30	Selecting and combining approaches and techniques	K2	§ 6.1	§ 45.6
Coverage-based testing				
LO31	Process-oriented test design overview	K2	§ 3.8	§ 45.2
LO32	State transition testing	K3	§ 3.9	§ 45.2, syllabus § 7.6
LO33	Code coverage	K2	§ 3.6	§ 46.8
LO34	Condition-oriented test design overview	K2	§ 4.1	§ 45.3, § 46.4 introduction

Learning objectives in the order in which the subjects appear in the book Quality for DevOps teams.		K-level	Section	
			in this syllabus	Literature in the book or syllabus
LO35	Condition Coverage (CC), Decision Coverage (DC), Condition Decision Coverage (CDC) & Multiple Condition Coverage (MCC)	K1	§ 4.2	§ 46.4.2, § 46.4.3, § 46.4.5
LO36	Modified Condition Decision Coverage (MCDC)	K3	§ 4.3	§ 46.4.2, § 46.4.4, syllabus § 7.7 intro, syllabus § 7.7.1
LO37	Semantic Test	K3	§ 4.4	§ 46.4.1, § 46.4.4, syllabus § 7.7.2
LO38	Elementary Comparison Test	K3	§ 4.5	§ 46.4.1, § 46.4.4, syllabus § 7.7.3
LO39	Data-oriented test design overview	K2	§ 5.5	§ 45.4
LO40	Equivalence partitioning	K1	§ 5.6	§ 46.5
LO41	Boundary Value Analysis	K1	§ 5.7	§ 46.5
LO42	Data Combination Test (including EP, BVA and Pairwise)	K3	§ 5.8	§ 46.6
Experience-based testing				
LO43	Experience-based testing overview	K1	§ 2.4	§ 43.4, § 47.1
LO44	Checklist	K3	§ 2.5	§ 29.1.1.1; § 46.7, § 47.2
Quality characteristics				
LO45	Quality characteristics and non-functional testing	K2	§ 4.6	Appendix
Terminology				
LO46	Terms relevant to quality and testing	K1	§ 1.2	Ch1 introduction § 5.5; § 18.3
Additional Subjects				
LO47	Static Code Analysis with tooling	K2	§ 3.4	§ 6.1, Syllabus § 7.4
LO48	Clean architecture (quality aspects)	K2	§ 3.1	Syllabus § 7.2
LO49	Unit testing principles	K2	§ 3.3	Syllabus § 7.5

Note: LO26 had been discarded, the number is not in use.

0.7. The TMAP®: High-performance quality engineering - exam

The format of the exam is multiple choice. There are 40 questions. There are no explicit questions regarding K1 learning objectives. Each correctly answered question for a learning objective at K2-level gives 1 point, at K3-level it gives 2 points. There are 20 K2 questions and 20 K3 questions so in total 60 points can be gained. To pass the exam, at least 66% of the points (that is 40 points) must be gained.

The exams and certificates are provided by the independent exam provider iSQI.

For more information about exams please visit:

www.isqi.org or www.TMAPcert.com.



0.8. Target audience and prerequisites for candidates

This training course is for all people working in or with high-performance IT delivery teams (such as DevOps and Scrum) that are responsible for or heavily involved in quality engineering such as QA professionals, testers and operations people. Other relevant roles include business analysts, product owners, developers, quality engineers, users, scrum masters, agile coaches, release train engineers, etc.

The candidates are expected to have basic IT knowledge and experience. There is no required previous certification, but attendees are expected to have the knowledge about and/or experience in the subjects of the training course "**TMAP®: Quality for cross-functional teams**", so this certification is highly recommended. Some subjects of that training course will return in this course, either to be elaborated on in more detail, or as a quick recap as the foundation for new subjects.

0.9. Accreditation of training providers

Training providers that want to prepare candidates for the exam will need to acquire accreditation from iSQI. For more information please contact TMAP2020@iSQI.org

Training providers may choose between creating their own material and having it accredited through iSQI or licensing the standard training material through iSQI.

0.10. Literature

Exam literature:

- The book “Quality for DevOps teams” (ISBN 978-90-75414-89-9), available on www.ict-books.com and other bookstores, both in paper and ePub version.
- TMAP glossary: <https://www.tmap.net/page/tmap-glossary-online>.
- Descriptions in chapter 7 of this syllabus, based on building blocks on www.tmap.net.
Note: for the exam, texts in this syllabus supersede texts on the website.

Additional literature:

- The TMAP body of knowledge website – www.tmap.net

Other additional literature (specifically for trainers to acquire more in-depth knowledge):

- The Agile Manifesto – www.agilemanifesto.org
- The Scrum Guide – www.scrumguides.org
- ISO25010 - www.iso.org/standard/35733.html
- Also please refer to the references in the book “Quality for DevOps teams”.

0.11. Acknowledgements

This syllabus was created by a diverse team. We would like to thank the following people (in no particular order) for their contributions in writing and reviewing this document:

Sogeti people: Eveline Moolenaars, Anja Leijssen, Berend van Veenendaal, Bert Linker, Dennis Geurts, Guido Nelissen, Koen Roos, Mark van der Walle, Othmar Hawker, Ralph Klomp, Richard Pommee, Rik Marselis, Ugur Eksi, Wouter Ruigrok, André van Pelt, Jolene Persoon, Henri Davids, Martin Gijzen and Daniël Venhuizen.

iSQI people: Stephan Goericke, Erika Paasche, Corinna Flemming - Vogt, Anke Fransen, Valida Saronjic.

TMAP Special Interest Group members: Cees van Barneveld, Bruno Lepretre, Leo van der Aalst, Okan Cakmak, Rob Flier, Guido Dulos, Gitte Ottosen and Daisy Fischlein Steffensen.

1. Session 1

Learning objectives

LO01, LO02, LO03, LO04, LO08, LO14, LO15, LO22, LO23, LO46.

1.1. The VOICE model of business delivery and IT delivery (LO01; K2)

High-performance IT delivery teams (such as in Scrum and DevOps) use the VOICE model as a foundation to structure and organize their work.

The candidate can give a description of the VOICE model and knows that it's an acronym of Value, Objectives, Indicators, Confidence and Experience.

The candidate can explain why "Value" and "Objectives" are vital starting points for IT delivery.

Book: chapter 3.

1.2. Terms relevant to quality and testing (LO46; K1)

High-performance IT delivery, Quality, Static testing, Dynamic testing, Error, Fault, Failure, Incident, Problem, Anomaly and Defect.

The candidate has knowledge of these terms.

Book: Chapter 1 introduction, section 5.5, section 18.3.

1.3. Introduction to Performing QA & testing topics (LO08; K2)

Every IT delivery model, framework, mindset, organization etc. has its own development approach, workflow, phases, roles, work products and/or activities. We defined a set of generic QA & testing activities – the so-called "topics" –, which are applicable – in one way or another – to all these different development approaches.

The candidate understands that there are two overarching groups: Organizing topics and Performing topics.

The candidate is able to describe the performing topics.

Book: chapter 11, chapter 13.

1.4. IT delivery models (LO02; K1)

An IT delivery model is a conceptual framework which supports a software development process and describes all assets and competencies. We distinguish three groups of IT delivery models: Sequential IT delivery, High-performance IT delivery and Hybrid IT delivery.

The candidate has knowledge of the groups of IT delivery models.

Book: chapter 7, chapter 9 introduction.

1.5. Scrum (LO03; K1)

Scrum is a framework with which people address and solve complex problems in an adaptive manner, while delivering the highest value products in a rewarding and creative way.

The candidate has basic knowledge of the Scrum framework including roles, events and artifacts.

Book: section 9.1.

1.6. DevOps (LO04; K1)

DevOps is a cross-functional systems engineering culture that aims at unifying systems development (Dev) and systems operations (Ops) with the ability to create and deliver fast, cheap, flexible and with adequate quality, whereby the team as a whole is responsible for the quality.

The candidate has knowledge of DevOps, including the DevOps activities.

Book: sections 1.1, 9.2 introduction, 9.2.1.

1.7. Quality measures (LO22; K2)

Quality was, is and remains a challenge within the IT industry. Quality engineering consists of a great number of possible activities, the so-called quality measures.

The candidate understands that quality measures are integrated with all DevOps activities and why the set of quality measures must be cohesive.

The candidate can explain the three groups of quality measures.

Book: chapter 28.

1.8. Quality Risk Analysis & Test Strategy (LO14; K3)

A test strategy is the allocation of quality measures to balance the investment in testing and to make an optimal distribution of effort over test varieties and test approaches to give insight in test coverage and test intensity. Often this is based on the quality risk levels and the pursued business value.

The candidate is able to determine where to focus the QA & testing activities by investigating the quality risks involved with the IT system and identifying appropriate QA and testing measures, i.e. in such a way that the aspects of the VOICE model are justified.

This learning objective has a strong relationship with learning objectives LO01 and LO15.

Book: section 5.2.1, 5.2.2; chapter 26, chapter 35 introduction.

1.9. Specification and Example (LO23; K3)

In order to achieve a shared common understanding of what “it” is that should be built and try to build “it” right the first time, you can use Specification and Example mapping approaches.

The candidate is able to conduct an example mapping session.

Book: section 35.2.

1.10. Acceptance criteria (LO15; K3)

A cross-functional team, which is common in DevOps, will agree to deliver an IT product with a specific quality level. This quality level is defined by the acceptance criteria for each user story. The team, the product owner and other stakeholders discuss and collaborate closely so that the acceptance criteria are supported by everyone involved.

The candidate can identify and/or enhance acceptance criteria to a situation and can apply the seven tips [Ravhani 2017] for defining acceptance criteria.

The candidate can write and review a scenario in Gherkin syntax.

This learning objective has a strong relationship with learning objective LO23.

Book: section 5.6; chapter 27; section 35.2.2.

2. Session 2

Learning objectives

LO06, LO07, LO09, LO10, LO11, LO43, LO44.

2.1. CI/CD pipeline (LO06; K2)

In DevOps, a CI/CD pipeline needs to be implemented. Continuous Integration & Continuous Deployment (CI/CD) is seen as the backbone to enable DevOps. It bridges, maybe even closes, the gap between development and operations by automating the building, packaging, testing, provisioning of infrastructure and deployment of applications.

The candidate understands which quality engineering activities are linked to which stages in a CI/CD pipeline.

Book: section 6.1; section 6.2; section 6.3 introduction; section 9.2.4.

2.2. Capabilities (LO07; K2)

With a CI/CD pipeline, steps in the software delivery process are automated. When creating such a – fully – automated CI/CD pipeline, tools with specific capabilities are needed. Tools can frequently change. Therefore, the capabilities need to be well defined to ensure a stable pipeline, and tools need to be selected, based on these capabilities.

The candidate understands that the capabilities are used to define the pipeline and to select suitable tools.

Book: section 6.3; section 6.4.

2.3. Infrastructure (LO11; K3)

In DevOps the responsibility for the infrastructure has shifted from a separate department to the DevOps team itself. As a result of this shift, some important parts of the infrastructure's quality become a team responsibility.

The candidate can perform an infrastructure verification.

Book: chapter 22.

Syllabus: section 7.1.

2.4. Experience-based testing overview (LO43; K1)

Experience-based testing is a group of test approaches that are based on the skills, intuition and experience of the tester. These approaches leave the tester free to design test cases in advance or to create them on the spot during the test execution, mostly testers will do both.

The candidate recognizes approaches that belong to experience-based testing and knows that some level of combination of experience-based and coverage-based testing should be in the test strategy.

Book: section 43.4, section 47.1.

2.5. Checklist (LO44; K3)

Previous experience is an important source of information to prepare and guide quality engineering activities such as reviewing and testing. This previous experience is often stored only in minds of people. An easy way to capture this experience is by listing it in a checklist.

The candidate is able to create, adjust or complement a checklist regarding unit testing, functional testing, non-functional testing and static testing.

Book: section 29.1.1.1; section 46.7, section 47.2.

2.6. Monitoring & control (LO09; K3)

Monitoring and control are intended to promptly identify, report and forecast (gaps in) expected and actual quality, related to the pursued business value.

The candidate can select the right area of monitoring in a given context.

The candidate is able to apply monitoring to check progress and to check to which level indicators are met.

The candidate can create a simple dashboard based on measuring indicators.

This learning objective has a strong relationship with learning objective LO10.

Book: section 4.1, chapter 17, section 35.9.

2.7. Reporting & alerting (LO10; K3)

Testing is about providing different levels of information. Usually there are multiple audiences for the information that the team generates based on their quality engineering activities.

DevOps teams and their stakeholders want to, and need to, have constant and direct insight into the status of the IT system. And if something (either in product or process) deviates from the expectations, they must be alerted as soon as possible. Therefore, DevOps teams will use state-of-the-art tools for reporting and alerting, where on-line real-time dashboards are today perceived as need-to-haves.

The candidate can select relevant information for dashboards & reports.

The candidate is able to analyze and draw conclusions from overview reports.

The candidate can select a proper way of alerting stakeholders.

This learning objective has a strong relationship with learning objective LO09.

Book: section 5.4; section 17.1.5, chapter 19.

3. Session 3

Learning objectives

LO19, LO24, LO29, LO31, LO32, LO33, LO47, LO48, LO49.

3.1. Clean architecture (quality aspects) (LO48; K2)

Clean architecture is about organizing your code in such a way so that it's easy to understand and easy to change as the development grows. By applying universal rules of software architecture, you can dramatically improve developer productivity throughout the life of any software system.

The candidate understands that loosely coupled code with high cohesion is the goal of clean architecture.

Syllabus: section 7.2.

3.2. Test-driven development & Specification and Example (LO24; K2)

Test-driven development (TDD) is a development method for software in which unit tests are written first and then the code. This cycle can be repeated as many times as needed to make the code fully functional according to the requirements.

The candidate comprehends the steps within Test-driven development.

The candidate is able to recall the Three laws of TDD and the principles behind them.

The candidate understands that TDD may be combined with one of the Specification and Example approaches.

Book: section 35.3.

Syllabus: section 7.3.

3.3. Unit testing principles (LO49; K2)

Writing unit tests basically is just like writing production code. However, the rules that apply for writing good production code do not always apply to creating a good unit test. Well written tests are assets while badly written tests are a burden to your application. Following unit testing principles helps in creating good unit tests that pay off more than they cost.

The candidate understands the practices and importance of unit testing.

The candidate can compare the four groups of costs and benefits of unit testing.

Syllabus: section 7.5.

3.4. Static Code Analysis with tooling (LO47; K2)

Static code analysis of source code identifies (potential) faults, vulnerabilities and poor coding practices automatically using tools. A commonly used tool is SonarQube.

The candidate understands the benefits of static code analysis and how a tool like SonarQube contributes to code quality.

Book: section 6.1.

Syllabus: section 7.4.

3.5. Automation (LO19; K2)

The demand for continuous testing has created a renewed focus on test automation. Test automation is one of the main opportunities to meet the need for quality at speed, but also requires a structured approach in order to effectively realize such a vision.

The candidate understands that automating everything that is repeated is contributing to efficient software development. (note: automate only when it is useful, automation is not a goal in itself)

The candidate can describe items that could be automated as part of “everything as code” automation.

The candidate understands continuous testing and the different test automation solutions.

Book: chapter 32 introduction, sections 32.2, 32.3 and 32.6.

3.6. Code coverage (LO33; K2)

Code coverage can be measured by specific tools during the execution of tests. We distinguish various code coverage types.

The candidate can explain why some code coverage types are preferred over others and which code coverage type to select given a certain context.

Book: section 46.8.

3.7. Mutation testing tests the tests (LO29; K3)

Can the products of testing also be tested? Certainly! And they should be tested! How can you see if the tests are complete? To some extent, this can be done by mutation testing.

The candidate can apply mutation testing to verify the quality of a test set.

Book: chapter 42.

3.8. Process-oriented test design overview (LO31; K2)

The process-oriented coverage group contains test design techniques that are based on processes, for example a business process or a program algorithm structure.

The candidate is able to select test design techniques that belong to process-oriented test design.

Book: section 45.2.

3.9. State transition testing (LO32; K3)

State Transition testing is a test design technique that focuses on states, events that initiate a transition to another state and actions resulting from such event. Tests are designed to execute valid and invalid state transitions. Multiple coverage levels can be achieved, indicated as n-switch coverage.

The candidate can apply the test design technique “State transition testing” to a given test basis with 0-switch coverage and 1-switch coverage.

Book: section 45.2.

Syllabus: section 7.6.

4. Session 4

Learning objectives

LO28, LO34, LO35, LO36, LO37, LO38, LO45.

4.1. Condition-oriented test design overview (LO34; K2)

The condition-oriented coverage group contains test design techniques that are based on the behavior of decision points and the conditions that determine the result of a decision.

The candidate is able to select test design techniques and coverage types that belong to condition-oriented test design.

Book: section 45.3, section 46.4 introduction.

4.2. Condition -, Decision - & Condition Decision - & Multiple Condition Coverage (LO35; K1)

CDC is a coverage type, from the coverage group Condition-oriented, that ensures the possible outcomes of each condition and of the decision are tested at least once. This implies both "condition coverage" and "decision coverage". MCC is a coverage type that covers all combinations of all condition values.

The candidate knows about the coverage types Condition Coverage (CC), Decision Coverage (DC), Condition Decision Coverage (CDC) and Multiple Condition Coverage (MCC).

Book: sections 46.4.2, 46.4.3 and 46.4.5.

4.3. Modified Condition Decision Coverage (LO36; K3)

MCDC is a coverage type, from the coverage group Condition, that ensures that every possible outcome of a condition is the determinant of the outcome of the decision, at least once. MCDC implies also "condition/decision coverage".

The candidate can apply the coverage type Modified Condition Decision Coverage (MCDC) to a given test basis.

This learning objective has a strong relationship with learning objectives LO37 and LO38.

Book: sections 46.4.2 and 46.4.4.

Syllabus: section 7.7 intro and section 7.7.1.

4.4. Semantic Test (LO37; K3)

A semantic test is a test with which for example the validity of data input is tested using the semantic rules for the relationships of the data on the input device and other data, for example in the database, in the system or on the input device. The semantic test is often executed in combination with the syntactic test.

The test basis consists of the semantic rules, being single decision points, that specify what a data item should comply with in order to be accepted by the system as valid input. Semantic rules are connected with the relationships between data. These relationships may be between the data within a screen, between data on various screens and between input data and existing data in the database.

The candidate can apply the Semantic test design technique to a given test basis in combination with Modified Condition Decision Coverage (MCDC).

This learning objective has a strong relationship with learning objective LO 36.

Book: section 46.4.1 and 46.4.4

Syllabus: section 7.7.2

4.5. Elementary Comparison Test (LO38; K3)

The elementary comparison test (ECT) is a thorough technique for the detailed testing of functionality. The necessary test basis is pseudo-code or a comparable specification in which multiple decision points and functional paths are worked out in detail.

The ECT aims at thorough coverage of the decision points and not specifically at combining functional paths. The basic coverage type used is Modified Condition Decision Coverage (MCDC). But different coverage types can be applied to the ECT.

The candidate can apply the Elementary Comparison test design technique to a given test basis in combination with Modified Condition Decision Coverage (MCDC).

This learning objective has a strong relationship with learning objective LO36.

Book: section 46.4.1 and 46.4.4

Syllabus: section 7.7.3

4.6. Quality characteristics and non-functional testing (LO45; K2)

When deciding on their test varieties many testers start with distinguishing between functional testing and non-functional testing. This refers to the quality characteristics. These are a very useful tool to identify various characteristics of quality that are important for the stakeholders of an IT-system.

The candidate can interpret the eight main quality characteristics for product quality and the five main quality characteristics for quality in use.

Book: Appendix.

4.7. Quality characteristic Maintainability (LO28; K2)

Teams are often focused on the cost-effective development of IT systems. In high-performance IT delivery the team should find a proper balance between development and maintenance costs. Maintainability is the degree of effectiveness and efficiency with which a product or system can be modified by the intended maintainers. Maintainability can be tested both statically and dynamically.

The candidate understands the ideas behind maintainability testing and can give examples of ways to perform these tests.

The candidate understands the ideas behind testability.

Book: chapter 41.

5. Session 5

Learning objectives

LO05, LO16, LO17, LO27, LO39, LO40, LO41, LO42.

5.1. Test varieties (LO27; K2)

IT products are different. People are different. Projects are different. Environments are different. So, it would be an illusion to think that one-size-fits-all exists for testing. You need variety in your testing.

The candidate understands that determining the varieties in testing is based on the relevant quality characteristics and other relevant perspectives, such as the spheres of testing, the testing pyramid and the testing quadrants.

The candidate can explain that test varieties may include static testing among which testability reviews. The candidate also understands the importance of agreeing on a test strategy.

This learning objective has a strong relationship with learning objectives LO14 and LO15.

Book: chapter 37.

5.2. Reviewing (LO16; K3)

Static testing consists of informal reviewing, formal reviewing and static analysis.

The candidate can use the n-amigos session approach to perform an informal review of a given test basis and provide feedback about the quality and/or identify anomalies.

Book: chapter 29, section 35.2.1, section 35.6.

5.3. Pull requests (LO17; K3)

When using a check-out/check-in mechanism for code, as is common in continuous integration pipelines, a pull request is part of the check-in process.

The candidate can review the changed code and can verify if the change was OK, using a checklist.

This learning objective has a strong relationship with learning objectives LO16 and LO44.

Book: section 29.1.1.1.

5.4. Cross-functional teams (LO05; K2)

Working in a cross-functional team means that the team as a whole is responsible for delivering value. The team has all competencies and skills to perform the necessary tasks and no team member has the monopoly on performing any task. This way the team can always go forward, even when a team member is temporarily not available. And of course, a team can work together with specialists from other teams or support groups for specific tasks. A person can have multiple roles sequentially or even in parallel. It is not common for people to have a specific function, since that would easily lead to monopolies on certain tasks.

The candidate demonstrates an understanding of how a cross-functional team operates and can state in which way a cross-functional team operates more effectively than a multi-disciplinary team or when working in silos.

Book: chapter 2 introduction; section 2.2 introduction, section 2.4, section 16.1.

5.5. Data-oriented test design overview (LO39; K2)

The data-oriented coverage group contains test design techniques that use the structure or behavior of the data that is used in the IT system.

The candidate is able to select test design techniques that belong to data-oriented test design.

Book: section 45.4.

5.6. Equivalence partitioning (LO40; K1)

In the application of equivalence classes, the entire value range of a parameter is partitioned into classes. In a specific class the system behavior is similar (equivalent) for every value of the parameter.

The candidate knows the basic concepts of Equivalence Partitioning (EP).

Book: section 46.5.

5.7. Boundary Value Analysis (LO41; K1)

Boundary Value Analysis is a test design technique based on the fact that around a boundary in the value range of a variable there's a higher risk of faults in a system.

The candidate knows the difference between two-value -, three-value – and four-value Boundary Value Analysis.

Book: section 46.5.

5.8. Data Combination Test (LO42; K3)

The Data Combination Test tests combinations of values of data items. Coverage can be determined in various ways. A classification tree visualizes the relations between data items.

The candidate can apply the Data Combination Test (DCoT) test design technique to a given test basis and can use a classification tree to visualize the relations between data items.

The candidate understands the different coverage levels for the DCoT.

Book: section 46.6.

6. Session 6

Learning objectives

LO12, LO13, LO18, LO20, LO21, LO25, LO30.

6.1. Selecting and combining approaches and techniques (LO30; K2)

There are a great number of approaches and techniques, which one(s) should you use? You can choose static testing or dynamic testing or (often) both. But what should you do in your specific situation?

The candidate understands how the factors test basis, quality risks, quality characteristics and skills play a role in selecting relevant approaches and techniques.

Book: section 45.6.

6.2. Test execution (LO20; K3)

Test execution is the execution of tests by running the system under test and this way obtain the actual results that can be compared with the expected results to determine whether the tests have passed or failed.

The candidate can perform a pre-test and can execute explicit and implicit testing and register the results such that these can be investigated and assessed.

This learning objective has a strong relationship with learning objective LO21.

Book: chapter 33.

6.3. Investigate & assess outcome (LO21; K3)

When the team members execute the test scenarios and test scripts, they compare the actual outcomes with the expected outcomes and assess the results.

The candidate can identify whether the executed test has passed or failed or was not run. The candidate can also perform the steps for creating an anomaly, in the case of a failed test.

This learning objective has a strong relationship with learning objective LO20.

Book: chapter 34.

6.4. Feature toggles (LO25; K2)

A feature toggle (also called feature flag) is a powerful technique, allowing teams to modify system behavior without changing code. A mechanism that enables deployment of features that are not finished yet, or of which the quality is uncertain. Code can be deployed to the production environment without being available to the users by turning off the feature toggle. At a later stage it can be made available by just turning the feature toggle on. And if a problem occurs it can be turned off again.

The candidate can compare the four categories of feature toggles and understands when which category can be applied.

Book: section 35.8.

6.5. Metrics (LO12; K3)

A DevOps team wants to be in control. Therefore, the team needs to measure relevant parameters. The resulting metrics give useful information about the status and are a starting point for improvement measures.

Everyone is involved in choosing “good” metrics and setting business goals; organization, team and individuals. But what are “good” metrics?

The candidate can apply the three fundamentals to determine “good” metrics.

The candidate can select efficiency and effectiveness metrics to a given situation.

Book: chapter 24 up to and including section 24.4.

6.6. Continuous improvement (LO13; K3)

DevOps teams work in an everchanging world where the common expectation is that quality and speed improve. They constantly need to improve their way of working and adapt to changed circumstances.

The candidate is able to apply the Deming cycle to a given situation.

Book: chapter 25 introduction, section 25.2.4.

6.7. Test data management (LO18; K3)

Some test varieties need a high volume of test data but the exact data doesn't matter that much. Other test varieties require a relatively limited set of test data but the values of the data must be carefully aligned across various systems, possibly even across multiple organizations. The implementation of the right test data management practices is a key consideration for the realization of significant time and efficiency gains in quality assurance and testing.

The candidate can apply data scrambling and rule-based masking based on a list of general personal data.

Book: chapter 31.

7. Description of additional subjects

This chapter contains the in-depth descriptions to support learning objectives that are not based on contents of the book "Quality for DevOps teams". These additional subjects are based on information that is available on the TMAP body of knowledge website (www.tmap.net).

Note: for the exam, the descriptions in this chapter supersede any texts on the website, even in case the website would contain other (more up-to-date) descriptions. This syllabus is regularly updated to include the latest insights.

7.1. Infrastructure-as-code and infrastructure verification

7.1.1. Infrastructure-as-code

Definition: Infrastructure-as-code (IaC) is the process of managing and provisioning computer environments through machine-readable definition files, rather than physical hardware configuration or interactive configuration tools.

Infrastructure-as-code (IaC) is an approach whereby an infrastructure configuration is scripted or described by files that are stored in version control, and changes are pushed out to the datacenter in a controlled manner. This parallels the discipline of source control and build promotion used in software development, hence 'as code'.

Definition: A test environment is a composition of parts, such as hardware and software, connections, environment data, tools and operational processes in which a test is carried out.

A popular open source tool for infrastructure automation is HashiCorp - Terraform. It uses a DSL (Domain Specific Language) to script the desired infrastructure. Using this approach provides consistent and repeatable environment changes, reducing the manual effort involved, especially in troubleshooting environmental differences.

Applying IaC will eliminate several problems. E.g.:

- Labor intensive; building and configuring a complete infrastructure for an IT-system traditionally is a time-consuming activity
- Error prone; manual building the infrastructure for an IT-system, following a checklist or not, is an error prone process
- Hard to exactly rebuild infrastructure for other environments, like another test or acceptance environment
- Difference between design and reality; all above mentioned possible problems result in a difference between designed and approved infrastructure architecture and the implemented infrastructure.

Using IaC, a lot of problems can be converted in advantages:

- Modularization; the DRY principle (Don't Repeat Yourself) applies also for IaC similar as applications code. Split the complete infrastructure into small reusable pieces of code.
- Maintainability and versioning
- Testability; See infrastructure verification
- Automation

- Repeatability; Similar infrastructures can be deployed over and over again
- Security, compliance and policies
- Collaboration and code review

The extra advantage of tools like Terraform is that it is agnostic for the target cloud platform. A single tool and domain specific language can be used for a variety of cloud providers.

Example of a Hashicorp - Terraform code snippet: Creation of AWS server instance and AWS database instance.

```
// Creation of an Amazon Web Services (AWS) server instance and an AWS database instance.  
// This example doesn't use variables.
```

```
// Declare AWS instance
```

```
resource "aws_instance" "web" {  
  ami = "${data.aws_ami.ubuntu.id}"  
  instance_type = "t2.micro"  
  
  tags = {  
    Name = "testland-webserver"  
  }  
}
```

```
// Declare data block which is used during creation of AWS instance
```

```
data "aws_ami" "ubuntu" {  
  most_recent = true  
  
  filter {  
    name = "name"  
    values = ["ubuntu/images/ubuntu-trusty-14.04-amd64-server-*"]  
  }  
  
  owners = ["0123456789"]  
}
```

```
// Declare AWS database
```

```
resource "aws_db_instance" "default" {  
  allocated_storage = "20"  
  storage_type = "gp2"  
  engine = "postgres"  
  engine_version = "12.3"  
  instance_class = "t2.micro"  
  name = "postgres"  
  username = "postgres"  
  password = "postgres"  
  final_snapshot_identifier = "postgres"  
}
```

7.1.2. Infrastructure verification

In DevOps the implementation of the complete IT-system is a responsibility of the team, this includes the implementation and configuration of the infrastructure. To make use of all benefits described in the previous section, it is recommended to define the infrastructure as code and apply the same quality engineering principles as done on application code.

Infrastructure verification is key when you make use of IaC, without automated tests IaC must be considered broken. The purpose of verifying the infrastructure is similar as it is for application source code.

Verification (and also validation) of the Infrastructure takes place regarding all fundamental DevOps activities. The following sections briefly describe this per fundamental DevOps activity.

Plan

Does the infrastructure architecture adhere to key quality characteristics? For example: security and maintainability, but also characteristics like fault-tolerance, fail-over, scalability and resilience should meet the desired requirements. The planning phase is the phase where the architectural design should be verified against policies and industry regulations. During the planning phase preventive quality measures are used to prevent insufficient quality.

Code

Transforming the architectural design into code is an agile process. The architecture is divided into small, reusable, flexible, easy codable and testable modules. The modules and also IaC should be verified on different aspects.

- Syntactical correctness
- Code formatting (linting)
- Code testing (deep linting)
- Accordance with company or team policies and industry regulations

With linting you make sure that all code uses the same formatting and styling regardless which team member writes the code. On top of that, deep linting can be applied which is called 'code testing'. Code testing is a technique to verify the IaC on the correctness of the types that are used. E.g. AWS uses instance-type for their computing machine. The IaC should not use unavailable, incorrect, incomplete or outdated types.

In accordance with the stages in the build pipeline dynamic testing is also part of testing the IaC, State testing is the test variety which is performed on the IaC. The purpose of state testing is verifying the IaC in a runtime environment, so the infrastructure is actually deployed on the target platform. State testing the IaC is not related to the state transition testing technique.

State testing the infrastructure is done in 3 steps

1. Deploy real infrastructure which needs to be tested
2. Verify basic attributes of the infrastructure:
 - a. Assigned compute power (CPU's/cores)
 - b. Assigned internal memory (RAM)
 - c. Assigned disk space
 - d. Version of OS (Eg.: RHEL8, MS Window Server 2019)
 - e. Patch level of OS
 - f. Type of database (Postgres, MySQL)
3. Undeploy the infrastructure under test
 - a. Verify that deployed IaC is completely removed

By state testing IaC the code is verified (and validated) in a real environment which eliminates the need for mocks, stubs or other artificial components. This testing type can be slow for large infrastructures and potentially also be very costly.

It's advised to perform the state test in an isolated 'sandbox' environment to prevent interference with existing infrastructure.

Detective quality measures are predominantly used during the coding activity to detect a possible insufficient level of quality.

Integrate

The most extended test for IaC is the integration test. During this test the IaC is tested including the deployment and configuration middleware components. Middleware components in the infrastructure can occur in many forms, for example: application servers, web application, firewall, database engines and web servers.

Attributes which should be assessed during this test are (non-exhaustive):

- Is server listening on correct port?
- Are security certificates correctly installed?
- Are the firewall rules implemented? (Thus, can server A connect to server B, or the opposite, is the connection from server A to server B denied.)
- Are databases correctly configured? Listening to correct port? Is naming of schema correct?
- Are proxy rules configured?

Deploy

Deployment of the IaC takes place at several stages. During the team test stage and the business test stage the IaC is verified in combination with the application under test.

Verification during the deploy activity has a focus on stability, security, performance, resource consumption. Relevant test varieties are smoke testing, performance testing and security testing. All tests are performed under close watch of monitoring-tooling to detect resource consumption.

The last deployment in the iteration is to the production environment. During the deployment activity detective quality measures are used.

Operate and Monitor

During the last activities of the DevOps cycle the complete IT-system including the IaC is live and used by the customer. The IT-system is monitored on several aspects and feedback is collected from the IT-system. This feedback is used to continuously improve the IT-system. Corrective and detective quality measures are used to detect and improve lack in quality.

Sources

<https://itnext.io/principles-patterns-and-practices-for-effective-infrastructure-as-code-e5f7bbe13df1>

<https://blog.gruntwork.io/5-lessons-learned-from-writing-over-300-000-lines-of-infrastructure-code-36ba7fadeac1>

<https://medium.com/faun/terraform-acceptance-testing-the-basics-5450d35a4421>

<https://www.infoq.com/presentations/iac-terraform-testing/>

<https://www.thoughtworks.com/radar/techniques/infrastructure-as-code>

<https://www.terraform.io/>

7.2. Clean Architecture

Any software project should not only consist of so-called 'clean code' but should also have a clean architecture. What does it mean to have a clean architecture? Over the years there have been many ideas about this, which may differ in their details, but are all based on the same principles. The most important objective is the separation of concerns. This is achieved by dividing software into layers to achieve loose coupling and high cohesion.

Coupling

Most software evolves over time. If we want to upgrade our software with a snazzy new front-end, a faster database or an easy-to-use framework, parts of the software will have to be replaced. To enable this, software should be designed in such a way that it supports such replacements by requiring minimal changes, by making sure that each of the components has (or makes use of) little or no knowledge of the definitions of other separate components. This is called loose coupling.

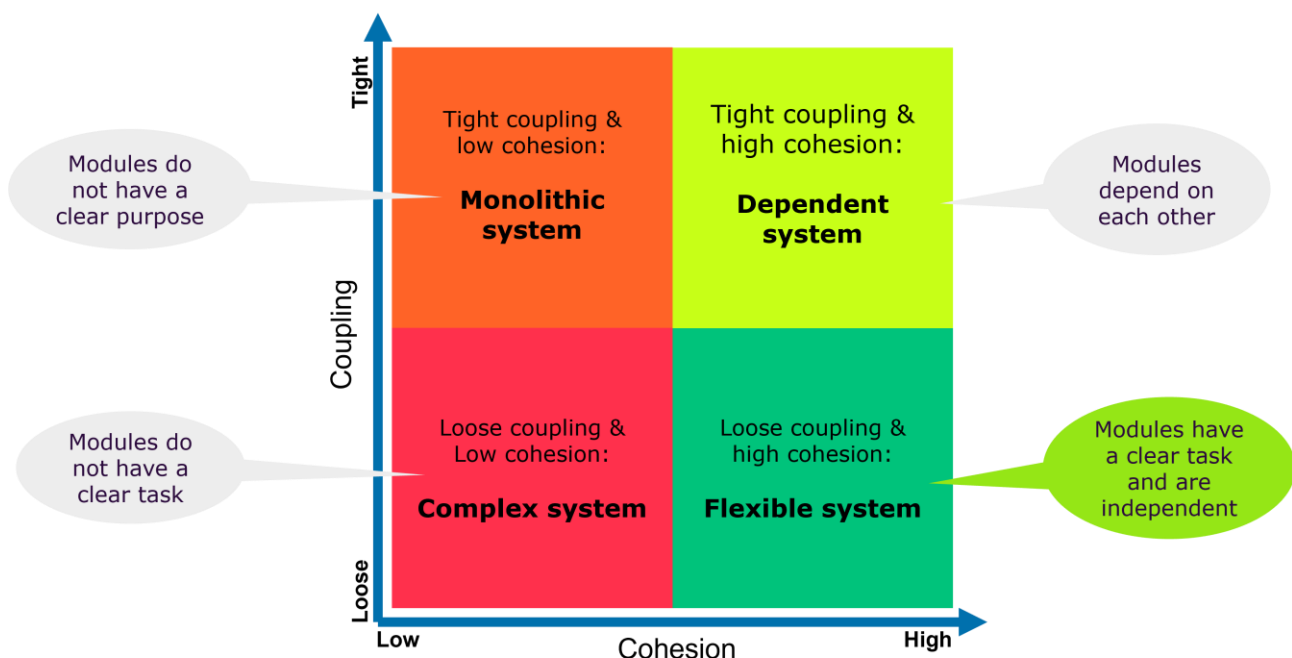
Many integrated products, such as laptops or tablets, are good examples of tight coupling: if your laptop screen breaks, you're probably better off buying a new laptop. Because the screen is fixed to the laptop and won't come loose, it makes replacing the screen very expensive. A loosely coupled computer would allow effortlessly changing the screen, which is the case for a desktop.

So, if we want to achieve loose coupling, the code should be written as generic as possible and only go into specifics whenever unavoidable. For instance, if we want to swap out a database for a new one, this should not affect the functionality of the code, but should only affect the class that is dedicated to connecting to the database.

Cohesion

Cohesion in software engineering is the degree to which the elements of a certain module (or class) belong together. Low cohesion (or coincidental cohesion) is when parts of a module are grouped arbitrarily, whereas high cohesion (or functional cohesion) is when parts of a module all contribute to a single well-defined task of the module.

The Model-View-Controller design pattern is a good example of high cohesion: all methods related to the model are grouped in a model class, all methods relating to displaying the information are grouped in the view class and all methods that are related to processing events are written in the controller class.



The picture on the previous page shows four quadrants:

- 1) Loose coupling and low cohesion results in a complex system with modules that don't have a clear task and have poor maintainability.
- 2) Tight coupling and low cohesion results in a monolithic system where modules don't have one clear purpose but serve various goals.
- 3) Tight coupling and high cohesion results in a dependent system where modules depend on each other and can't function without each other.

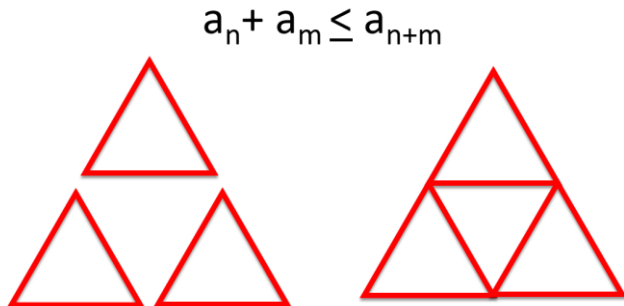
4) Goal: Loose coupling & High cohesion

In software development we strive for loose coupling and high cohesion, because it leads to increased module reusability, better system maintainability and reduced module complexity, making the code easier to understand and debug. From a quality engineering perspective this is the best way to achieve quality at speed (both in development and in operations).

7.3. Test-Driven Development & Specification and Example approach

TDD is a design method

Design is superadditive (the whole is greater than the sum of its parts) and TDD is about designing a solution to a problem.



One of the biggest problems with software development, is not knowing what will happen, when things interact. Our code might be used in scenarios we didn't expect, and therefore our code might not be able to handle it. Applying TDD helps in identifying these unsuspected interactions.

The TDD process is described in chapter 35.3 of the book.

7.3.1. TDD Process: The three laws of TDD

1. *You are not allowed to write any production code unless it is to make a failing unit-test pass.*
2. *You are not allowed to write any more of a unit-test than is sufficient to fail; and compilation failures are failures.*
3. *You are not allowed to write any more production code than is sufficient to pass the one failing unit-test.*

Reference: Uncle Bob¹

These laws can be translated into:

1. Write a unit test.
2. You should only write this single unit test.
3. Write the code for the unit test to pass, but no more code than that.

At first glance these 3 laws might seem very unproductive, since it will require the developer to switch between writing unit-tests and code. But there are many benefits of doing that, which we will make clear in the following section.

¹ <http://butunclebob.com/ArticleS.UncleBob.TheThreeRulesOfTdd>

Design

TDD will make the design testable. Any piece of code can be written so complicated, that it will be impossible to test. But if a test is written first, then the code to make the test pass, is automatically testable.

Also, TDD will make you avoid too complex thinking. You might get a great idea, then expanding it, then expanding it even more, only to get disturbed and forgetting most of it. Complex thinking is very demanding on the brain, which might make you overlook critical situations. TDD will make you take one unit-step at a time, for you to be able to foresee possible consequences for each unit, instead of dealing with the whole complexity at once.

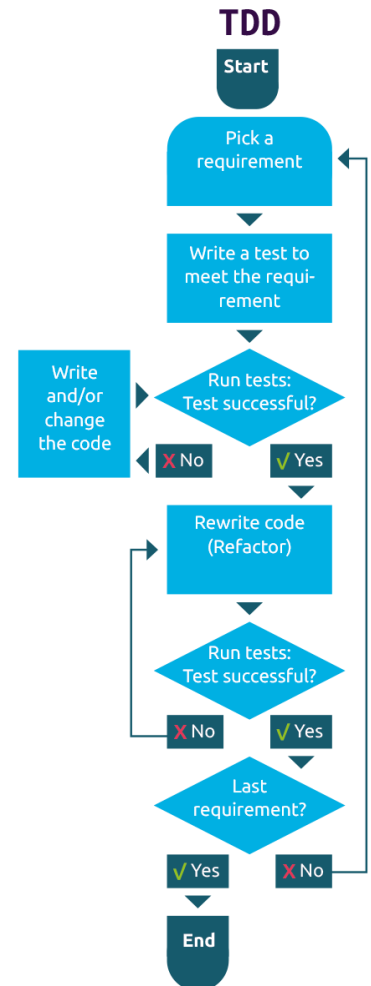
Improving code

It will give other developers the option to improve, optimize or clean the existing code. Anyone with a bright idea, can try to improve the code, with less fear of breaking existing known functionality.

Debugging

Unit-testing will not only avoid a lot of debugging but will also make it easier to debug correctly. Why use time on figuring out something you already have figured out once? A unit-test is a way to document, how something was debugged. Not only will a unit-test remember it better than you, but another developer can simply use the unit-test, without asking you for help.

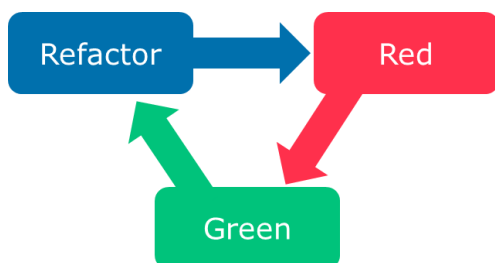
(the picture on the right of this page is figure 35.5 of the book)



7.3.2. TDD Principles

We distinguish several TDD principles:

Red, Green Refactor



As described in the book:

Writing a unit-test without writing any code, will make the unit-test fail (RED)

Writing the code to make the unit-test pass, will make the unit-test pass (GREEN)

Updating the code, while the unit-tests still passes, will not break the code. (REFACTOR)

Keep It Simple Stupid (KISS)

Writing a unit-test before any code, will minimize the amount of test cases, because you will only test the code for what is needed.

Writing a unit-test after code, will make either:

TMAP: High-performance quality engineering – syllabus

- The number of test cases explode (to cover all the possible scenarios (both needed and unneeded for the product)).
- The test cases irrelevant, because they test some parts of code, which we don't know if they are needed for the product or not.
- The test cases to never be coded, because of bad experience with the 2 previous points (explosion or irrelevance).

(note: KISS is also known as Keep it Simple & Short)

It's All About Collaboration

TDD is about guiding the team to design and build the right software. The unit-test (which may also be called component test) is designed to test if the solution will meet the goal. Writing code that passes the unit test, is your proof that the code meets the goal.

After a unit-test is deployed, the ownership is transferred from the developer to the team.

What a developer can do by him/herself:

- **Add** any **unit-test** and make it pass,
- **Change** any piece of **code**, as long as all existing unit-tests still **pass**.
- **Ask** the product owner, architect, etc., when there is no answer to a problem.

What a developer can't do by him/herself (but should do together with (a) team member(s)):

- **Changing** or **removing** any **unit-test**, because it will influence the design and break the product. Use your team to minimize the problems.
- **Insource risk** from the product owner, architect, etc.

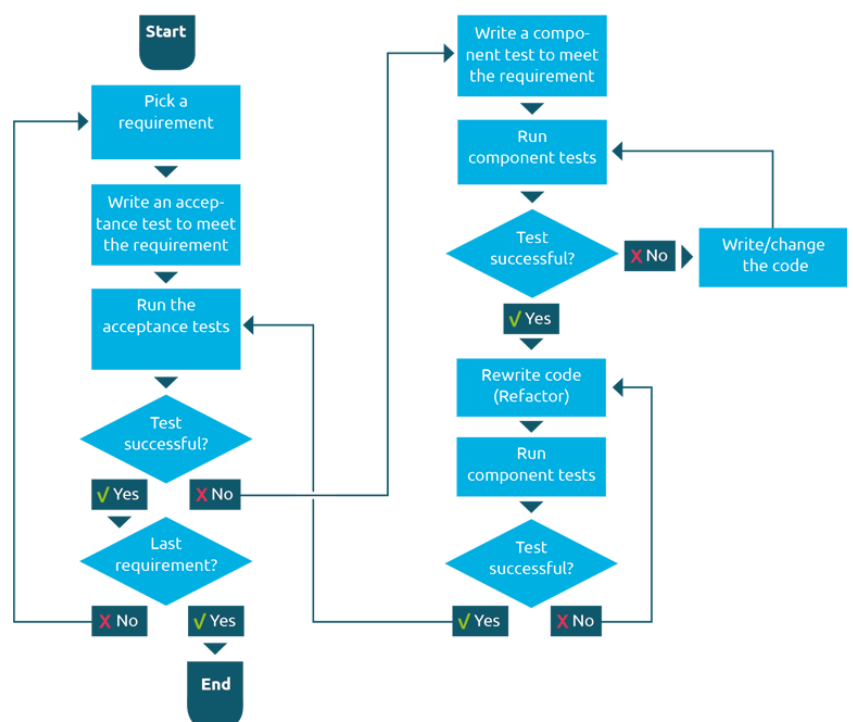
7.3.3. TDD and Specification and Example (SaE)

TDD can be amplified by Specification and Example (SaE) approaches, for example Acceptance Test Driven Development (ATDD) and Behavior Driven Development (BDD). The difference between ATDD and BDD is in the area that ATDD scenarios focus on the "what" question and BDD gives substance to the "how" question. These two also complement each other.

The figure on this page shows how SaE approaches interact with TDD. SaE is shown on the left and TDD on the right.

(this is figure 35.6 from the book)

Looking at the Agile testing quadrants as described in §37.3 of the book, one sees that SaE and TDD apply to the left side of the quadrants. SaE is Business facing and involves the business analyst (BA) and testing roles. TDD is Technology focusing and fits within a Developer role, i.e. part of creating code. However, both have the goal of "Guiding the team", in other words to help the team building the right system.



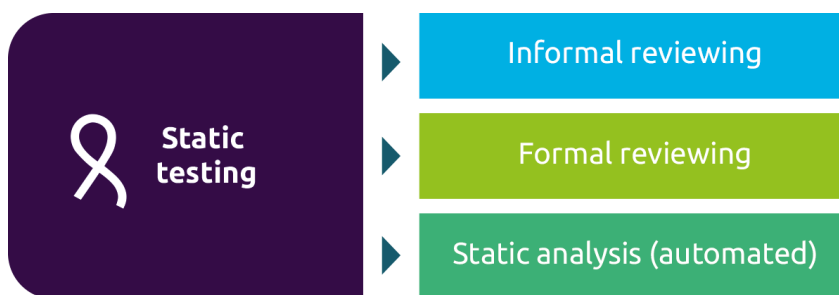
From a BA perspective, ATDD and BDD scenarios can be created which serve as input for performing Development tasks. The value of SaE scenarios in combination with TDD (and unit tests/component tests) is that they cover scenarios that comprise two or more units. A higher coverage at unit level can be pursued combined with a limited number of SaE scenarios. Consider, for example, a business process in which two sub-processes run parallel to each other. The separate sub-processes can be tested at unit level, but the interaction between the two can be covered at BDD and ATDD level.

By combining the different approaches, we can generate an effective and balanced set of scenarios. These scenarios can additionally be identified using test design techniques. Consider, for example, performing a boundary value analysis.

7.4. Static code analysis with SonarQube

Static code analysis is the analysis of software that is performed without actually executing programs, in contrast with dynamic analysis, which is analysis performed on programs while they are executing. Static code analysis is performed on some version of the code.

Static code analysis is part of static analysis which is one of the three groups of static testing as shown in this picture:



Static code analysis is a collection of algorithms and techniques used to analyze source code in order to automatically find (potential) faults, vulnerabilities or poor coding practices. An example of a result of static testing are compiler warnings (which can be useful for finding coding errors), but static code analysis with more sophisticated tools takes that idea a step further to find faults that are traditionally found by dynamic testing.

The tasks solved by static code analysis can be divided into 3 categories:

- Detecting faults or vulnerabilities in programs. These can for example be logic faults (such as unreachable code) and writing on a read only variable.
- Recommendations on code formatting. Some static analyzers allow you to check if the source code corresponds to the code formatting standard used in your company.
- Metrics computation. Software metrics are a measure that lets you get a numerical value of some property of software or its specifications. An example is cyclomatic complexity that can be used to indicate the maintainability of the code.

There are multiple tools that cover one or more aspects of static code analysis. A well-known tool is SonarQube.

SonarQube is an open-source platform for continuous inspection of code quality to perform automatic reviews with static analysis of code to detect faults, code smells, and security vulnerabilities on 20+ programming languages.

SonarQube analyzes source code, measuring the quality level and providing reports for your projects. It combines static and dynamic analysis tools and enables quality to be monitored continuously over time. Everything that affects a code base, from minor styling details to critical design errors, is inspected and evaluated by SonarQube. This way SonarQube enables developers to access and track code analysis data. This data ranges from styling errors, potential faults, and code problems to design inefficiencies, code duplication, lack of test coverage, and excess complexity.

The SonarQube platform analyzes source code from different aspects and hence it drills down to your code layer by layer, moving from the module level down to the class level. At each level, SonarQube produces metric values and statistics, revealing problematic areas in the source that require inspection or improvement. This is very effective in large teams managing a big codebase. Whenever the code is built by a developer it is immediately analyzed, so they will notice any fault or vulnerability themselves as they were inserting them, rather than having to fix them at a later time.

SonarQube can be implemented in your CI/CD pipeline and issues will be only detected when your code is pushed to the repository. For specific aspects of static code analysis, tools can be embedded in the IDE to perform the check real-time.

Sources:

<https://dzone.com/articles/why-sonarqube-1>

https://en.wikipedia.org/wiki/Static_program_analysis.

7.5. Unit testing principles

7.5.1. Unit testing

Unit testing is a very important quality measure that supports the IT delivery objective to deliver quality at speed. (note: unit testing is also called component testing)

The main goal of unit testing is to verify that the implementation does what it intends to do. It is about individually testing the smallest units of code, referred to as modules or components. This allows developers to isolate each component, and whenever they are not according to expectation, fix problems at an early stage of the development lifecycle. Developers aim to test each part of the software in individual components as early as possible in the development process. For example, you have a password field and the requirements are that it needs to be at least 8 characters long, must contain alphanumeric characters and at least one symbol. Good practice is to create test cases that meet these criteria and, more important, also test cases that do not meet these criteria. So, a password like "secret01" should fail and "Secret123!" should pass the test.

Writing unit tests basically is just like writing production code. Well-written tests are assets while badly written tests are a burden. Following unit testing principles helps in creating good unit tests that pay off more than they cost.

7.5.2. Principles

The aim of DevOps teams is to build quality in. Testing the code on desired functionality and quality while writing the code is therefore an effective and efficient way to verify that the implementation actually does what it is intended to do. Some benefits of unit testing are:

- It provides a fast feedback-loop for verifying code changes.
- It provides a safety net – we know that the code works.
- It reduces costs and technical debt (e.g. because it reduces rework).
- It can be applied as regression test.
- It forms the basis for test automation.
- It facilitates easy verification of changes during maintenance.

Unit tests are an effective way to find faults or detect regressions, but unit tests should not be the only testing that is done. Unit tests, by definition, examine each unit of code separately. Unit testing (as part of Test Driven Development) supports designing software components robustly. But when an application is run for real, all those units also have to work together, and the whole is more complex and subtle than the sum of its independently tested parts. So other varieties of testing must also be organized by the team. For example, refer to the testing pyramid for more information. (chapter 37 of the book "Quality for DevOps teams").

That sounds great, but still some people may wonder: Why do you actually want a secondary system to help design or verify your code? Doesn't your source code itself express the design and behavior of your solution?

The benefit of unit testing is correlated with the non-obviousness of the code under test

If you have code of any normal length, already it is not obvious at a single glance – so working out its exact behavior would take time and careful thought. Next additional design - and verification assistance (e.g., through unit testing) is essential to be sure that all situations are handled correctly. For example, if you're coding a system of business rules or parsing a complex hierarchical string

format, there will be too many possible code paths to check at a glance. In scenarios like these, unit tests are extremely helpful and valuable.

Unit testing takes time, apply good practices to be efficient

Designing and executing unit tests of course will require effort and time. On the other hand, it also brings benefits which makes this investment worthwhile.

Here we give some examples of the efforts needed for designing and executing unit tests.

- The time needed for writing unit tests
- The time spent fixing and updating unit tests, either because you've deliberately refactored interfaces between code units or the responsibilities distributed among them, or because tests broke unexpectedly when you made other changes

Some people avoid improving and refactoring application code out of fear that it may break a lot of unit tests and hence incur extra work. This of course is reversing the idea of quality engineering, keeping unit tests in sync with the application code is part of the work and when applying test driven development, creating and improving unit tests is just part of the development process.

FIRST-U rules

To write good unit tests apply the "FIRST-U" rules: Fast, Isolated/Independent, Repeatable, Self-validating, Timely and Understandable.

These rules are described below:

- **Fast**

Unit tests should be fast otherwise they will slow down your development/deployment time and will take longer time to pass or fail. E.g. in TDD short iterations of "change or add code" and running unit tests are performed. If the tests aren't fast enough, this methodology loses its power. Typically, on a sufficiently large system, there will be a few thousand unit tests. If you have 2000 unit tests and the average unit test takes 200 milliseconds to run (which is be considered fast), then it will take 6.5 minutes to run the complete suite. 6.5 minutes may not seem long but imagine you run them multiple times a day, it will use a significant amount of your time. And imagine when the count of these tests increases because new functionalities are added to the application, it will further increase the test execution time. Then the value of your suite of unit tests diminishes as their ability to provide continual, comprehensive, and fast feedback about the health of your system also diminishes.

- **Isolated/Independent**

Never ever write tests which depend on other test cases. No matter how carefully you design them, there will always be possibilities of false alarms. To make the situation worse, you may end up spending a lot of time figuring out which test in the chain has caused the failure.

You should be able to run any one test at any time, in any order.

By making independent tests, it's easy to keep your tests focused only on a small part of behavior. When this test fails, you know exactly what has gone wrong and where. No need to debug the code itself.

The Single Responsibility Principle (SRP) of SOLID Class-Design Principles says that classes should be small and single-purpose. This can (and should) be applied to your tests as well. If one of your test cases can break for more than one reason, consider splitting it into separate test cases.

- **Repeatable**

A repeatable test is one that produces the same result each time you run it. To accomplish repeatable tests, you must isolate them from anything in the external environment, not under your direct control. In these cases use mock objects. They are intended for this very purpose.

On occasion, you'll need to interact directly with an external environmental influence such as a database. You'll want to set up a private sandbox to avoid conflicts with other developers whose tests concurrently alter the database. A good practice is to use in-memory databases.

- **Self-validating**

Tests must be self-validating. This means each test must be able to determine if the actual output is according to the expected output. This determines if the test is passed or failed. There must be no manual interpretation of results. (Manually verifying the results of tests is a time-consuming process that can also introduce more risk)

Make sure you don't do anything silly, such as designing a test to require manual arrange steps before you can run it. You must automate any setup your test requires – even do not rely on the existence of a database and pre-cooked data.

Create an in-memory database, create a schema and insert dummy data and then test the code. This way, you can run this test many times without fearing any external factor which can affect test execution and its result.

- **Timely**

Practically, you can write unit tests at any time. You can wait up to code is production-ready, but you're better off focusing on writing unit tests in a timely fashion. Using Test Driven Development is a good practice to follow.

As a suggestion, you should have guidelines or strict rules around unit testing. You can use review processes or even automated tools to reject code without sufficient tests.

The more you unit test, the more you'll find that it pays off to write smaller chunks of code before tackling a corresponding unit test. First, it'll be easier to write the test, and second, the test will pay off immediately as you flesh out the rest of the behavior in the surrounding code.

- **Understandable**

This is probably the most important practice, even though often missed.

A unit test should have a title in the form of a user story or a description of what it does and what to expect. Don't just give a unit test a simple number such as test1, test2, test3, etc... but assign useful and meaningful names.

Anatomy of a unit test case

The typical anatomy of every unit test is *arrange-act-assert* or *given-when-then*. This pattern is a standard across the industry.

In the arrange (given) section the unit which is tested is initialized in a specific state. Mocks are created, and the expected result is set.

The act section is the actual execution of the unit test case.

The assert section compares the actual output of the act section with the expected output set in the arrange section.

Below is a snippet of Java unit test code which give an example of the anatomy of a unit test

```

@Test
@DisplayName("Test - Allow access boatride TestLand")
void testAllowAccessBoatride() {
    // Arrange the test into a specific state.
    // Arrange: Create an attraction
    final Attraction boatRide = new Boatride(new LengthValidator());

    // Arrange: Create a visitor, set length of visitor to 1.85 mtr
    final Visitor adultVisitor = Visitor.builder()
        .length(185)
        .build();

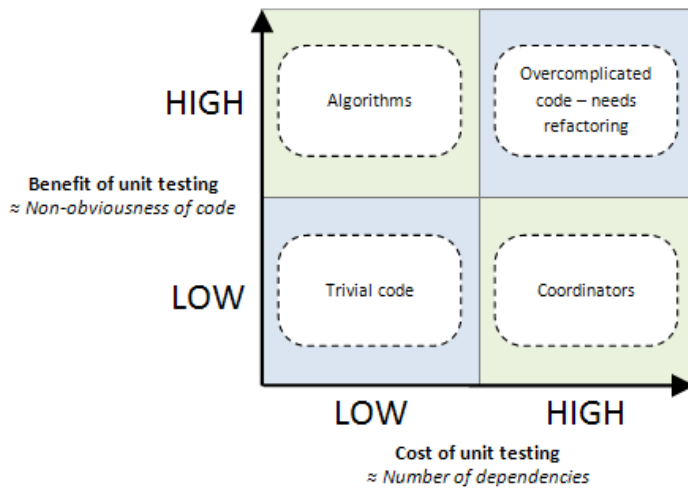
    // Act: Actual invocation of LengthValidator, and testing the functionality
    boolean isAllowed = boatRide.checkLengthVisitor(adultVisitor);

    // Assert: Check output of Act stage with expected result
    assert(true, isAllowed);
}

```

The cost of unit testing a certain code unit is very closely correlated with its number of dependencies on other code units.

Below the costs and benefits of unit testing are put in a simple diagram.



- *Complex code with few dependencies (top left).* Typically, this means self-contained algorithms for business rules or for things like sorting or parsing data. This cost-benefit argument goes strongly in favor of unit testing this code, because it's cheap to do and highly beneficial.
- *Trivial code with many dependencies (bottom right).* This quadrant has been labelled "coordinators", because these code units tend to glue together and orchestrate interactions between other code units. This cost-benefit argument is in favor of not unit testing this code: it's expensive to do and yields little practical benefit. Your time is finite; spend it more effectively elsewhere.
- *Complex code with many dependencies (top right).* This code is very expensive to write with unit tests, but too risky to write without. Usually you can sidestep this dilemma by decomposing the code into two parts: the complex logic (algorithm) and the bit that interacts with many dependencies (coordinator).
- *Trivial code with few dependencies (bottom left).* We needn't worry about this code. In cost-benefit terms, creating unit tests is so easy that you should just do it.

Sources:

<https://blog.stevensanderson.com/2009/08/24/writing-great-unit-tests-best-and-worst-practises>
<https://howtodoinjava.com/best-practices/first-principles-for-good-tests>
<https://blog.stevensanderson.com/2009/11/04/selective-unit-testing-costs-and-benefits>.

7.6. State Transition Testing

Many systems show state-based behavior. State-based models are used to define this behavior. These models can also be used to design tests. With state transition testing several distinct coverage levels can be achieved.


Test approach:	Coverage-based - process-oriented
Test variety:	Functionality testing, process flow testing, menu-structure testing, and more
Test basis:	State Transition diagram(s) and state table(s)
Coverage type:	State transitions

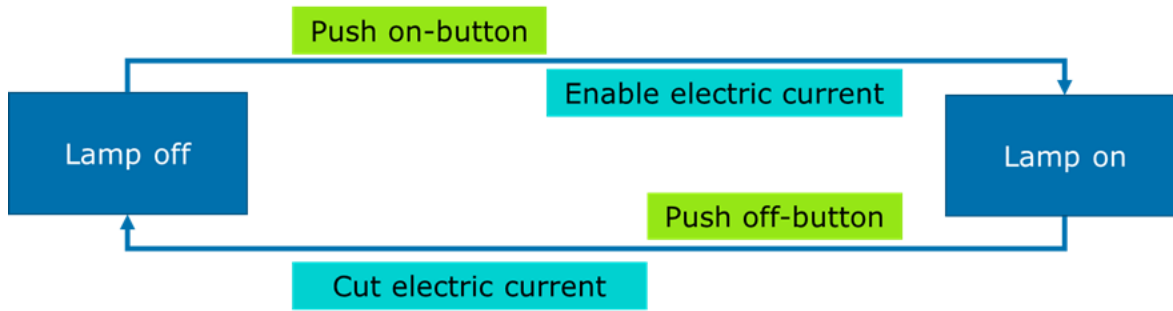
Description

State Transition testing is a process-oriented test design technique that focuses on states, events that initiate a transition to another state and actions resulting from such events. Tests are designed to execute valid and invalid state transitions. State transition testing is used to test whether the system correctly responds to events for example by transitioning from one state to another. Multiple coverage levels can be achieved, indicated as n-switch coverage.

State transition testing is often used to test embedded software that controls machines, but also to test menu-structures in GUI-based systems or other types of systems that have distinct states and a process for getting from one state to another.

The correct behavior of the system is described in a state transition diagram that gives an overview of all states, the transitions between these states, the events that trigger transitions and the actions that result from events.

State	A state is a distinguishable situation of a system. A system can only be in one state at any point in time. A system can transition from one state to another.
 Transition	A transition is a change from one state to another. A transition may also be to the same state (a "transition-to-self").
Event	An event is an occurrence inside or outside a system that can result in a transition.
Action	An action is behavior executed at a particular point in the system. It can also be a string of actions.



Pictured above is an example of a simple state transition diagram. Our example system is a lamp. The lamp can be off or on. If the on-button is pushed, the electric current is enabled and the lamp is turned on. If the off-button is pushed, the electric current is cut and the lamp is turned off.

The transitions between states can also be shown in a state table as shown below.

State	Event	Push on-button	Push off-button
Lamp off		Enable electric current Lamp on	-
Lamp on		-	Cut electric current Lamp off

The left side of the table contains the states, and the top contains the events.

When a transition is valid the resulting state is shown in the corresponding cell of the table. Above the resulting states, the actions may be added, this is optional.

When the transition is invalid, it is shown as a hyphen ('-').

Within state transition testing you can choose to test individual transitions or combinations of transitions.

Since a state transition diagram doesn't show invalid transitions, only test cases with valid transitions can be derived from a state transition diagram.

A state table indicates valid transitions and invalid transitions, so a state table can be used for testing both.

The level of test coverage is related to the number of consecutive transitions that are covered. If every single transition is tested, we achieve "0-switch coverage". 0-switch coverage means that we do not focus on testing consecutive transitions. If sequences of two transitions are tested, so all combinations of two consecutive transitions are tested, we achieve "1-switch coverage". If a higher number of consecutive transitions are tested, we speak of "n-switch coverage" where 'n' is the number of consecutive transitions minus 1. For example, "2-switch coverage" tests combinations of 3 consecutive transitions.

Usually only 0-switch coverage and 1-switch coverage are applied, sometimes 2-switch.

Identify test situations

Test situations for state transition testing can be:

- The individual states – this enables testing if all states can be reached
- The individual transitions – this can be used to ensure 0-switch coverage
- The combinations of “n” transitions – this can be used to ensure “n-1”-coverage

a) Coverage of all states	<ul style="list-style-type: none"> ▪ lamp off ▪ lamp on
b) 0-switch coverage	<ul style="list-style-type: none"> ▪ lamp off – push on-button – lamp on ▪ lamp on – push off-button – lamp off
c) 1-switch coverage	<ul style="list-style-type: none"> ▪ lamp off – push on-button – lamp on – push off-button – lamp off ▪ lamp on – push off-button – lamp off – push on-button – lamp on
d) 2-switch coverage	<ul style="list-style-type: none"> ▪ lamp off – push on-button – lamp on – push off-button – lamp off – push on-button – lamp on ▪ lamp on – push off-button – lamp off – push on-button – lamp on – push off-button – lamp off

etcetera for n-switch (*higher coverage levels are usually only sensible with complex diagrams*)

Create logical test cases

There are two possibilities for creating logical test cases:

- 1) Create a test case for every individual test situation
- 2) Combine multiple test situations in a test case.

Examples:

a) Coverage of all states	1)	<ul style="list-style-type: none"> ▪ TC1: lamp off ▪ TC2: lamp on <p>→ this achieves coverage of all states with two test cases</p>
	2)	<ul style="list-style-type: none"> ▪ TC1: lamp off – push on-button – lamp on <p>→ this achieves coverage of all states with just one test case</p>
b) 0-switch coverage	1)	<ul style="list-style-type: none"> ▪ TC1: lamp off – push on-button – lamp on ▪ TC2: lamp on – push off-button – lamp off <p>→ this achieves 0-switch coverage with two test cases</p>
	2)	<ul style="list-style-type: none"> ▪ TC1: lamp off – push on-button – lamp on – push off-button – lamp off <p>→ this achieves 0- switch coverage with just one test case</p>
c) 1-switch coverage	1)	<ul style="list-style-type: none"> ▪ TC1: lamp off – push on-button – lamp on – push off-button – lamp off ▪ TC2: lamp on – push off-button – lamp off - push on-button – lamp on <p>→ this achieves 1- switch coverage with two test cases</p>
	2)	<ul style="list-style-type: none"> ▪ TC1: lamp off – push on-button – lamp on – push off-button – lamp off – push on-button – lamp on <p>→ this achieves 1- switch coverage with just one test case</p>

When would you use one test case per test situation, and when would you combine multiple test situations in a test case?

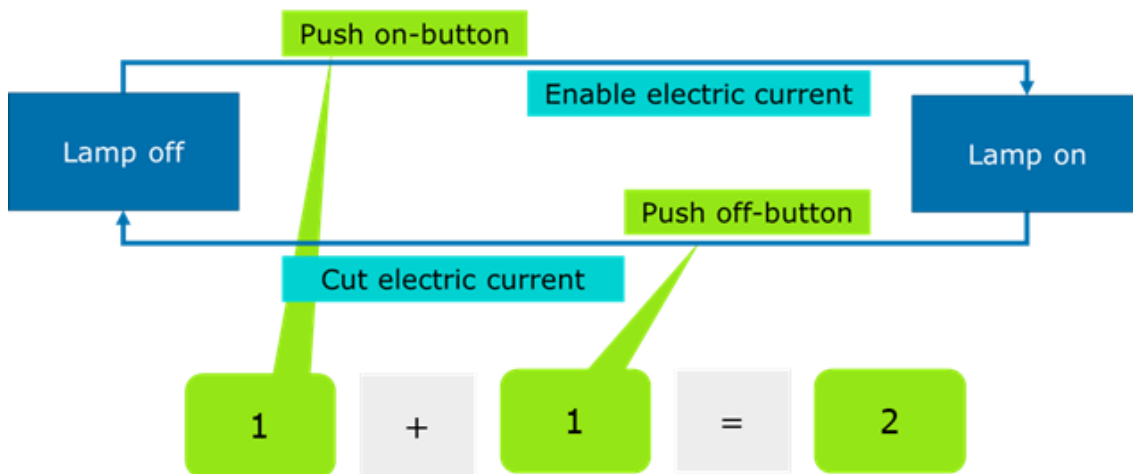
In general, for progression testing you would use many short test cases so that in case of a failure it will be relatively easy to investigate the problem. Also, in unit testing you want to test small parts of the code, so you would also use many short test cases.

For regression testing you would use a few long test cases since you mainly want to establish whether the system that previously passed the test was not negatively impacted by a change. As an example, in end-to-end testing you want to test an entire business process and don't bother about all possible exceptions, so you would also create few long test cases.

How to calculate the number of test situations

To calculate the number of test situations needed for 0-switch coverage, simply count the number of transitions.

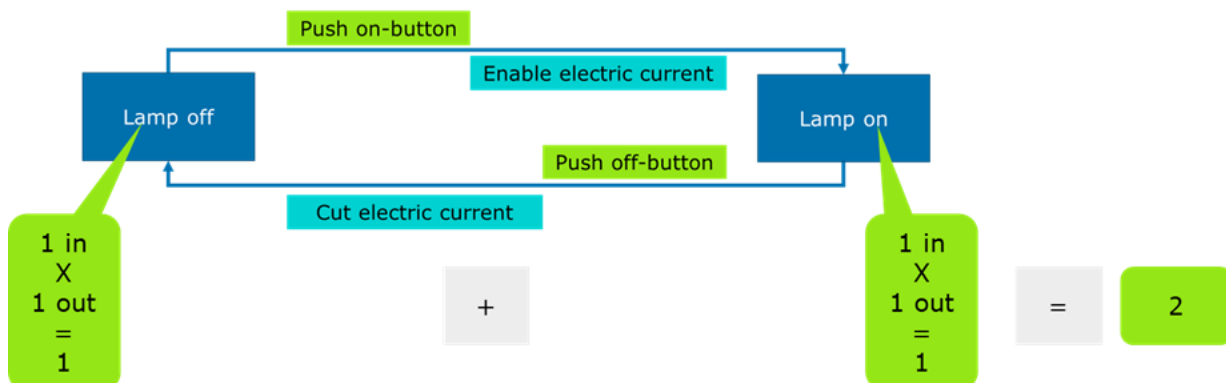
To achieve 0-switch coverage, every test situation must be part of at least one testcase.



The number of test situations for 1-switch testing is calculated as follows:

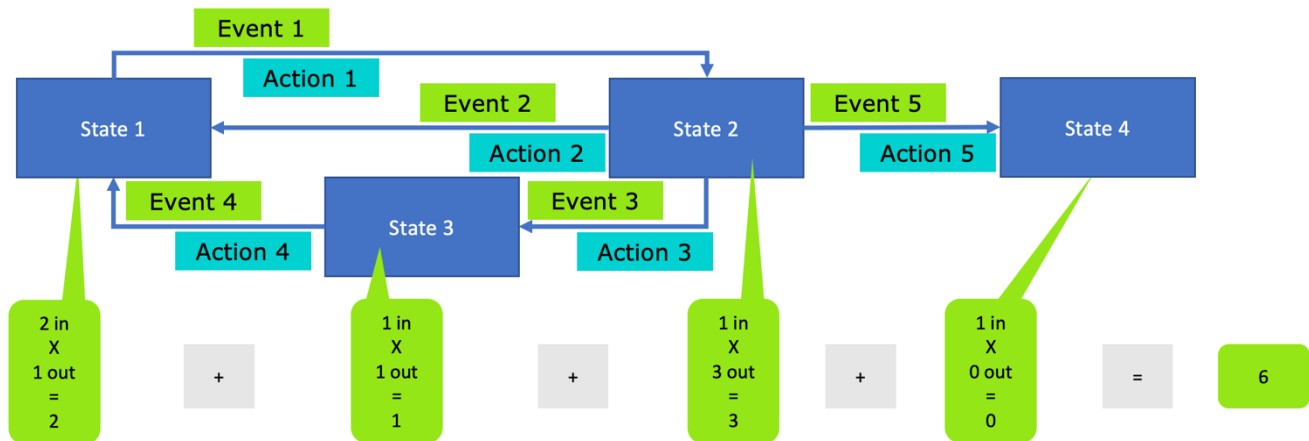
- For every state → multiply the number of in-coming transitions by the number of out-going transitions
- Add the results for all states
- This gives the number of test situations needed for 1-switch coverage.

To achieve 1-switch coverage, every test situation must be part of at least one test case.



Calculate test situations for a more complex example

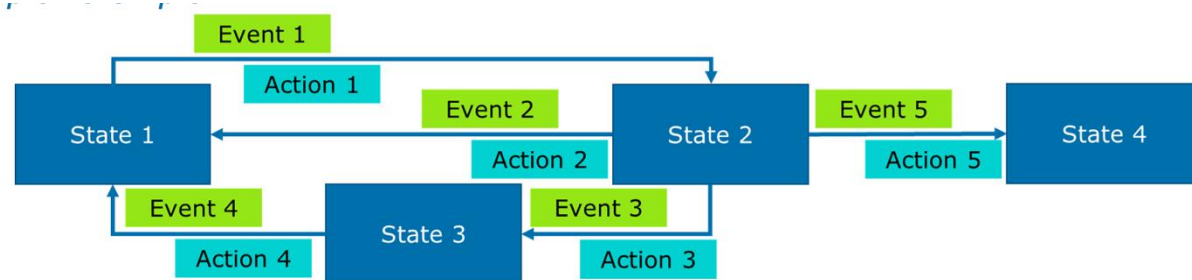
To illustrate the calculation of the number of test situations for 1-switch coverage, we also give a more complex system as an example.



For every state that has outgoing transitions, make all combinations of two consecutive transitions to create all test situations as shown below:

- Test situation 1: State 1 – Event 1 – State 2 – Event 2 – State 1
- Test situation 2: State 1 – Event 1 – State 2 – Event 3 – State 3
- Test situation 3: State 1 – Event 1 – State 2 – Event 5 – State 4
- Test situation 4: State 2 – Event 2 – State 1 – Event 1 – State 2
- Test situation 5: State 2 – Event 3 – State 3 – Event 4 – State 1
- Test situation 6: State 3 – Event 4 – State 1 – Event 1 – State 2

For this complex state transition diagram example, we also show how to create test cases based on the test situations.



As described above there are two approaches to creating test cases:

1. Create a test case for every individual test situation (“many short test cases”)
2. Combine as many test situations as possible in one test case (“a few long test cases”)

For this current example, both approaches can be used.

First, we use short test cases, one test case for each test situation):

- Test case 1: State 1 – Event 1 – State 2 – Event 2 – State 1
- Test case 2: State 1 – Event 1 – State 2 – Event 3 – State 3
- Test case 3: State 1 – Event 1 – State 2 – Event 5 – State 4
- Test case 4: State 2 – Event 2 – State 1 – Event 1 – State 2
- Test case 5: State 2 – Event 3 – State 3 – Event 4 – State 1
- Test case 6: State 3 – Event 4 – State 1 – Event 1 – State 2

Next, we combine as many as possible test situations, this example all test situations can be combined into just one long test case:

Test case 1: State 1 – Event 1 – State 2 – Event 2 – State 1 – Event 1 – State 2 – Event 3 – State 3 – Event 4 – State 1 – Event 1 – State 2 – Event 5 – State 4

Create physical test cases

To create physical test cases, for each logical test case describe the actions needed to position the system in the starting state, then trigger each of the consecutive transitions and verify if the resulting state and the actions performed are according to the expected outcome.

Create test scenarios

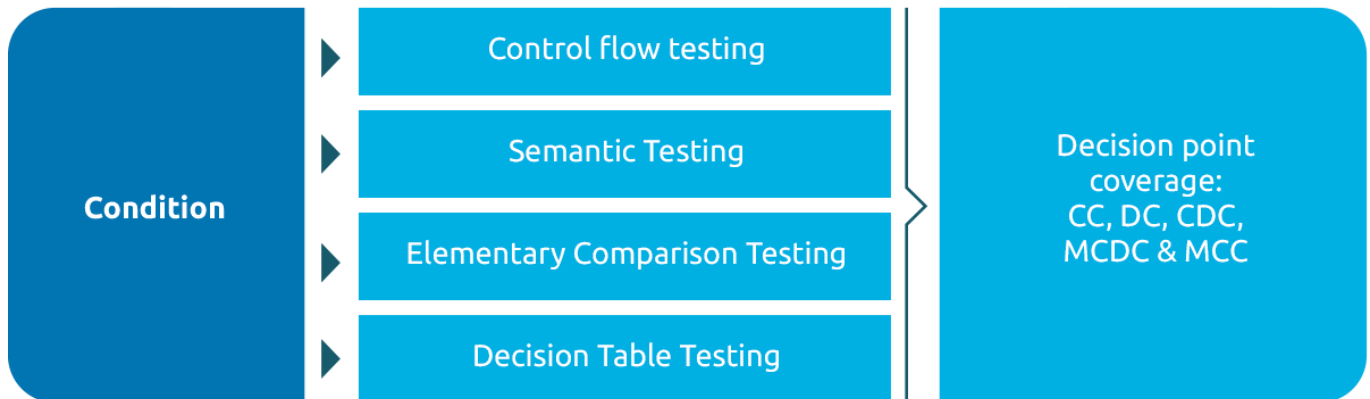
Usually every physical test case is a test scenario in itself so there will be little need to create specific test scenarios.

Note: A State Transition Testing template (excel) is available at www.TMAP.net.

7.7. Condition-oriented testing with MCDC

Modified Condition Decision Coverage (MCDC) makes sure every condition within a decision determines every possible outcome of that decision. This coverage type is a good combination of effectiveness (good coverage) and efficiency (not too many test cases).

This coverage type can be used in various test design techniques as shown in this picture:



MCDC is the standard coverage type for Semantic Testing and Elementary Comparison Testing.

Where Semantic testing focuses on testing individual decision points, Elementary comparison testing is used for testing functionality that consists of multiple decision points.

This text gives an overview of MCDC, Semantic and Elementary comparison testing. Additional in-depth information can be found on the TMAP body of knowledge (www.TMAP.net).

The following example will be used to explain MCDC, Semantic and Elementary comparison testing:

Example

Last year, QualityLand opened Agile Water Paradise. This is an outdoor water area that, besides a swimming pool, offers several attractions, such as: the Crazy Conditions River, the Magic Boat Ride, Shoot-the-chute and the Giant Waves Pool.

Since it is outdoor, this area is only open from May to September, with one exception: at Halloween (October 31st) this area is also open for a special event, the Super-Duper Masked Water Party (which was a huge success, last year).

At first, tickets for Agile Water Paradise could only be purchased as an addition to QualityLand tickets. However, market research learned that selling separate tickets for Agile Water Paradise would add great business value. Therefore, new functionality must be implemented to make this possible.

The functionality for the online purchase of separate tickets has been described as follows:

When ordering online, the customer first chooses a date for the visit to Agile Water Paradise. If a date is chosen on which Agile Water Paradise is closed, the following message should appear: "Unfortunately, Agile Water Paradise is closed on the date of your choice. Please choose another date."

If a valid date is chosen, the customer can enter the number of tickets, QualityLand discount card number (if applicable) or Student card number (if applicable). The customer can also check the box for attending the daily Summer Splash Party at the Giant Waves Pool for which an additional fee must be paid. This additional fee also applies to the Super-Duper Masked Water Party at Halloween. For the latter, the fee is automatically added when the chosen date is the date of Halloween.

The price per ticket is calculated as follows:

```

IF      (chosen date ≥ May AND chosen date ≤ September) OR chosen date = October 31st
THEN IF  (QualityLand discount card OR Student card) AND number of tickets > 4
      THEN price per ticket = € 15.00
      ELSE price per ticket = € 17.50
      ENDIF
      IF   chosen date = October 31st OR Summer Splash Party = Y
      THEN price := price + € 3.50
            IF   number of tickets ≥ 10
            THEN discount of 10%
            ENDIF
      ELSE no additional fee
      ENDIF
ELSE message: "Unfortunately, Agile Water Paradise is closed on the date of your choice. Please
choose another date."
ENDIF

```

7.7.1. Application of MCDC

Modified Condition Decision Coverage (MCDC) is a coverage type, from the coverage group Condition-oriented test design, that ensures that every possible outcome of a condition at least once is the determinant of the outcome of the decision.

MCDC implies condition coverage (CC), decision coverage (DC) and condition decision coverage (CDC).

The important concept in this definition is "determinant". If the outcome of the condition changes (from true to false or vice versa) then the outcome of the whole decision point changes with it.

If a decision point consists of the conditions A, B and C (or more), then MCDC guarantees:

- That there is at least 1 test situation in which the outcome is TRUE, owing to the fact that condition A is TRUE.
- That there is at least 1 test situation in which the outcome is FALSE, owing to the fact that condition A is FALSE.

In other words: changing *only* the value of A from TRUE to FALSE changes the outcome of the decision from TRUE to FALSE (and vice versa).

- The same applies to all other conditions in the decision point.

MCDC is a thorough level of coverage. The big advantage of this coverage type is its efficiency: if a decision point consists of N conditions, usually only N+1 test situations are required for MCDC. Compared with the maximum number of test situations (the complete decision table) of 2^N , that is a considerable reduction, particularly if N is large (complex decision points). This combination of "thorough coverage" with "relatively few test situations" makes this coverage type a powerful weapon in the tester's arsenal.

According to the definition of MCDC, every condition should determine the outcome of the decision at least once. Then all the other conditions in that situation should be given a value that does not influence the outcome of the decision. This value is called the "neutral value" and is explained below.

Neutral value explained

Take, for example, a decision (R), that consists of a combination of two conditions (A, B).

Let's first have a look at a decision for which the outcome of the decision is only TRUE if both conditions are TRUE. In other words:

$$R = A \text{ AND } B$$

What MCDC aims to achieve is defining test situations in such a way that the outcome of the decision changes by *only* changing the value of A. And then by *only* changing the value of B. (And if there were more conditions, this would apply to *every* condition.)

Note: TRUE is represented by 1 and FALSE is represented by 0.

Let's concentrate on condition A being the determinant of the outcome: we want to change the outcome of the decision by *only* changing the value of A.

Since we only want to change the value of A, we are looking for a 'neutral' value for B: the same value for B in every test situation in which A is the determinant.

$$\text{Test situation 1: } A = 0 \text{ AND } B = ? \rightarrow R = 0$$

If $A = 0$, for the outcome to be FALSE, B can either be 1 or 0.

$$\text{Test situation 2: } A = 1 \text{ AND } B = ? \rightarrow R = 1$$

If $A = 1$, for the outcome to be TRUE, B can only be 1.

Hence, the only value for B that we can use in *both* test situations is 1. So, if the operator is AND, the neutral value for B is 1.

Now what is the neutral value if the operator is OR? The decision now is only FALSE if both conditions are FALSE:

$$R = A \text{ OR } B$$

Since we only want to change the value of A, we are looking for a 'neutral' value for B: the same value for B in every test situation in which A is the determinant.

$$\text{Test situation 1: } A = 1 \text{ OR } B = ? \rightarrow R = 1$$

If $A = 1$, for the outcome to be TRUE, B can either be 1 or 0.

$$\text{Test situation 2: } A = 0 \text{ OR } B = ? \rightarrow R = 0$$

If $A = 0$, for the outcome to be FALSE, B can only be 0

Hence, the only value for B that we can use in *both* test situations is 0. If the operator is OR, the neutral value for B is 0.

Summarized:

- Neutral value with AND: 1
- Neutral value with OR: 0

6-step plan for deriving test situations with MCDC

To apply MCDC, every condition in the decision point must be the determinant once. If a condition is the determinant this means that changing the value of *only* that condition from TRUE to FALSE changes the outcome of the decision from TRUE to FALSE (and vice versa). Every other condition must then be given a neutral value.

There are 6 steps to follow when applying MCDC. With this 6-step plan a table is created that contains all the necessary test situations. First, the 6-step plan is explained below with a relatively simple example. After that, an example is given for more complex combinations of conditions.

Let's have a look at the third decision point from the example:

Example

```
IF      chosen date = October 31st OR Summer Splash Party = Y
THEN   price := price + € 3.50
ELSE   no additional fee
ENDIF
```

This decision point here is made up of 2 conditions with the structure: $R = A \text{ OR } B$.

The 6-step plan is set out below, resulting in the test situations with which this decision point is covered by MCDC.

Step 1, 2, 3 and 4

1. Create a table with 3 columns and fill in the first row
2. Add 1 row for every condition in the decision.

Every added row will contain the 2 test situations in which the relevant condition determines the outcome of the decision point. The condition will determine the outcome "1" once and the outcome "0" once.

In the first column, enter the description of every condition.

3. Fill in the rest of the cells in the table with a number of dots equal to the number of conditions in the decision. The first dot in each cell will represent the value of A (1 or 0), the second dot the value of B (and so on from left to right, in case of more than 2 conditions in the decision point).

Each cell becomes a test situation, that indicates which combination of TRUE/FALSE applies to the conditions.

4. Enter "1" diagonally in the second column and "0" in the third column.

This is actually entering the determining values for every condition. The meaning of for instance the cell that belongs to row "A" and column "1" is: "This is the test situation in which the value of condition A = 1 and that value determines an outcome of the decision of 1."

These first 4 steps result in:

R = A OR B	1 A B	0 A B
A: chosen date = October 31st	1 .	0 .
B: Summer Splash Party = Y	. 1	. 0

At www.tmap.net you can find an excel-template in which steps 1-4 have been applied already.

Step 5

At the remaining dots, enter the neutral value that goes with the operator.
In this case, A and B are connected through the operator OR. So for both, the neutral value is 0.

R = A OR B	1 A B	0 A B
A: chosen date = October 31st	1 0	0 0
B: Summer Splash Party = Y	0 1	0 0

Step 6

Score out duplicate test situations.

R = A OR B	1 A B	0 A B
A: chosen date = October 31st	1 0	0 0
B: Summer Splash Party = Y	0 1	0 0

The 6-step plan described above works for every composite decision point, however complex. With composite decision points in which both "AND" and "OR" occur, care should be taken at step 5 (entering the neutral values). The example below will explain this.

Take the second decision point from the example:

Example

IF (QualityLand discount card OR Student card) AND number of tickets > 4
THEN price per ticket = € 15.00
ELSE price per ticket = € 17.50

In short, this can be written as: $R = (A \text{ OR } B) \text{ AND } C$.

After the first 4 steps, the table of test situations looks as follows:

R = (A OR B) AND C	1 (€ 15.00) A B C	0 (€ 17.50) A B C
A: QualityLand discount card	1 . .	0 . .
B: Student card	. 1 .	. 0 .
C: Number of tickets > 4	. . 1	. . 0

Now the neutral values should be entered for each of the 6 situations. With the top 2 situations (A is the determining value), the neutral values can immediately be determined: B is connected to A via the operator "OR" and should therefore be given the neutral value "0". C is connected with A via the operator "AND" and should therefore be given the neutral value "1".

The same applies to the middle 2 situations (B is the determining value). However, for the bottom situations (C is the determining value) an interim step is necessary: it is not A that is directly connected with C, nor is it B. It is the combination "(A OR B)" that is connected with C, via the operator "AND". Thus "(A OR B)" should assume the neutral value of "AND", and that is "1". In other words: (A OR B) = 1. For the values for A and B there are 3 possibilities of achieving this, i.e. "1 1", "1 0" or "0 1". Only 1 of the 3 need to be selected to reach the goal of MCD C. In principle, it does not matter which. The only difference in the 3 possibilities is that in selecting "1 0" or "0 1" a test situation can be scored off, while that is not possible with the choice of "1 1".

If the neutral value of "0 1" is selected for (A OR B), then after the 6 steps, the table looks as follows:

R = (A OR B) AND C	1 (€ 15.00) A B C	0 (€ 17.50) A B C
A: QualityLand discount card	1 0 1	0 0 1
B: Student card	0 1 1	0 0 1
C: Number of tickets > 4	0 1 1	0 1 0

This phenomenon, that several possibilities exist for neutral values, always occurs in cases of an operator between brackets, and if it's an OR you can select 1 0, 0 1 and 1 1.

The number of possible selections differs if the decision point is like this (the AND and OR have been swapped):

R = (A AND B) OR C	1 (TRUE) A B C	0 (FALSE) A B C
A: condition A	1 1 0	0 1 0
B: condition B	1 1 0	1 0 0
C: condition C	1 0 1	1 0 0

Now the values for A AND B in the last line can be 1 0, 0 1 or 0 0. But choosing 0 0 would result in an extra test situation. So in reality 1 0 and 0 1 are the best choices. And 1 0 is the easiest choice because this is just filling in the neutral values for AND (1) and OR (0).

7.7.2. Semantic Test

The semantic test is a test design technique for testing the validity of data input using semantic rules for the *relationships* of the data on the input device and other data, for example in the database, in the system or on the input device.

Note: The semantic test is often executed in combination with the syntactic test.

Test approach:	Coverage-based – condition oriented
Test variety:	Validation test, Functional testing, Security testing
Test basis:	The test basis consists of the semantic rules, being single decision points, that specify what a data item should comply with – in relation to other data – in order to be accepted by the system as valid input. Semantic rules may be established in various documents, but are usually described in: <ul style="list-style-type: none"> ▪ Functional specifications of the relevant function or input screen ▪ The business rules that apply to the functions overall
Coverage type:	Modified condition decision coverage (MCDC) (other coverage types possible, see below)

The coverage type used is modified condition decision coverage (MCDC). But other coverage types can be applied to the Semantic test as well. For instance:

- Decision points: multiple condition coverage (MCC)
With this, the possibilities within the decision point can be tested even more thoroughly.
- Boundary Value Analysis
With this, the possibilities within the decision point can be tested even more thoroughly.
- Decision points: condition decision coverage (CDC)
For a lighter variant.

With the semantic test, user-friendliness aspects can be tested as well, by assessing the messages that occur in invalid situations thus:

- Are they understandable and unambiguous?
- Do they offer clear indications of how the invalid situation can be resolved?

Below, the Semantic test is explained step by step on the basis of 2 of the generic steps of test design:

1. Identifying the test situations
2. Creating logical test cases

Setting up a semantic test is very straightforward: each semantic rule is tested separately. Each rule leads to one or more test situations and each test situation generally leads to one test case.

To explain the Semantic test, we have a look at the example again:

Example

If the customer enters a QualityLand Discount Card number or a Student Card number, an extra check takes place. The customer needs to enter the 'valid thru' date of the card to check if the card is still valid on the chosen date of visit. Furthermore, it is checked if the entered card number exists.

```

IF    'valid thru' date ≥ chosen date AND (Discount card number exists OR Student card number exists)
THEN  message: "Card accepted"
ELSE  message: "Card not accepted"
ENDIF

```

1 – Identifying test situations

A semantic rule is a decision point that consists of one or more conditions connected by AND and/or OR. The test situations are derived by applying modified condition decision coverage (MCDC).

A AND (B OR C)	1 "Card accepted" A B C	0 "Card not accepted" A B C
A: 'valid thru' date ≥ chosen date	1 0 1 (1) (or 1 1 0, or 1 1 1, but 1 1 1 gives a logical contradiction ²)	0 0 1 (3) (or 0 1 0, or 0 1 1, but 0 1 1 gives a logical contradiction)
B: Discount card number exists	1 1 0 (2)	1 0 0 (4)
C: Student card number exists	1 0 1	1 0 0

2 – Creating logical test cases

The test situations from step 1 are directly the logical test cases:

Logical test cases				
	TC 1	TC2	TC3	TC4
'Valid thru' date	≥ chosen date	≥ chosen date	< chosen date	≥ chosen date
Discount Card number	n/a	exists	n/a	does not exist
Student card number	exists	n/a	exists	n/a
Predicted result:	Accepted	Accepted	Not accepted	Not accepted

² The combination B = true and C = true gives a logical contradiction: discount card number and student card number cannot be entered simultaneously. Therefore this combination may not occur in the test situations.

7.7.3. Elementary Comparison Test

The elementary comparison test (ECT) is a thorough technique for the detailed testing of functionality.

Test approach:	Coverage-based – condition oriented
Test variety:	Functional testing
Test basis:	Pseudo-code or a comparable specification in which the multiple decision points and functional paths are structurally worked out in detail
Coverage type:	Modified condition decision coverage (MCDC) (other coverage types possible, see below)

The ECT aims at thorough coverage of the decision points and not at combining functional paths. The coverage type used is modified condition decision coverage (MCDC) to achieve an optimum combination of effectiveness (good testing) and efficiency (limited number of test cases).

If there is a need for higher coverage, other coverage types can be applied with ECT as well. For example:

- Decision points: multiple condition coverage (MCC)
 - With this, the possibilities within the decision points (specifically selected, if necessary) can be tested even more thoroughly.
- Boundary Value Analysis
 - With this, the possibilities within the decision points (specifically selected, if necessary) can be tested even more thoroughly.

The ECT is a preferred technique for testing very important functions and / or complex calculations.

Below, the ECT is explained step by step on the basis of 2 of the generic steps of test design:

1. Identifying the test situations
2. Creating logical test cases

To explain the ECT, we have a look at the example again:

Example

The price per ticket is calculated as follows:

```

IF      (chosen date ≥ May AND chosen date ≤ September) OR chosen date = October 31st
THEN IF  (QualityLand discount card OR Student card) AND number of tickets > 4
      THEN price per ticket = € 15.00
      ELSE price per ticket = € 17.50
      ENDIF
      IF   chosen date = October 31st OR Summer Splash Party = Y
      THEN price := price + € 3.50
            IF   number of tickets ≥ 10
            THEN discount of 10%
            ENDIF
      ELSE no additional fee
      ENDIF
ELSE message: "Unfortunately, Agile Water Paradise is closed on the date of your choice. Please
choose another date."
ENDIF
  
```

1 – Identifying test situations

The test basis consists of pseudo-code or a comparable formal function description which can be copied directly in this step. If not, an extra activity should be carried out in order to convert the existing specifications into pseudo-code.

The decision points in the pseudo-code are provided with unique identification. It is usual to use the codes D1, D2, etc. for this (or D01, D02, etc. if there are many decision points).

Example

```

D1   IF      (chosen date ≥ May AND chosen date ≤ September) OR chosen date = October
      31st
D2   THEN IF  (QualityLand discount card OR Student card) AND number of tickets > 4
      THEN price per ticket = € 15.00
      ELSE price per ticket = € 17.50
      ENDIF
D3   IF   chosen date = October 31st OR Summer Splash Party = Y
      THEN price := price + € 3.50
D4   IF   number of tickets ≥ 10
      THEN discount of 10%
      ENDIF
      ELSE no additional fee
      ENDIF
      ELSE message: "Unfortunately, Agile Water Paradise is closed on the date of your choice.
Please choose another date."
ENDIF
  
```

Per decision point, modified condition decision coverage (MCDC) is applied in a separate table. The resulting test situations are numbered. The combination of this number and the decision point

provides a unique identification of the test situations (such as D1-1, D1-2, etc.). The numbering begins with the test situations in column "1" (true) and then those in the column "0" (false).

D1: (A AND B) OR C	1 (go to D2) A B C	0 (message & to end) A B C
A: chosen date ≥ May	1 1 0 (D1-1)	0 1 0 (D1-3)
B: chosen date ≤ September	1 1 0	1 0 0 (D1-4)
C: chosen date = October 31st	1 0 1 (D1-2) (or 0 1 1 , or 0 0 1 , but both give logical contradictions ³)	1 0 0

D2: (A OR B) AND C	1 (€ 15.00) A B C	0 (€ 17.50) A B C
A: QualityLand discount card	1 0 1 (D2-1)	0 0 1 (D2-3)
B: Student card	0 1 1 (D2-2)	0 0 1
C: number of tickets > 4	1 0 1	1 0 0 (D2-4) (or 0 1 0 , or 1 1 0)

D3: A OR B	1 (+ € 3.50 & to D4) A B	0 (to end) A B
A: chosen date = October 31st	1 0 (D3-1)	0 0 (D3-3)
B: Summer Splash Party = Y	0 1 (D3-2)	0 0

D4: A	1 (discount 10%) A	0 (no discount) A
A: number of tickets ≥ 10	1 (D4-1)	0 (D4-2)

³ In D1, the combination A = false and B = false gives a logical contradiction and therefore this combination may not occur in the test situations: date cannot be simultaneously lower than May and higher than September. Also, the combination B = true and C = true is not possible: date lower than September and date = October 31st is also not possible as a combination.

Detailed elaboration of the test situations:				
Test situations D1	D1-1	D1-2	D1-3	D1-4
chosen date	$\geq 05, \leq 09$	31-10	< 05	$> 09, \neq 31-10$
Test situations D2	D2-1	D2-2	D2-3	D2-4
QualityLand discount card	Y	N	N	Y
Student card	N	Y	N	N
number of tickets	> 4	> 4	> 4	≤ 4
Test situations D3	D3-1	D3-2	D3-3	
chosen date	31-10	$\geq 05, \leq 09$	$\geq 05, \leq 09$	
Summer Splash Party	N	Y	N	
Test situations D4	D4-1	D4-2		
number of tickets	≥ 10	< 10		

This elaboration is also helpful with identifying the mutual exclusive test situations (see next page).

Before these test situations are combined into logical test cases, two intermediate steps are taken:

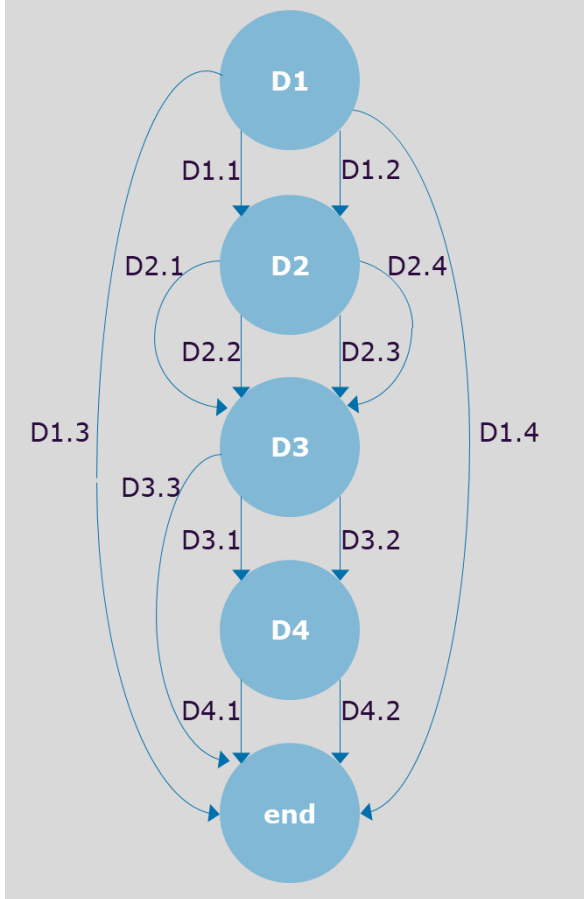
- Identify the possible 'chains' of test situations
- Identify mutually exclusive test situations

Identify the possible 'chains' of test situations using a graph

The creation of logical test cases can be made easier with the aid of a graphic representation of the possible 'chains' of test situations: a graph.

In this graph each decision point and the end point is represented by a circle and each test situation by a line that goes from one circle to another.

Graph for the example:



A logical test case runs through the graph from beginning to end, forming a chain of test situations. The graph also gives insight into the minimum number of test cases necessary to cover all the test situations. This is determined by the maximum number of parallel lines in the graph (in this example this is 6).

Identify mutually exclusive test situations

A logical test case should not contain "mutually exclusive test situations", for that makes the test case inconsistent and therefore inexecutable. Therefore it is wise to:

- Investigate which parameters (or closely related parameters) occur in more than one decision point, and (per parameter) identify which values are mutually exclusive. The detailed elaboration of the test situations (see previous page) is very helpful with this.
- Sum up the combinations of mutually exclusive test situations.

Mutually exclusive test situations in the example

The parameter "chosen date" occurs in the decision points D1 and D3. This leads to the following mutually exclusive test situations: D1-1 cannot be combined with D3-1; D1-2 cannot be combined with D3-2 and D3-3.

The parameter "number of tickets" occurs in the decision points D2 and D4. This leads to the following mutually exclusive test situations: D2-4 cannot be combined with D4-1.

2 – Creating logical test cases

A test case runs through the functionality from start to end and will come across one or more decision points on its path. With each decision point, the test case will test one of the defined test situations. Every test situation must be covered by at least 1 logical test case.

The logical test cases are created with the aid of a matrix. The rows contain the test situations and the columns contain the logical test cases. With each test case, it is indicated by one or more crosses which test situations should be tested by this test case. This matrix also serves as a check on the complete coverage of test situations.

The columns "Value" and "Next" have been added. These indicate for each test situation what the outcome of the decision is (directly obtainable from the tables in step 1) and to which subsequent decision point (or end of process) this leads (directly obtainable from the graph). This helps to prevent the tester from placing a cross at a test situation where the test case does not go.

Furthermore, mutually exclusive test situations have to be taken into account (this means you have to check whether some test situations cannot be combined in one test case).

Tip: the sooner you start with test situations that can't be combined with certain other test situations, the more not yet covered test situations are still available that they can be combined with.

Logical test cases								
Test situation	Value	Next	TC1	TC2	TC3	TC4	TC5	TC6
D1-1	1	D2	X				X	
D1-2	1	D2		X				X
D1-3	0	end			X			
D1-4	0	end				X		
D2-1	1	D3		X				
D2-2	1	D3					X	
D2-3	0	D3						X
D2-4	0	D3	X					
D3-1	1	D4		X				X
D3-2	1	D4	X					
D3-3	0	end					X	
D4-1	1	end		X				X
D4-2	0	end	X					
Predicted result:			21.00	16.65	msg	msg	15.00	18.90

A test case can be worked out in both a logical test case and a physical test case. In the example above the result is described as logical test cases.

A tester may execute a test based on the logical test case. However, this requires that the tester finds or creates test data at the time of test execution. Normally, executing based on logical test cases only works well if the execution is done by the same person that created the logical test cases. If someone else, or in case of test automation a machine, is executing the tests, each logical test case must be elaborated into a physical testcase.

Logical test case worked out as a Physical test case – example

For instance logical test case 2 (D1-2, D2-1, D3-1, D4-1) can be worked out as a physical test case as follows:

- Chosen date: October 31st
 - QualityLand discount card: yes
 - Student card: no
 - Number of tickets: 10
 - Summer Splash Party: no
- Predicted result: (€ 15.00 + € 3,50) -/- 10% = € 16.65

This syllabus is maintained by the members of the TMAP Special Interest Group and the Sogeti Academy. You can contact the Sogeti Academy in the Netherlands at academy.nl@sogeti.nl.

About Sogeti

Part of the Capgemini Group, Sogeti operates in more than 100 locations globally. Working closely with clients and partners to take full advantage of the opportunities of technology, Sogeti combines agility and speed of implementation to tailor innovative future-focused solutions in Digital Assurance and Testing, Cloud and Cybersecurity, all fueled by AI and automation. With its hands-on 'value in the making' approach and passion for technology, Sogeti helps organizations implement their digital journeys at speed.

A global leader in consulting, technology services and digital transformation, Capgemini is at the forefront of innovation to address the entire breadth of clients' opportunities in the evolving world of cloud, digital and platforms. Building on its strong 50-year heritage and deep industry-specific expertise, Capgemini enables organizations to realize their business ambitions through an array of services from strategy to operations. Capgemini is driven by the conviction that the business value of technology comes from and through people. It is a multicultural company of almost 220,000 team members in more than 40 countries. The Group reported 2019 global revenues of EUR 14.1 billion.

Visit us at www.sogeti.com

This document contains information that may be privileged or confidential and is the property of the Sogeti Group.
Copyright © 2022 Sogeti.

