International Standard

ISO/IEC/IEEE 29119-5

Software and systems engineering — Software testing —

Part 5: Keyword-driven testing

Ingénierie du logiciel et des systèmes — Essais du logiciel —

Partie 5: Essais axés sur des mots-clés

Second edition
2024-12

Reference number
ISO/IEC/IEEE 29119-5:2024(en)

COPYRIGHT PROTECTED DOCUMENT

ISO copyright office
CP 401 • Ch. de Blandonnet 8
CH-1214 Vernier, Geneva
Phone: +41 22 749 01 11
Email: copyright@iso.org
Website: www.iso.org

Institute of Electrical and Electronics Engineers, Inc
3 Park Avenue, New York
NY 10016-5997, USA

Email: stds.ipr@ieee.org
Website: www.ieee.org

Published in Switzerland

Contents

Page

Foreword ..... **v**

Introduction ..... **vi**

1 Scope **1**

2 Normative references **1**

3 Terms and definitions **1**

4 Conformance **5**
4.1 Intended usage 5
4.2 Full conformance 5
4.3 Tailored conformance 5

5 Introduction to keyword-driven testing **5**
5.1 Overview 5
5.2 Layers in keyword-driven testing 8
5.2.1 Overview 8
5.2.2 Domain layer 9
5.2.3 Test interface layer 10
5.2.4 Multiple layers 10
5.3 Types of keywords 11
5.3.1 Simple keywords 11
5.3.2 Composite keywords 12
5.3.3 Navigation/interaction (input) and verification (output) 15
5.3.4 Keywords that determine test results 15
5.4 Keywords and data 16

6 Application of keyword-driven testing **17**
6.1 Overview 17
6.2 Identifying and documenting keywords 17
6.3 Composing test cases 18
6.4 Keywords and data-driven testing 19
6.5 Modularity and refactoring 19
6.6 Keyword-driven testing in the test design and implementation process 20
6.6.1 Overview 20
6.6.2 Create test model (TD1) 21
6.6.3 Identify test coverage items (TD2) 21
6.6.4 Derive test cases (TD3) 21
6.6.5 Create test procedures (TD4) 22
6.7 Converting non-keyword-driven test cases into keyword-driven test cases 22

7 Keyword-driven testing frameworks **23**
7.1 Overview 23
7.2 Components of a keyword-driven testing framework 23
7.2.1 Overview 23
7.2.2 Keyword-driven editor 25
7.2.3 Decomposer 25
7.2.4 Data sequencer 26
7.2.5 Manual test assistant 26
7.2.6 Tool bridge 26
7.2.7 Test execution environment and test execution engine 26
7.2.8 Keyword library 26
7.2.9 Data 27
7.2.10 Script repository 27
7.2.11 Documentation and organizational processes 27
7.3 Basic attributes of the keyword-driven testing framework 27
7.3.1 General information on basic attributes 27
7.3.2 Documentation 27

7.3.3 Keyword-driven editor (tool) 28
7.3.4 Decomposer and data sequencer 29
7.3.5 Manual test assistant (tool) 29
7.3.6 Tool bridge 29
7.3.7 Test execution engine 29
7.3.8 Keyword library 30
7.3.9 Script repository 30
7.4 Advanced attributes of frameworks 30
7.4.1 General information on advanced attributes 30
7.4.2 Documentation 30
7.4.3 Keyword-driven editor (tool) 31
7.4.4 Decomposer and data sequencer 31
7.4.5 Manual test assistant 31
7.4.6 Tool bridge 31
7.4.7 Test execution environment and test execution engine 31
7.4.8 Keyword library 32
7.4.9 Test data support 33
7.4.10 Script repository 33
8 Data interchange **33**
Annex A (informative) **Typical keyword conventions** **34**
Annex B (informative) **Benefits and issues of keyword-driven testing** **35**
Annex C (informative) **Getting started with keyword-driven testing** **37**
Annex D (informative) **Roles and tasks** **39**
Annex E (informative) **Basic keywords** **41**
Annex F (informative) **Examples of the application of keywords** **47**
Annex G (informative) **Example data format for keywords** **51**
Bibliography **55**
IEEE notices and abstract **56**

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work.

The procedures used to develop this document and those intended for its further maintenance are described in the ISO/IEC Directives, Part 1. In particular, the different approval criteria needed for the different types of document should be noted. This document was drafted in accordance with the editorial rules of the ISO/IEC Directives, Part 2 (see www.iso.org/directives or www.iec.ch/members_experts/refdocs).

IEEE Standards documents are developed within the IEEE Societies and the Standards Coordinating Committees of the IEEE Standards Association (IEEE-SA) Standards Board. The IEEE develops its standards through a consensus development process, approved by the American National Standards Institute, which brings together volunteers representing varied viewpoints and interests to achieve the final product. Volunteers are not necessarily members of the Institute and serve without compensation. While the IEEE administers the process and establishes rules to promote fairness in the consensus development process, the IEEE does not independently evaluate, test, or verify the accuracy of any of the information contained in its standards.

ISO and IEC draw attention to the possibility that the implementation of this document may involve the use of (a) patent(s). ISO and IEC take no position concerning the evidence, validity or applicability of any claimed patent rights in respect thereof. As of the date of publication of this document, ISO and IEC had not received notice of (a) patent(s) which may be required to implement this document. However, implementers are cautioned that this may not represent the latest information, which may be obtained from the patent database available at www.iso.org/patents and https://patents.iec.ch. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

Any trade name used in this document is information given for the convenience of users and does not constitute an endorsement.

For an explanation of the voluntary nature of standards, the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the World Trade Organization (WTO) principles in the Technical Barriers to Trade (TBT), see www.iso.org/iso/foreword.html. In the IEC, see www.iec.ch/understanding-standards.

This document was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 7, *Software and systems engineering*, in cooperation with the Systems and Software Engineering Standards Committee of the IEEE Computer Society, under the Partner Standards Development Organization cooperation agreement between ISO and IEEE.

This second edition cancels and replaces the first edition (ISO/IEC/IEEE 29119-5:2016), which has been technically revised.

The main changes are as follows:

— updated reference to processes (test design and implementation process) according to ISO/IEC/IEEE 29119-2:2021;

— updated reference to documents (test procedure) according to ISO/IEC/IEEE 29119-3:2021;

— included definitions for terms used in this document, based on ISO/IEC/IEEE 29119-1:2022.

A list of all parts in the ISO/IEC/IEEE 29119 series can be found on the ISO and IEC websites.

Any feedback or questions on this document should be directed to the user's national standards body. A complete listing of these bodies can be found at www.iso.org/members.html and www.iec.ch/national-committees.

Introduction

The purpose of the ISO/IEC/IEEE 29119 series is to define an internationally agreed set of standards for software testing that can be used by any organization when performing any form of software testing and using any life cycle.

This document defines a unified approach for describing test cases in a modular way, which assists with the creation of items like keyword-driven test specifications and test automation frameworks. The term "keyword" refers to the elements which are, once defined, used to compose test cases, such as with building blocks. This document explains the main concepts and application of keyword-driven testing. It also defines attributes of frameworks designed to support keyword-driven testing.

The concepts relating to software testing defined in ISO/IEC/IEEE 29119-1 are also applicable to this document.

The test process model on which the keyword-driven testing framework is based is defined in ISO/IEC/IEEE 29119-2. It comprises test process descriptions that define the software testing processes at the organizational level, test management level and dynamic test level. Supporting diagrams describing the processes are also provided in ISO/IEC/IEEE 29119-2. However, this document describes a specific implementation of the test design and implementation process of ISO/IEC/IEEE 29119-2 for application in keyword-driven testing, in particular in TD3 (6.6.4) and TD4 (6.6.5).

The templates and examples of test documentation as defined in ISO/IEC/IEEE 29119-3 are also applicable to this document.

Software test design techniques that can be used during test design are defined in ISO/IEC/IEEE 29119-4. The application of ISO/IEC/IEEE 29119-4 is assumed when designing test cases that are then described by keywords according to this document.

Software and systems engineering — Software testing —

Part 5: Keyword-driven testing

1 Scope

This document defines an efficient and consistent solution for keyword-driven testing by:

— giving an introduction to keyword-driven testing;

— providing a reference approach to implement keyword-driven testing;

— defining requirements on frameworks for keyword-driven testing to enable testers to share their work items, such as test cases, test data, keywords, or complete test specifications;

— defining requirements for tools that support keyword-driven testing; these requirements are applicable to any tool that supports the keyword-driven approach (e.g. test automation, test design and test management tools);

— defining interfaces and a common data exchange format to ensure that tools from different vendors can exchange their data (e.g. test cases, test data and test results);

— defining levels of hierarchical keywords, and advising use of hierarchical keywords; this includes describing specific types of keywords (e.g. keywords for navigation or for checking a value) and when to use "flat" structured keywords;

— providing an initial list of example generic technical (low-level) keywords, such as "inputData" or "checkValue"; these keywords can be used to specify test cases on a technical level and can be combined to create business-level keywords as required.

This document is applicable to all those who want to create keyword-driven test specifications, create corresponding frameworks, or build test automation based on keywords.

2 Normative references

There are no normative references in this document.

3 Terms and definitions

For the purposes of this document, the following terms and definitions apply.

ISO, IEC and IEEE maintain terminology databases for use in standardization at the following addresses:

— ISO Online browsing platform: available at https://www.iso.org/obp

— IEC Electropedia: available at https://www.electropedia.org/

— IEEE Standards Dictionary Online: available at https://ieeexplore.ieee.org/xpls/dictionary.jsp

NOTE For additional terms and definitions in the field of systems and software engineering, see ISO/IEC/IEEE 24765, which is published periodically as a “snapshot” of the SEVOCAB (Systems and software engineering vocabulary) database and is publicly accessible at https://www.computer.org/sevocab.

3.1
actual result
set of behaviours or conditions of a *test item* (3.23), or set of conditions of associated data or the *test environment* (3.17), observed as a result of *test execution* (3.18)

EXAMPLE Outputs to screen, outputs to hardware, changes to data, reports, and communication messages sent.

[SOURCE: ISO/IEC/IEEE 29119-1:2022, 3.5]

3.2
composite keyword
keyword (3.8) that comprises two or more other keywords

3.3
data sequencer
component that replaces *keyword* (3.8) parameters with data, if necessary, across several hierarchy levels

3.4
decomposer
component that deconstructs the *high-level keywords* (3.7) across all levels into a sequence of *low-level keywords* (3.14) that eventually comprise it

3.5
domain layer
highest level of abstraction for the *test item* (3.23)

Note 1 to entry: *Keywords* (3.8) on this level are chosen in a way that is familiar to domain experts.

3.6
expected result
observable predicted behaviour of the *test item* (3.23) under specified conditions based on its specification or another source

[SOURCE: ISO/IEC/IEEE 29119-1:2022, 3.35]

3.7
high-level keyword
keyword (3.8) that covers complex activities that can be composed from other keywords and is used by domain experts to assemble *keyword test cases* (3.13)

3.8
keyword
one or more words used as a reference to a specific set of actions intended to be performed during the execution of one or more *test cases* (3.16)

Note 1 to entry: The actions include interactions with the user interface during the test, verification, and specific actions to set up a test scenario.

Note 2 to entry: Keywords are named using at least one verb.

Note 3 to entry: *Composite keywords* (3.2) can be constructed based on other keywords.

Note 4 to entry: Keywords often have parameters so they can be used with different data.

3.9
keyword library
keyword dictionary
repository containing a set of *keywords* (3.8) reflecting the language and level of abstraction used to write *test cases* (3.16)

3.10
keyword-driven testing
testing (3.27) using *test cases* (3.16) composed from *keywords* (3.8)

3.11
keyword-driven testing framework
test framework (3.20) covering the functional capabilities of a keyword-driven editor, *decomposer* (3.4), *data sequencer* (3.3), *keyword library* (3.9), data repository, *test environment* (3.17), and including organizational processes that define their interaction and use

Note 1 to entry: For manual *keyword-driven testing* (3.10) the functional capabilities of a manual test assistant are also covered, while for automated keyword-driven testing the functional capabilities of a *test execution engine* (3.19), *tool bridge* (3.28) and script repository are also covered.

3.12
keyword execution code
implementation of a *keyword* (3.8) that is intended to be executed by a *test execution engine* (3.19)

3.13
keyword test case
sequence of *keywords* (3.8) and the required values for their associated parameters (as applicable) that are composed to describe the actions of a *test case* (3.16)

3.14
low-level keyword
keyword (3.8) that covers only one or very few simple actions and is not composed from other keywords

3.15
manual testing
humans performing tests by entering information into a *test item* (3.23) and verifying the results

Note 1 to entry: Automated testing uses tools, robots, and other *test execution engines* (3.19) to perform tests. Manual testing does not use these items.

[SOURCE: ISO/IEC/IEEE 29119-1:2022, 3.46]

3.16
test case
set of preconditions, inputs and *expected results* (3.6), developed to drive the execution of a *test item* (3.23) to meet *test objectives* (3.24)

[SOURCE: ISO/IEC/IEEE 29119-1:2022, 3.85, modified — Notes 1 to 3 to entry have been removed.]

3.17
test environment
environment containing facilities, hardware, software, firmware, procedures, needed to conduct a test

Note 1 to entry: A test environment can contain multiple environments to accommodate specific test level or test types (e.g. a unit test environment, a performance test environment).

Note 2 to entry: A test environment can comprise several interconnected systems or virtual environments.

[SOURCE: ISO/IEC/IEEE 29119-1:2022, 3.95]

3.18
test execution
process of running a test on the *test item* (3.23), producing *actual results* (3.1)

[SOURCE: ISO/IEC/IEEE 29119-1:2022, 3.99]

3.19
test execution engine
tool that executes *test cases* (3.16) by passing data to the *test item* (3.23), triggering the test item to run and receiving data from the test item

Note 1 to entry: A typical test execution engine can be a part of a *test framework* (3.20), a capture and playback tool or a hardware robot.

3.20
test framework
structured set of principles, guidelines, and practices used for organizing, selecting and communicating *testing* (3.27)

[SOURCE: ISO/IEC/IEEE 29119-1:2022, 3.103]

3.21
test interface
interface to the *test item* (3.23) used to either stimulate the test item, to get responses [e.g. *actual results* (3.1)], or both

Note 1 to entry: The graphical user interface (GUI), application programming interface (API) or service-oriented architecture (SOA) interfaces are typical test interfaces.

Note 2 to entry: Stimulating the test item can involve passing data into it via computer interfaces or attached hardware.

Note 3 to entry: Getting responses includes getting information from the test item or associated hardware.

3.22
test interface layer
lowest level of abstraction for *keywords* (3.8), which interacts with the *test item* (3.23) directly and encapsulates the atomic (lowest level) interactions at the *test interface* (3.21)

3.23
test item
test object
work product to be tested

EXAMPLE Software component, system, requirements document, design specification, user guide.

[SOURCE: ISO/IEC/IEEE 29119-1:2022, 3.107]

3.24
test objective
reason for performing *testing* (3.27)

[SOURCE: ISO/IEC/IEEE 29119-1:2022, 3.114, modified — EXAMPLE has been removed.]

3.25
test procedure
sequence of *test cases* (3.16) in execution order, with any associated actions required to set up preconditions and perform wrap-up activities post execution

[SOURCE: ISO/IEC/IEEE 29119-1:2022, 3.120]

3.26
test result
indication of whether or not a specific *test case* (3.16) has passed or failed, i.e. if the *actual results* (3.1) correspond to the *expected results* (3.6) or if deviations were observed

[SOURCE: ISO/IEC/IEEE 29119-1:2022, 3.122]

3.27
testing
set of activities conducted to facilitate discovery and evaluation of properties of *test items* (3.23)

Note 1 to entry: Testing activities include planning, preparation, execution, reporting, and management activities, insofar as they are directed towards testing.

[SOURCE: ISO/IEC/IEEE 29119-1:2022, 3.131]

3.28
tool bridge
component that maps and transforms the *keywords* (3.8) to code and data to be executed by the *test execution engine* (3.19)

4 Conformance

4.1 Intended usage

The requirements in this document are contained in Clause 7.

This document provides requirements on the components of frameworks supporting the application of keyword-driven testing. It also provides requirements on conventions for the definition of keywords.

It is recognized that some organizations, projects, or teams may not need to use all of the components defined in this document. Therefore, implementation of this document typically involves selecting a set of components or parts of components suitable for the organization or project.

There are two ways that an implementation can be claimed to conform to this document – full conformance and tailored conformance. The organization shall assert whether it is claiming full or tailored conformance to this document.

4.2 Full conformance

Full conformance is achieved by providing evidence that all of the keyword-driven testing requirements defined in Clause 7 have been met.

4.3 Tailored conformance

When this document is used for implementing components of frameworks that do not qualify for full conformance, the subset of components for which tailored conformance is claimed should be recorded. Tailored conformance is achieved by demonstrating that all of the requirements for the recorded subset of components defined in Clause 7 have been satisfied.

Where tailoring occurs, the justification shall be provided, either directly or by reference, whenever a requirement defined in Clause 7 is not followed. All tailoring decisions shall be recorded with their rationale, including the consideration of any applicable risks. Tailoring decisions shall be agreed to by the relevant stakeholders.

EXAMPLE A tool vendor provides only part of a keyword-driven testing framework in its portfolio, and thus decides not to implement requirements that are covered by complementary tools (e.g. a vendor only provides an execution engine, but no keyword-driven editor – then the execution engine can still conform to this document).

5 Introduction to keyword-driven testing

5.1 Overview

Keyword-driven testing is a test case specification approach that is commonly used to support test automation and the development of test automation frameworks. However, it can also be used if no automation approach is planned or established.

In principle, keyword-driven testing can be applied at all testing levels (e.g. component testing, system testing) and all types of testing (e.g. functional testing, reliability testing). Keyword-driven testing benefits include the following:

— ease of use;

— understandability;

— maintainability;

— test information reuse;

— support of test automation;

— potential cost and schedule savings.

The fundamental idea in keyword-driven testing is to provide a set of "building blocks", referred to as keywords, that can be used to create manual or automated test cases without requiring detailed knowledge of programming or test tool expertise. The ultimate goal is to provide a basic, unambiguous set of keywords comprehensive enough so that most, if not all, required test cases can be entirely composed of these keywords. The vocabulary included in these dictionaries or libraries of keywords is, therefore, a reflection of the language and level of abstraction used to write the test cases, and not of any standard computer programming language.

For test automation, each keyword must be implemented in software.

NOTE 1 When keywords are not used, test cases are usually written using natural language or written in a computer programming language. Compared with natural language, keywords have the advantage of being less ambiguous and more precise. Compared with a computer programming language, when keywords are well-defined and well-structured, they have the advantage of being understandable by many people who do not have software engineering skills.

A keyword is usually defined at the following two levels.

— At a low level, each keyword is associated with a detailed set of one or more actions that describe the exact steps that are to be performed.

— At a high level, each keyword is ideally assigned a name which is meaningful to non-technical users. This keyword can require a set of input parameters, which would also belong to this level in the structure. The keyword and the parameters together comprise a high-level description of the actions associated with a test case.

Thus, instead of describing each individual action that needs to be taken in each test case, tests can be defined at a higher level of abstraction using keywords. Higher levels of abstraction can be achieved by using composite keywords, which are comprised of other keywords to describe associated actions.

An example of the benefits obtained from both manual and automated keyword-driven test cases is enhanced maintainability. Consider a case where the precise set of actions to carry out a commonly repeated operation has changed. The modularity introduced by keywords allows modification of only the actions for the changed operation in the relevant lower-level keyword, leaving the test cases untouched. Without modularity, it can be necessary to modify each occurrence of this operation in all of the test cases.

Modularization has helped popularize this approach. If test automation is required, a framework can be created to interpret manually created keyword test cases as executable test automation scripts. This is achieved by implementing test automation code for each keyword (e.g. keyword execution code).

NOTE 2 Testing tools can be used to support keyword-driven testing, but the available tools can be limited in their capability to support all the concepts described in this document.

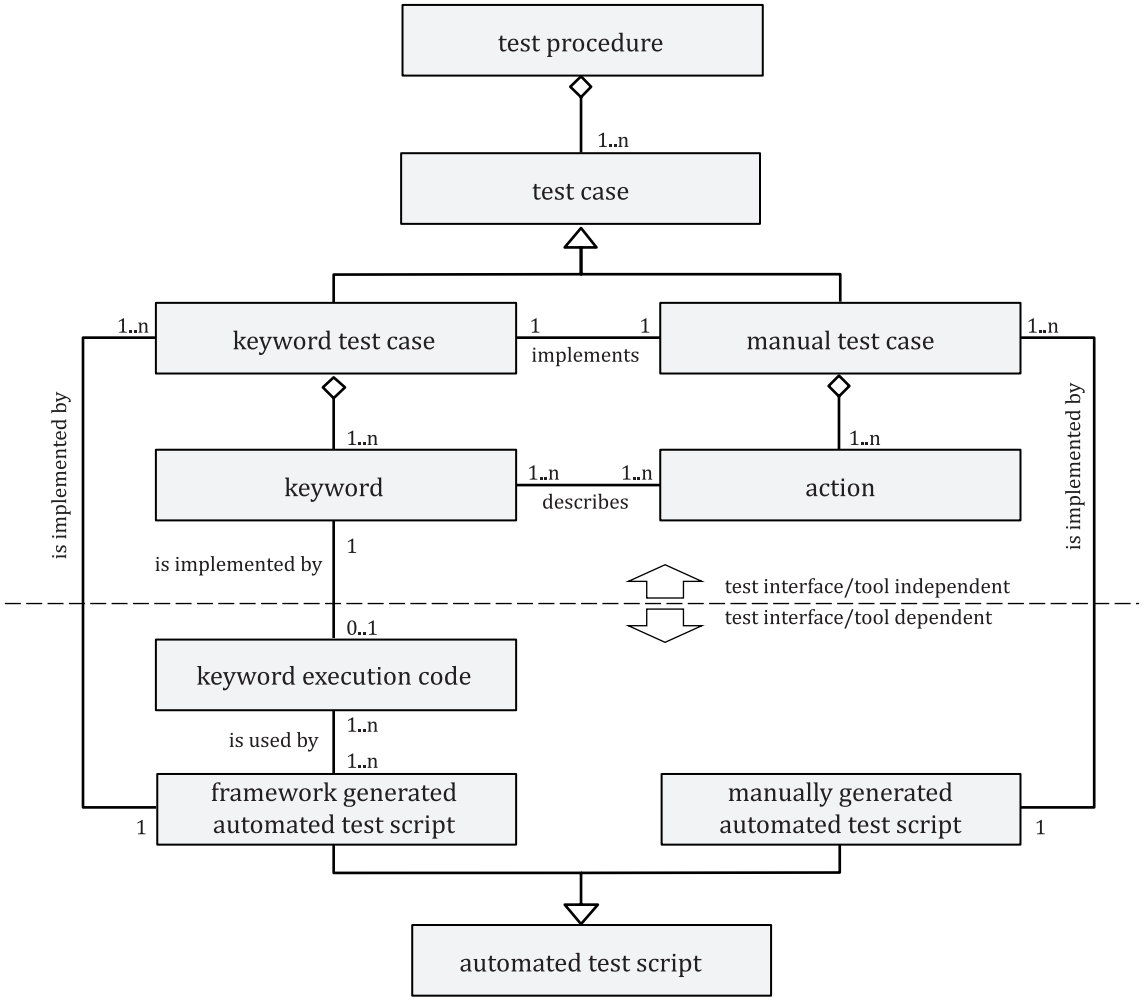


Figure 1 — Relationships between keyword-driven testing entities

A test procedure can have multiple test cases in it, and a test case can, in turn, be part of different test procedures as shown in Figure 1. Test cases can be either manual test cases or keyword test cases. A keyword test case implements a manual test case.

A keyword test case is typically composed of a sequenced series of keywords. Keywords should be chosen to be modular and generic so that they can be reused in many test cases. Keywords can also be used more than once in the same test case. A test case is composed from test actions. Keywords represent test actions.

NOTE 3 It is possible to map several keywords to a single action. It is also possible to define keywords in a way that each keyword represents one action. In these cases, a one-to-one relationship exists between actions and keywords. However, a test designer can decide to structure keywords in a different way (e.g. use more than one keyword to implement an action, or to combine two or more actions into one keyword). Thus, this relationship is not a 1:1 relationship in Figure 1.

Test automation is an option that can be chosen when implementing keyword-driven testing, but a manual approach is also possible. If keyword test cases are automated, each keyword is implemented by keyword execution code. Keyword execution code is specific to the chosen tool or test execution engine and additionally depends on the test interface. For the manual approach, the action described by a keyword is executed manually, so there is no keyword execution code. That is why in Figure 1 the relationship between keyword and keyword execution code is 1 to 0..1.

Test automation is typically highly technical and tool-dependent since it depends on the test interface and on the capabilities of the available tools. In general, keywords can be independent of the test interface (e.g. user interface) and the tools used to execute the test cases.

In this context, automated test scripts can either be generated automatically by a framework or developed manually by a test automation specialist. Automated test scripts are typically developed by testers with programming experience.

NOTE 4 When developing automated test scripts, it is beneficial to align the structure (e.g. levels) of the keywords with the implementation of the automated test scripts.

If a keyword test case or a set of keyword test cases is automated, the framework for keyword-driven testing generates the automated test script based on the keyword execution code.

NOTE 5 A framework for keyword-driven testing does not necessarily "generate" code. The required code can also be prepared by testers and be executed by the framework.

Annex B provides an overview of advantages and disadvantages of keyword-driven testing.

5.2 Layers in keyword-driven testing

5.2.1 Overview

Keywords can represent actions at different abstraction levels. For example, one keyword can refer to a very complex set of activities, like the creation of a contract, which includes a lot of steps, while another keyword can refer to a very simple action, like pressing a button on a graphical user interface. The first keyword is close to the business and end user domain, while the second is closer to the test interface. Keywords that are written at a similar level of detail, and have a similar relationship to the stakeholder's view, are said to belong to the same abstraction layer.

Keyword-driven testing can be organized by using one or more layers. Typical layers are the end user domain layer and the test interface layer.

While many implementations of keyword-driven testing comprise two or three abstraction layers, in some cases it can be necessary to structure keywords in more layers.

The topmost layer is the most abstract layer, which is generally aligned with the wording of the application's users. In practice, the topmost layer is usually the domain layer. However, in some situations the domain layer is not required, and another, more abstract layer is used (e.g. if the test cases are supposed to span several different end user domains, a meta domain layer can be introduced).

The lowest layer is the most detailed layer. It is commonly aligned with the names of test interface elements (e.g. "SelectMenu"). In practice, this layer is usually, but not always, the test interface layer (e.g. as sometimes a test interface layer is not required, or for specific reasons, even more detailed layers can be used).

Most keyword-driven test systems have more than one layer due to factors such as having understandable keyword test cases, maintainability and division of work relying on a multi-layer structure. If only one layer is implemented, it is commonly either at a low level, which affects the readability of the keyword test cases, or at a high level, which can result in more keyword execution code.

In Figure 2, an example is given, showing how test cases for two different test interfaces can be structured by using two layers of keywords.

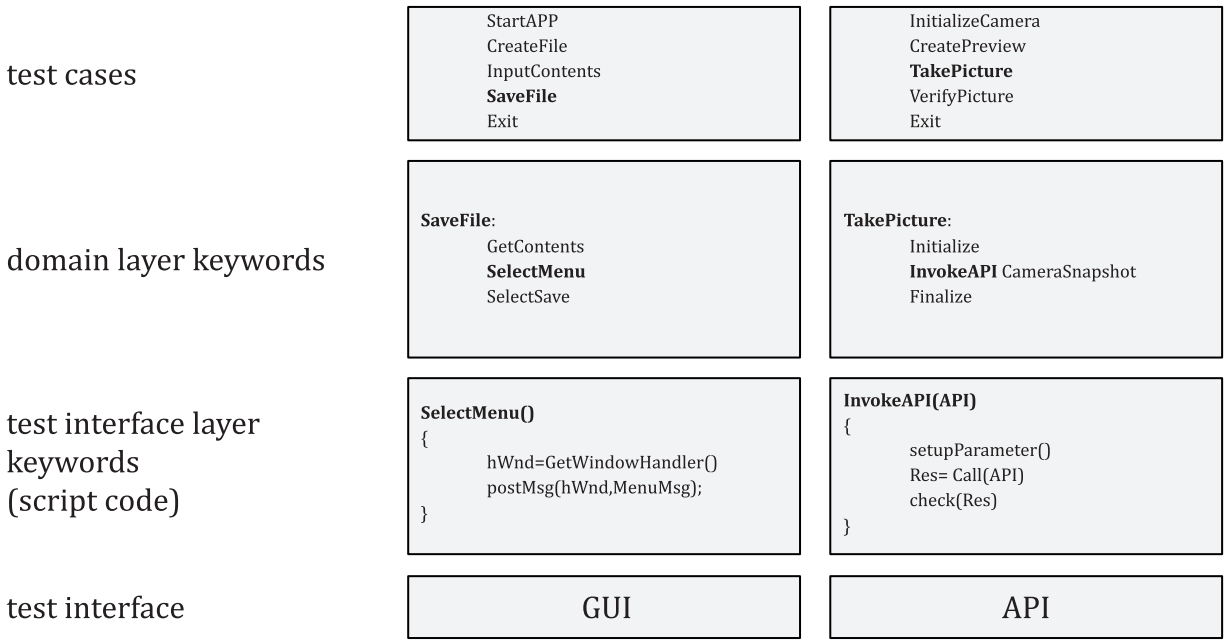


Figure 2 — Example for defining test cases by keywords at several layers

EXAMPLE In Figure 2, two test cases are shown which are designed using a domain layer and a test interface layer. One of the examples sketches a test case for a GUI application, the other for a camera API. In both examples, the implementation of the test cases in respect to test automation is done on the test interface layer. From top to bottom, the example shows a test case for each test item, provides a possible structure of the composite keywords used at the domain layer for both test items, and outlines the implementation of one of the simple keywords on the test interface layer for each test item.

5.2.2 Domain layer

Keywords in the domain layer correspond to business or domain related activities and reflect the terminology used by domain experts. Because of this, it can be easier for testers at the domain or business level to create test cases.

Keywords developed for the domain layer are generally implementation-independent; that is, the keywords define tests that work regardless of the technology used to implement the test item.

EXAMPLE 1 A keyword test is developed to test a word processing application. Domain layer keywords correspond to the activities that are part of the “business of word processing”:

StartApplication <app_name>

ClearBuffer

EnterText “Hello World!”

ReplaceText “Hello”, “Goodbye”, “ALL_OCCURRENCES”

VerifyText “Goodbye World!”

StopApplication

This test is valid for any text editor application that provides a global replace function (e.g. Notepad, MSWord, Notepad++, GED, EMACS).

EXAMPLE 2 Frequently used domain layer keywords are "Login" and "CreateAccount".

Tests constructed using domain layer keywords are relatively immune to changes in the implementation of the test item, and can prevent expensive rework over the lifetime of the test item.

NOTE Extensive changes to the application, (e.g. changes in the workflow) can require test cases to be reworked.

5.2.3 Test interface layer

Keywords at the test interface layer refer to a specific type of test interface (e.g. the GUI). The actions needed to address the test items can usually be easily identified. The total number of keywords is typically smaller than at the domain layer, since the test interface is limited.

EXAMPLE 1 A GUI can be used as the test interface. As the GUI controls (along with the associated actions) are mapped to a fixed set of keywords, a small number of keywords is needed. In the same case the domain-related keywords can be very versatile, and testers can decide to extend them according to their needs, which leads to a much bigger number of keywords.

If automation is desired, the keyword execution code for keywords at the test interface layer is often simpler. However, for a keyword test case composed from keywords at the test interface layer, it can be difficult to see how the interface layer keywords are related to the business domain.

Interface layer keywords usually reflect the underlying implementation technology for the interaction with the test item. For example, keywords such as MenuSelect and PressButton reflect a GUI operation. Using EXAMPLE 1 above, they would not be applicable to text editors using a command line interface, such as vi, because they correspond to window-based operations.

EXAMPLE 2 A graphical user interface is the test interface. Keywords are chosen to cover single actions such as "Click" or "Select". These keywords are applied to different elements like lists, grids, or images, and the specific element can be selected by using the keyword with a parameter (see 5.4). Some combinations of actions and elements can be excluded.

5.2.4 Multiple layers

To combine the advantages of several layers (e.g. domain layer and test interface layer), a framework is required, which can help manage hierarchical keywords (see Clause 7 for details about testing frameworks). This way a high-level keyword at the business level (e.g. domain layer keyword), can be built from several lower-level keywords at a more technical level (e.g. test interface layer keywords).

Figure 3 illustrates how multiple layers can be used in keyword-driven testing.

In complex settings, three or more layers of keywords are necessary. In the figure, additional intermediate layers are represented by three dots "…".

Using multiple layers requires composite keywords (see 5.3.2.).

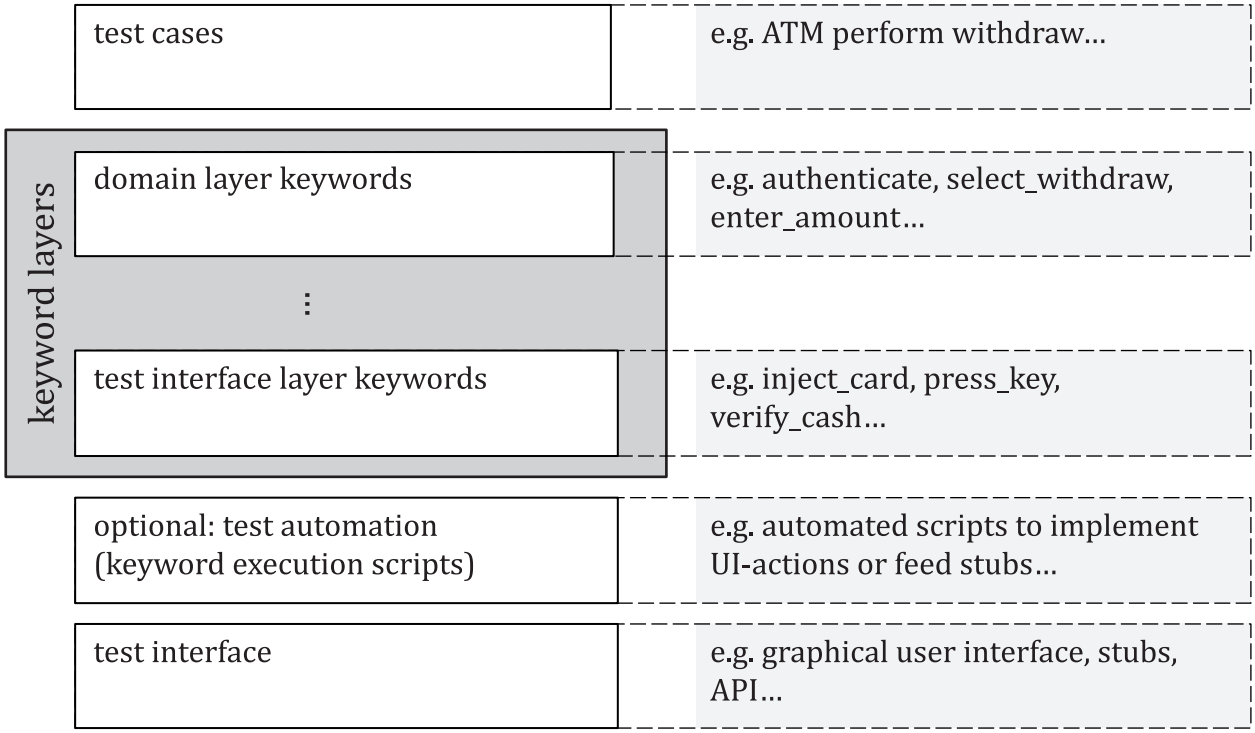


Figure 3 — Multiple layers in keyword-driven testing

NOTE 1 Figure 3 explicitly shows two keyword layers – a domain layer and a test interface layer – and indicates that in between there can be intermediate layers. It is possible, and can be sufficient, to organize keywords in only one layer. However, there can also be situations in which more than two layers are needed.

NOTE 2 In Figure 3, the domain layer keywords are taken from the domain of an ATM test and are meant to be used to create test cases. The keywords from the test interface layer refer to simple actions that can be applied to the test interface. The keyword "verify_cash" in this example is related to the test interface, and is supposed to cover only one small activity, and used as part of domain layer keywords. In another example, a different design is possible that covers several actions and is then part of the domain layer.

5.3 Types of keywords

5.3.1 Simple keywords

Simple keywords, which are not composed from other keywords and are often used at the test interface layer (e.g. "MenuSelect" or "PressButton"), can be the connection between the test execution tool and higher-level keywords at an intermediate layer or domain layer.

Using only keywords at the test interface layer can be sufficient for the definition of test cases and their execution. Exclusive use of simple keywords will lead to test cases with many actions.

Depending on the test item, keywords at the test interface layer possibly need to interact with different systems such as databases, the system registry, or SOA messages. This challenge is normally supported by the automation framework by providing a predefined set of keywords in order to make the technical environment as clear as possible.

In a similar way, the automation framework supports access to the test interface or other interfaces on which the keyword operates (e.g. mouse, keyboard, and touch screen).

Depending on the test interface, it can be possible to operate with a very limited number of simple keywords. A limited vocabulary of keywords is beneficial for composing test cases, since they are easier to remember, use and maintain. If test automation is required, a very limited number of keywords can result in an

increased effort to implement the keyword execution code. This is because if a small number of keywords still must cover the same complexity, an individual keyword must be more flexible or powerful.

EXAMPLE An implementation is structured by the required actions such as "select". That particular action is only implemented once due to an objective to have a small number of keywords. In this case, the single keyword "select" addresses several types of interface elements, such as for a GUI, lists, tables and radio buttons. The keyword "select" is for that reason associated with a complex implementation.

5.3.2 Composite keywords

Simple keywords are sufficient to compose and execute test cases but are often insufficient to reflect functional features.

Composite keywords are keywords composed from other keywords. This means that keywords can be organized in different layers (see 5.2). For composite keywords, composite parameters (e.g. a data structure) can be required.

It is often useful to use business-level keywords, such as "login user". This keyword can be composed of a sequence of lower-level keywords, such as "enter username", "enter password" and "push login-button". For more complex business objects, such as large forms for the preparation of contracts, a keyword like "filloutContractformPage1" can be valuable.

EXAMPLE 1 It is common to use composite keywords for verification, (e.g. retrieve a value from the application, compare it with an expected result, and log the result of the comparison in the test execution log).

It is also possible to define a keyword at a higher level (e.g. domain level) with a single keyword at a lower level (e.g. test interface level) to express a different semantic meaning.

EXAMPLE 2 For navigation purposes, a high-level keyword "GoToResultsScreen" is defined by the lower level keyword “Click ResultsButton”

It is also possible to combine several simple keywords to create a complex operation with a higher level of functionality, such as "CreateCustomerAccount", which can include a large number of basic steps.

A composite keyword is a ‘package’ containing a sequence of other keywords. The set of parameters for a composite keyword can be the union of the set of parameters of the keywords that comprise the composite keyword; sometimes however, the implementer of a composite keyword can choose to ‘hide’ one or more parameters by assigning it a literal value within the composite. This is done in the example in Figure 4. The interface layer keyword "Enter_value" has two parameters: the id of the referenced object and the value that is to be inserted. Only the value (e.g. username) is visible on the top layer keyword "login", while the id is hidden from a tester, who only used the composite keyword. This is especially useful if the detailed technical information is irrelevant to the person who designs test cases and operates at the domain layer.

EXAMPLE 3 Figure 4 illustrates how a keyword for a login procedure can be designed as a composite keyword in three layers. At the domain layer, this keyword can be used, [e.g. 'Login ("John","secret")']. This keyword is composed of three keywords at the intermediate layer, "Set_User", "Set_Pwd" and "Close_Login". "Set_User" and "Set_Pwd" both use one of the parameters of the higher layer keyword "Login", while the keyword "Close_Login" requires no parameters or data at all. At the third layer (the interface layer), the simple keywords "Set_context", "Enter_value" and "Select" are used. On this third layer, literal values are used, such as "Login_Window", which has not been provided with the domain layer keyword but will be used the same way every time one of the intermediate layer keywords is used.

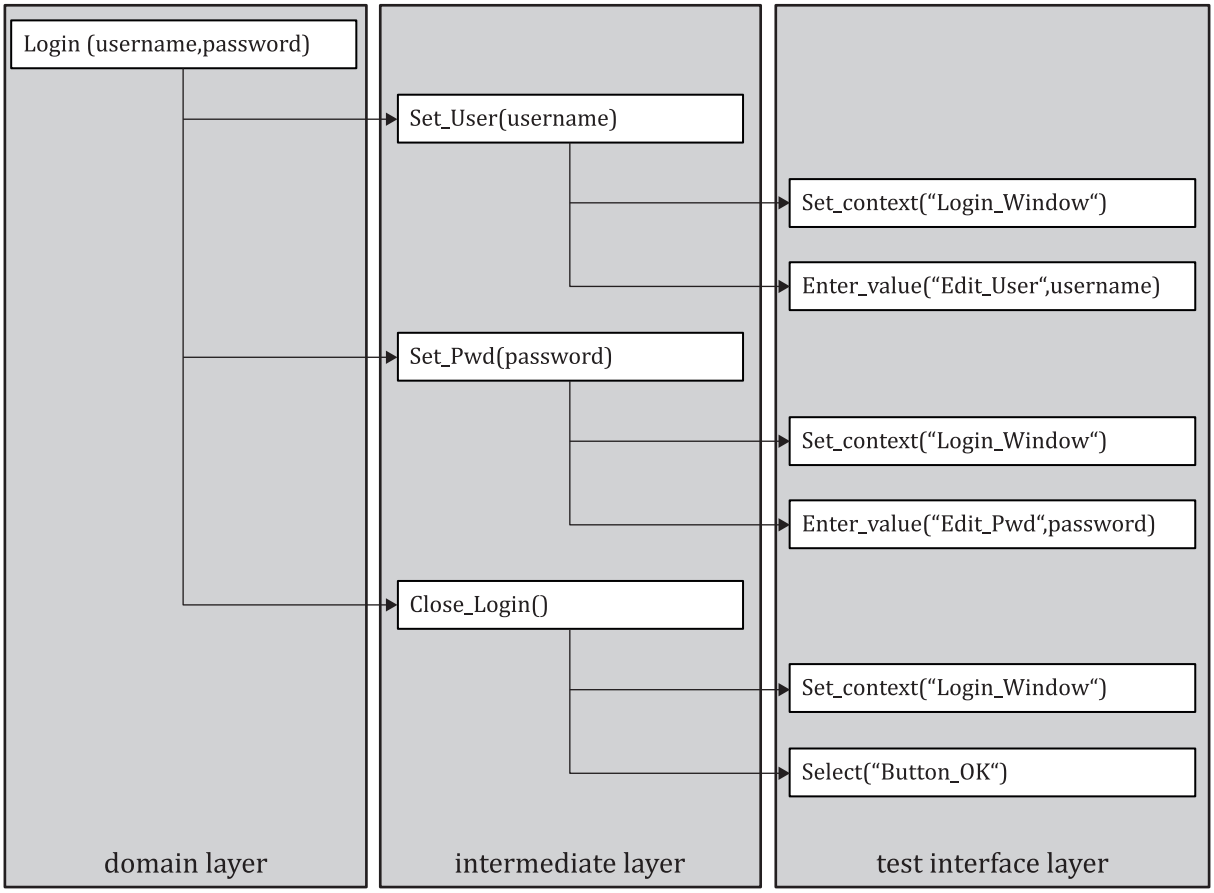


Figure 4 — Example for using composite keywords with data

Figure 5 explains the relationship between different types of keywords, keyword test cases and the level of keywords that are eventually applied to the test item. The keyword can be low-level, high-level or a composite keyword.

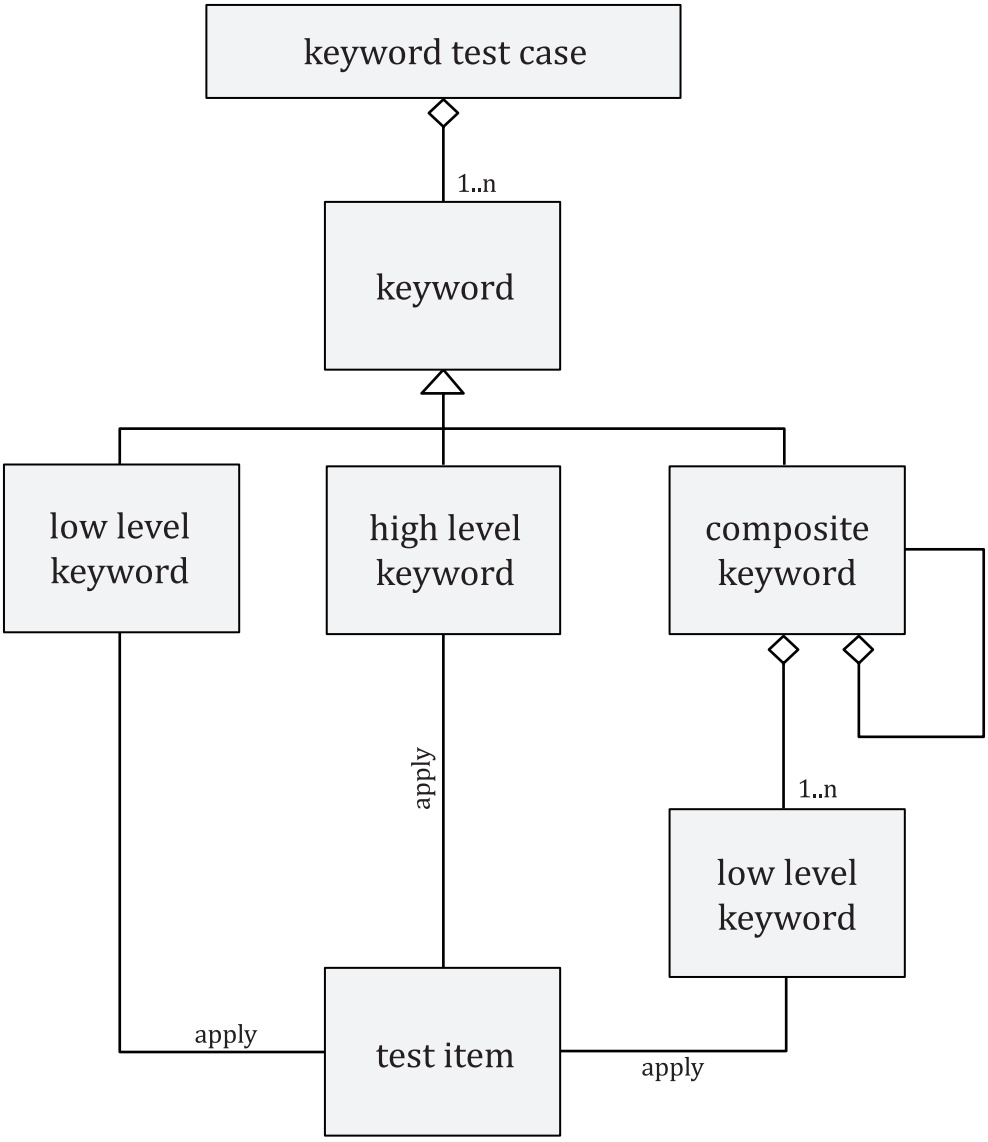


Figure 5 — Keyword test cases composed from keywords at different levels

If composite keywords are not used, keyword test cases can be built from low-level keywords, such as from the test interface layer. Through this approach, testing of the test item is accomplished by using low-level keywords.

NOTE 1 In Figure 5, the composite keyword can be either a low-level keyword or a high-level keyword.

Consequently, the test cases are understandable for human testers and machine-readable test execution engines. On the other hand, when reading such test cases, it can be hard to recognize the use case or business case addressed by the test case.

By exclusively using high-level keywords, such as business keywords or domain keywords, the derived keyword test cases are generally more understandable in respect to the addressed use cases or test cases. Testing of the test item is accomplished by using high-level keywords. Thus human testers need more information about the detailed steps needed to execute the more abstract keywords, especially if they are not familiar with the business domain. If test execution engines are used, these test execution engines need more information about the detailed steps needed to execute the more abstract keywords as well.

After combining low-level keywords to form composite keywords at a higher level (e.g. combining keywords from the test interface layers with composite keywords at the domain layer), the keyword test case can be

composed from these high-level keywords. Such test cases are very easy to understand, as they resemble the related use cases or business cases.

NOTE 2 Figure 5 shows, for composite keywords, two levels of keywords: composite high-level keywords, and low-level keywords. It is possible to have more levels, such as intermediate composite keywords, which are composed of lower level keywords and are used to compose higher-level keywords.

To execute the tests, the high-level keywords can be decomposed into low-level keywords, usually using a framework (see 7.2). So testing of the test item is accomplished by only using low-level keywords, which makes it easy for human testers or test execution engines to identify the necessary actions to perform the test since they have simple steps to follow.

Keywords from different layers should not be mixed and used in one keyword test case, as it can be the source of maintenance problems.

5.3.3 Navigation/interaction (input) and verification (output)

Keywords can be classified into at least two categories: navigation steps (i.e. input to the test item) and verification steps (i.e. output from test item).

Most keywords belong to the first category (i.e. the navigation steps) because most actions are needed to prepare the test item or perform certain actions on it which will lead to a result. Navigation steps usually are steps that do not verify and log the test result.

The result is then checked by one or more other actions i.e. the verification steps.

The verification steps are related to the result of the test case. For example, if the condition of a verification step is not met, then the test result is set to "failed".

It can be useful to allow navigation steps to be used for verification.

EXAMPLE 1 A navigation step "AddUser" is required to prepare data for a test case. In some cases, it can be used in a context where the addition of a user is supposed to succeed, in other cases it can be used in a situation where the addition is expected to fail. Thus, the keyword can verify whether it successfully creates a user, without marking the test case as "failed". However, the test designer can also decide to mark a test case as "failed" due to the failed execution of that navigation step, although the actual intent of the test case is to verify a result which appears later in the process.

See reasons for tests failing in 5.3.4.

Keywords are typically semantically independent from each other. Therefore, if a keyword is meant to trigger an expected result, the verification of this expected result is part of the same keyword and not in another keyword.

EXAMPLE 2 Pairs of keywords like "Open the dialog" – "Verify the dialog is opened" are normally avoided when the second keyword is exclusively used following the first one.

5.3.4 Keywords that determine test results

Keywords can be used to determine the test status and to capture test results. This can include the following:

— test output;

— conformance to success criteria;

— test execution log files;

— hardware outputs;

— system status;

— test failure(s).

There are different reasons why a test execution can fail that can include the following:

— the conducted checks in the test case reveal a mismatch between actual outputs and expected outputs, which can indicate a software failure;

— some steps in the test case cannot be executed, because test execution is blocked.

NOTE Blocked test cases include test cases that cannot be executed due to faults in keywords, keyword execution code or the test environment.

It is useful to recognize the cause of a failed execution at first glance without having to analyse the cause in detail. Thus the framework sets different test results (e.g. failed and/or blocked) accordingly.

The result of an individual keyword execution normally impacts the test result, but that impact depends on the context.

EXAMPLE A keyword is defined to enter text into an edit field. The keyword works the same but the results are interpreted differently depending on context. If the text field is expected to be active and text entered successfully then the test result is set to passed. Conversely, if no text is entered to an active field, the test result is set to failed. On the other hand, if the text field is expected to be inactive and text entered successfully the result is set to failed, whereas if text cannot be entered the test result is set to passed.

The test framework can be designed to handle blocked keywords on the test item. Keywords can then be optionally marked either as “may be blocked” or as “must not be blocked”. In the first case, a blocking (unsuccessful execution) of the keyword would not affect the test result; in the second case, the test result will be affected. A keyword can be marked either globally (the property is default for all applications in test cases) or overridden when it is used in a test case.

The test framework can additionally provide an error recognition mechanism that can take care of errors returned by a keyword. Failures can be logged and described as clearly as possible in order to simplify the correction of errors in the automation framework and investigate its cause, which can be a software-defect.

5.4 Keywords and data

Keyword-driven testing can be enhanced if keywords are associated with data. To allow an association with data, in many cases keywords have parameters which can be fixed or list-driven.

Most keywords have at least one parameter to specify the object they apply to. Some have another parameter to specify input, (e.g. true/false, a string to type, an option to select in a combo box). This input generally depends on the type of control and the type of action.

NOTE In cases where a keyword represents a verification step, the required input for the keyword can be the expected output or a state for the referenced object.

Some keywords can also accept a number of optional inputs; in such cases, the framework must hold default values for those that are not provided (e.g. “Click UI_Element 456,123” can refer to a specific co-ordinate in the UI_Element, while “Click UI_Element” with no specified co-ordinate can default to clicking the center of that element).

For composite keywords, which can cover extensive functionality, the number of parameters can grow and the test data can become complex. It is a good practice to decouple the data from the actions. Therefore, multiple parameters can be stored separately and a unique reference to the data is used as input for the keyword.

EXAMPLE A composite keyword "createCustomer" requires data such as first name, surname and address of the customer. Instead of documenting the test data with the keyword test case, it is stored in a database. This allows a single reference to the complex data in the database, and the test case can be extended by providing several sets of data which are associated with the same sequence of actions.

Data-driven testing is a method of storing test data separately from the sequence of actions, which is independent of keyword-driven testing but is frequently used in conjunction with keyword-driven testing. In data-driven testing, for one test case with a defined sequence of actions, multiple sets of data can be provided. The sequence of actions is then executed for each of the sets of data. Depending on the implementation, the

data is either stored in a table, spreadsheet, or database. Data-driven testing is an option to decouple the parameters from the test which matches very well with the concepts of keyword-driven testing.

See 6.4 for more details on data-driven testing.

6 Application of keyword-driven testing

6.1 Overview

This clause addresses some concepts which contribute to a successful implementation of keyword-driven testing. While all of these concepts are not required for each keyword test case, test design can benefit from them.

There are six concepts covered in this clause.

a) identifying and documenting keywords (6.2);

b) composing test cases (6.3);

c) keywords and data-driven testing (6.4);

d) modularity and refactoring (6.5);

e) keyword-driven testing in the test design and implementation process (6.6);

f) converting non-keyword-driven test cases into keyword-driven test cases (6.7).

Additional guidance on the introduction of keyword-driven testing as well as examples on roles and tasks can be found in Annex D. Annex C provides assistance for applying keyword-driven testing.

6.2 Identifying and documenting keywords

Identifying keywords is a pivotal task in keyword-driven testing as the contents, granularity and structure of the keywords can impact the way keyword test cases are defined. It is important to name keywords in a way that appears natural to the people who will be working with them.

When identifying keywords, the following steps are executed:

a) determine the layers needed in the given context and define what sort of keywords (e.g. functionality, granularity) are supposed to be assigned to the layers;

b) identify keywords in the layer based on the definition or scope of each layer.

Generally, keywords are defined by first identifying sets of actions that are expected to occur frequently in the testing. A name (e.g. the keyword) is applied to an action or group of actions. Keywords are applicable in a range of situations. At this point it is useful to determine which of the actions are information-dependent (e.g. time, data, situation), and so identify which keywords need to be associated with parameters.

A keyword is described by the following information:

— the name of the keyword, which tells the reader what this keyword is expected to do;

— the parameters of the keyword, which can be empty;

— documentation on the keyword, including the layer in which this keyword is expected to be used, the keyword type (e.g. navigation or verification), the context in which it is to be used, the actions included with the keyword, either as a description, or as a reference to keywords on a lower layer (see next bullet), and the objectives of the keyword;

— if the keyword is composed from other keywords, a list of the included keywords in the order in which they are used.

Simple keywords can be identified by observing different interactions available at the test input interface, such as interactions with keyboard, mouse, touchscreen, microphone, API.

Composite keywords can be identified by observing common actions that the user performs at the UI level.

EXAMPLE "GoTo" would be used instead of "ClickButton", or "Select" instead of "ClickRowInTable".

Typical business behaviour can be encapsulated in a composite keyword (e.g. "CreateNewUser"). Other complex manipulations like interactions with databases can also be candidate keywords in the framework.

The following issues should be considered.

— Uniqueness: each keyword should be unique in its context of use.

— Reusability: the keywords should be defined in a way that best supports future reusability.

— Completeness: keywords should be defined with a view to all known elements and possible interactions of the test interface (e.g. all known objects in the GUI and its dialogs).

— Clarity: all keywords should be defined with a clear and consistent structure. All keywords in a layer should have a similar abstraction level.

— Specificity: keywords should not be redundant and should be mutually exclusive (i.e. keywords will represent distinct actions), to ease the test design and to decrease the maintenance effort.

NOTE 2 In some environments it can be useful to use an object-oriented approach to identify and describe keywords. Keywords can, in that approach, be identified by analysing the available objects and methods on the objects in the domain. Keywords can then be described in a style like "OBJ.Action Parameter", where "OBJ" refers to the object which is to be addressed (e.g. a button in a user dialogue box), "Action" refers to the activity (e.g. "press" for a button), and "parameter" refers to a list of additionally-needed parameters. This approach can be useful if all stakeholders performing keyword-driven testing within that environment are familiar with object-orientation.

6.3 Composing test cases

Keyword test cases can be composed from previously-defined keywords. In the process of writing test cases, it can occur that missing keywords are discovered and can, therefore, be defined after that point.

NOTE Keyword test cases can be composed from composite keywords and used to build end-to-end tests.

Within the test case specification (see ISO/IEC/IEEE 29119-3), test cases can be documented using appropriate notation, including the use of tables or databases. The format depends on the available infrastructure (e.g. availability of a test management tool and the plans for automated execution).

Keyword test cases usually contain keywords from a single layer. A clear distinction is made between the layers. This distinction opens the option to distribute the design of different layers to different testers (see Clause 7).

EXAMPLE 1 Test of an ATM using only simple keywords at the test interface layer:

enterValue("Card", 123000789)

enterValue(“PIN”, 1234)

selectObject("Button", "OK")

selectObject(“Button”, “Payment”)

enterValue(“Amount”, “200”)

selectObject(“Button”, “executePayment”)

verifyObject(“Payment”, ”200”)

EXAMPLE 2 Test of an ATM using domain layer keywords:

signInUser(123000789, 1234)

executePayment(“200”)

verifyPayment(“200”)

More examples can be found in Annex E and in Annex F.

6.4 Keywords and data-driven testing

Keywords combined with parameters and separate data sets for these parameters (e.g. data-driven) can offer improved testing. Data-driven testing can be applied when the same sequence of keyword actions are to be used with different sets of data. In this combined approach, the data can be stored separately from the keyword test cases. The data can be stored in constructs such as tables, databases, or real-time generators, and is then read into the keyword test cases. The repeated keyword sequence using different data, in effect, creates new tests.

EXAMPLE Data-driven testing is useful for multi-lingual testing or internationalization testing, where the same test cases are to be performed at user interfaces but with different languages. Not all internationalization issues are easily addressed by data-driven testing: in the case of lexicographical sorting, the requested item can not only have a different label but also a different position.

The following guidelines are taken into consideration for data-driven testing with keywords:

— A keyword does not have to be “loop aware”. In other words, a keyword ideally works the same whether it is part of a linear sequence or is contained within a loop (such as in a data-driven test). This places the burden of managing the data file and fetching its content on the framework, not on the keyword. It implies that the only method of getting data into and out of a keyword is through its parameters.

— Multiple, non-nested loops in a test case, can be implemented, but should not be used. Good data-driven test design suggests the use of a single loop in test cases that are data-driven.

— Nested data-driven loops should not be used. Nesting data-driven loops by more than two is normally avoided.

— The format of the data file and its contents are implementation defined. This document does not dictate the format of the file (e.g. an implementation can support data from Excel files, text files, or any other file type). Neither does this document dictate the format of the data items within the file (e.g. XML, ASCII or Unicode text, binary encoding, or any other format is permitted).

NOTE Although keyword-driven testing and data-driven testing are concepts that can be used independently in theory, in practice keyword-driven testing includes data-driven testing.

6.5 Modularity and refactoring

Modularity in keyword-driven testing is used to improve the longevity of the test cases. However, with the passage of time, changes in the test item, new test cases or new people on the team can all lead to maintenance issues.

Possible issues are as follows:

— redundant keywords: where two or more keywords for the same objective come into existence;

— unused keywords;

— conflicts where changes in keywords (e.g. structure or semantic), which fix an issue in a number of test cases, create new issues in other test cases; this has associated cost factors;

— Uncoordinated changes in keywords (e.g. name, semantics, parameters) cause rework or invalidate test cases of other testers.

To avoid these issues, the following maintenance actions should be considered.

— A framework for keyword-driven testing should provide a way of creating a cross-reference for the used keywords, allowing identification of which keywords are used in which places and how frequently they are used. This shows if, and how much, a change in a keyword will affect existing test cases.

— In some organizations, an authority is required who is responsible for all keywords, additions and any changes to existing keywords or how they are used. That authority assures consistency throughout the project including both development and testing stages.

— On a regular basis (e.g. once a month), a keyword review meeting can be held. At this meeting, testers can decide about the introduction or modification of keywords and discuss the structure of the keywords. If an authority is in charge of keywords, they should also be in attendance.

— A clear structure to document the keywords should be produced. Keywords can be grouped by layer, test item and region in the test item (e.g. dialog, objective or others). Keywords that are supposed to be usable by all testers will normally be stored separately from keywords which are only useful for a limited number of people.

— Keywords can be subjected to configuration management practices (e.g. the authority mentioned above). The ability to change keywords would normally be limited to those who need to do changes or the authority. All changes need to be well documented. Access to prior versions (e.g. the option for rolling back) should also be provided.

6.6 Keyword-driven testing in the test design and implementation process

6.6.1 Overview

The test design and implementation process defined in ISO/IEC/IEEE 29119-2 (see Figure 6) is applicable to this document. This clause describes the relationship between the activities of the test design and implementation process and keyword-driven testing.

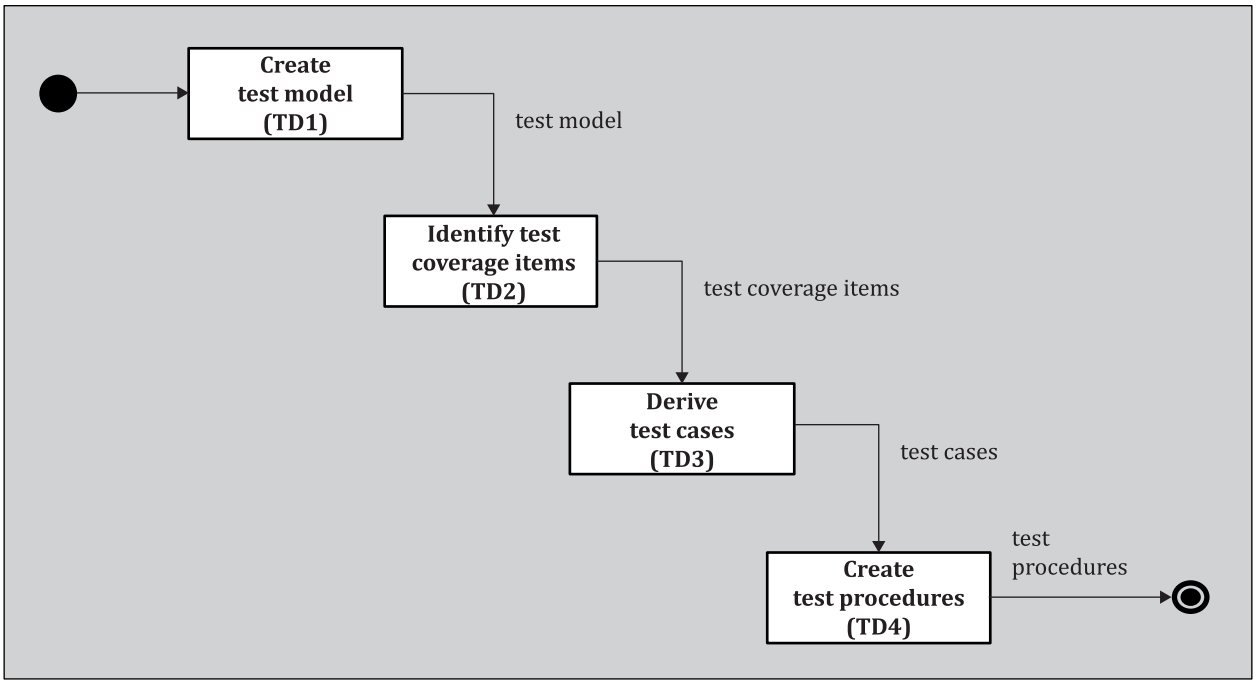


Figure 6 — ISO/IEC/IEEE 29119-2 test design and implementation process

The test design and implementation process in ISO/IEC/IEEE 29119-2:2021, Figure 10 (as shown in Figure 6) describes four steps from 'Create test model' (TD1) to 'Create test procedures' (TD4).

The requirements of TD3 to TD4 in ISO/IEC/IEEE 29119-2 are applicable to the test design and implementation process used in keyword-driven testing.

TD1 to TD2 are not addressed in this document. Keyword-driven testing is relevant when performing the activities TD3 – 'Derive test cases' and TD4 – 'Create test procedures'. These are covered in 6.6.4 and 6.6.5.

6.6.2 Create test model (TD1)

The creation of the test model in keyword-driven testing is as defined in ISO/IEC/IEEE 29119-2.

6.6.3 Identify test coverage items (TD2)

The identification of test coverage items in keyword-driven testing is as defined in ISO/IEC/IEEE 29119-2.

6.6.4 Derive test cases (TD3)

6.6.4.1 Overview

In TD3, keyword-driven testing is focused on composing keyword test cases.

A keyword test case is specified using one or more keywords. New keywords can be created or existing keywords (e.g. created from an earlier test case) can be reused.

According to ISO/IEC/IEEE 29119-2, test cases are derived by the following steps to exercise test coverage items:

— determine preconditions;

— select input values;

— select actions (where necessary);

— determine expected results.

These design activities identify the different actions that need to be performed to prepare the test item, execute the test and verify the test result. Keywords can fulfil one or more of these actions.

Examples of keyword-driven test cases can be found in Annex F.

6.6.4.2 Determine preconditions

The tester identifies the needed test preconditions and decides which of them can be achieved using keywords. Composite high-level keywords (see 5.3.2) can be appropriate in a situation where multiple actions are required. Additionally in some testing, keywords can be used to prepare more general preconditions at the system level (e.g. importing data into a database or setting parameters for application start which can be used over a series of test cases) before other keywords establish the preconditions that are specific to an individual test case.

6.6.4.3 Select input values

The tester selects input values based on test design considerations and then implements these input values in keywords. In situations where test cases are supposed to be executed by the same actions, but with different sets of input values to provide different test outcomes, the keywords are normally developed to support data-driven testing (see 5.6).

6.6.4.4 Select actions

The tester identifies any required actions. If the required actions are not available from existing keywords, either new keywords are created or existing keywords are combined into composite keywords, as appropriate, to provide the needed functionality.

NOTE As keywords are sometimes already defined, the iterative nature of the test design and implementation process suggests that keywords can be subject to refactoring.

6.6.4.5 Determine expected results

The tester determines expected results. Keywords (new or existing) can be used to describe checks in the keyword test case that are used later, during test execution, to compare these expected results with the actual results (e.g. conformity with generic rules that are the subject of the test). These keywords include logging of the test result and can additionally include logging of extended information, e.g. for analysis in case of a defect (see 5.3.3 and 5.3.4).

6.6.5 Create test procedures (TD4)

The developed keyword test cases become the primary input for creating test procedures.

A framework for keyword-driven testing can provide mechanisms for assembling test procedures from various test cases by applying different criteria (e.g. choosing test cases meant to be run in the same test environment set-up, or in data-driven testing, providing values for parameterized test cases which in turn are based on keywords with parameters).

Additionally, a keyword framework can provide mechanisms to determine the execution order within a test procedure. The framework can also be designed to ensure that required preconditions are set up before test execution. This can require additional generic keywords.

6.7 Converting non-keyword-driven test cases into keyword-driven test cases

If keyword-driven testing is to be introduced into an existing project, existing test cases can be converted into keyword test cases.

In addition to the advantages of keyword-driven testing, reasons for deciding to convert existing test cases into keyword-driven test cases include the following.

— Uniformity: ensuring that all test cases have a similar structure and style will enhance readability, maintainability and so reduce costs. Future maintenance can be cheaper if only one style of test case is to be maintained.

— Efficiency: keywords identified from existing test cases can be reusable in future test cases.

— Automation: the same automation framework can be used for old and new test cases.

— Understandability: test cases can be used and maintained by non-technical testers (often with business knowledge).

Reasons to keep existing test cases and not convert them into keyword-driven test cases include the following.

— The number of existing test cases is large compared with the number of additional test cases that are needed.

— There is proven and maintainable automation for the existing test cases.

— The cost of converting the test cases is expected to exceed the benefits.

The decision to move to keyword-driven testing for existing projects should be carefully considered.

7 Keyword-driven testing frameworks

7.1 Overview

If keyword-driven testing is used, everything necessary to support the approach is organized in a keyword-driven testing framework. The complexity of the framework depends on its purpose. The components of a framework are described in 7.2.

This subclause addresses several points to consider when implementing or using frameworks for keyword-driven testing.

While 7.3 describes basic attributes for keyword-driven testing frameworks that are necessary for the implementation of keyword-driven testing, 7.4 lists advanced attributes that provide additional benefits and are desirable, but are not expected to be available in all frameworks.

The attributes are divided into general, test design tool, and test execution engine aspects as follows.

— General aspects are mostly tool-independent.

— Test design tool aspects are related to a software tool component of the framework that is used to manage keywords, compose test cases from keywords, and assign data.

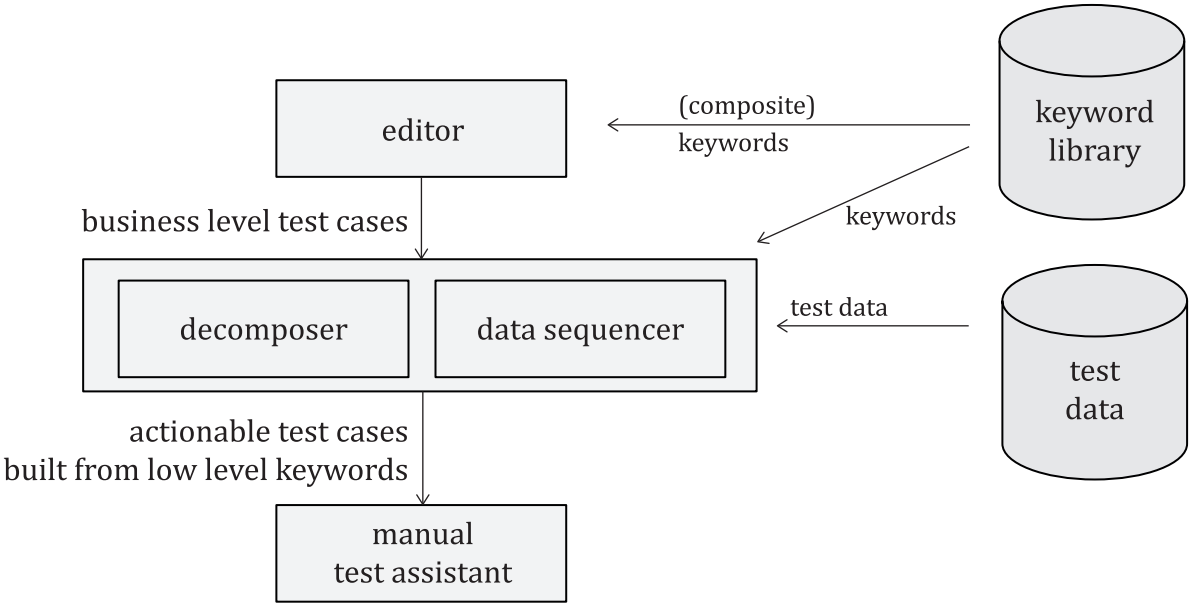
— Test execution engine aspects are related to the software tool component of the framework which is used to execute the test cases as automated tests.

A framework is likely composed of a series of tools each providing different parts of the needed capabilities. It is sufficient if the framework as a whole meets the named requirements.

7.2 Components of a keyword-driven testing framework

7.2.1 Overview

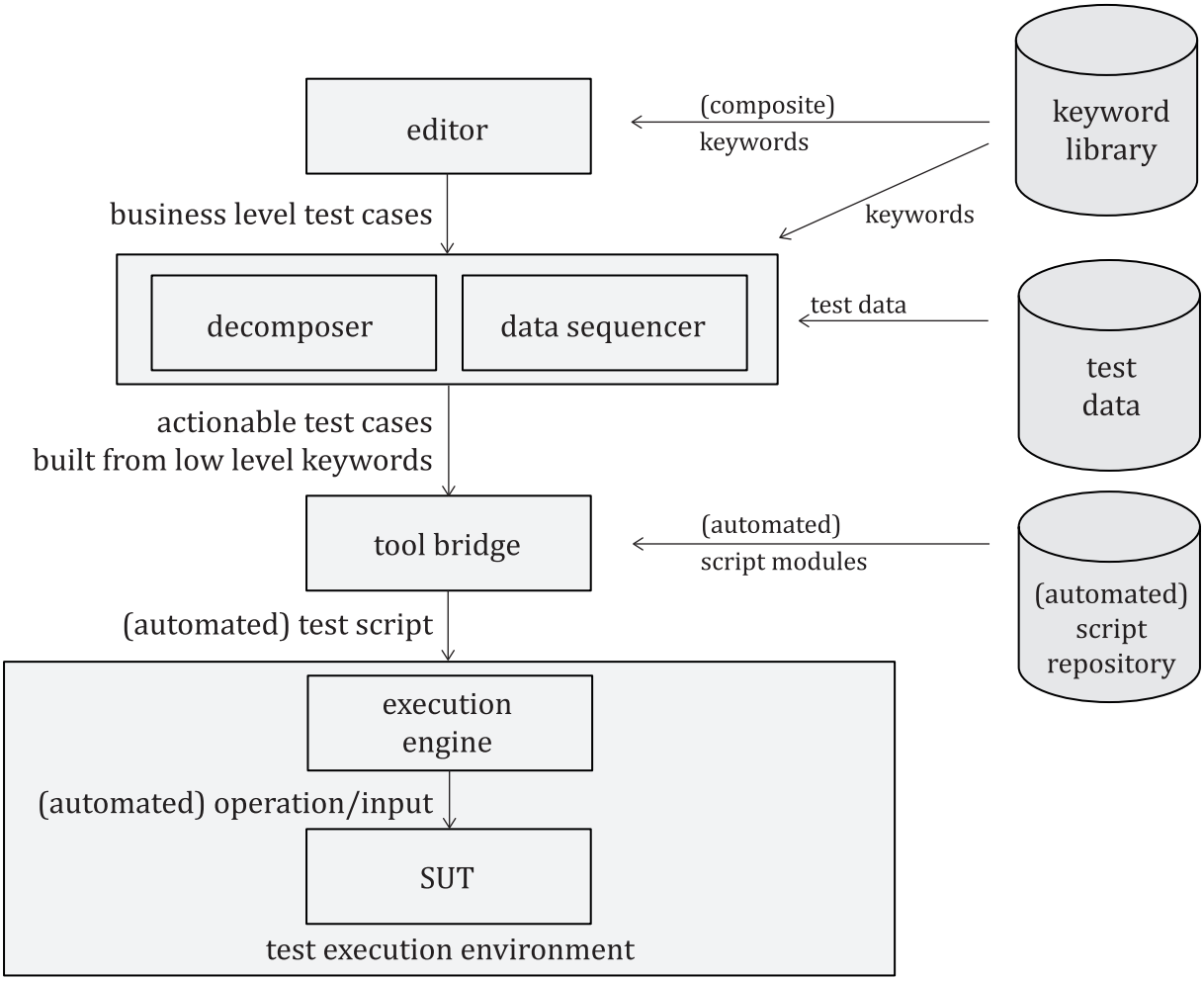
The keyword-driven testing framework can be realized in a variety of ways including by commercial tools, custom tools, and solutions in the form of script libraries. It comprises functional units (or functional areas) shown in Figure 7. This document does not describe how these functional units are to be implemented. In practice, a commercial or custom software tool can cover these functional units in parts or completely. One or more software tools, along with custom implementations, libraries and organizational processes can form a keyword-driven testing framework. Tools can have a different overall organization. A framework must cover the requirements described in 7.3 to conform to this document (see Clause 4).



NOTE Not shown in this figure are the documentation and organizational processes (see 7.2.11).

Figure 7 — Components of a keyword-driven testing framework for manual test execution

In Figure 7, a keyword-driven test framework is shown that is restricted to the support of manual testing. If test automation is required, the framework would be comprised of additional elements, as shown in Figure 8, and the manual test assistant is not required. Instead, a test execution engine is used to run the test cases against the test item. A tool bridge is used as a link between the keywords and their representation in the automated test execution environment. The tool bridge's main task is to transform the necessary information into a suitable format for the test execution engine.



NOTE Not shown in this figure are the documentation and organizational processes (see 7.2.11).

Figure 8 — Components of a keyword-driven testing framework for automated test execution

The functional components and processes which form a keyword-driven testing framework are explained in 7.2.2 to 7.2.11.

NOTE These component descriptions do not assume any specific implementation of the components. In practice, one specific tool can cover some of these components (e.g. keyword-driven editor, decomposer, data sequencer and manual test assistant can be included seamlessly in one test management tool). Other implementations can provide only parts of one of the components in one tool.

7.2.2 Keyword-driven editor

The keyword-driven editor must be available to compose keyword test cases from keywords. The keywords can be taken from a keyword library (7.2.8).

In practice, the keyword-driven editor can be implemented in various ways.

EXAMPLE Possible implementations of a keyword-driven editor include, but are not limited to, a spreadsheet application, a dedicated standalone application or can be part of a test management tool.

7.2.3 Decomposer

The decomposer must be available if composite keywords are used. The main task of the decomposer is to transform the composite keyword test case, which consists of a sequence of high-level keywords, into the appropriate sequence of low-level keywords.

7.2.4 Data sequencer

The data sequencer must be available if keyword-driven testing is to be applied with several sets of data associated with one keyword test case (i.e. data-driven testing is also being used). The main task of the data sequencer is to transform the sequence of keywords (e.g. low-level or high-level) which are not yet associated with data to a list of keywords with specific data.

By doing this, the original actions, which have been written only once in the keyword-driven editor, are repeated for any desired set of data. All parameters or placeholders are replaced by the final value needed in the respective test case.

The data sequencer can work both on high- and low-level keywords.

NOTE Depending on the implementation, the tasks of the decomposer and the data sequencer can be performed in arbitrary order. Both tasks can be done by the same software implementation.

7.2.5 Manual test assistant

The manual test assistant only must be available for manual test execution. Its task is to present the test cases as prepared by the decomposer and data sequencer in an actionable way to the human tester. The tester then performs every single action, as well as documenting the test execution and the results.

In practice, the manual test assistant is frequently part of a test management tool.

7.2.6 Tool bridge

The tool bridge only must be available for automated test execution and has a similar function to the manual test assistant used to support manual test execution.

The task of the tool bridge is to provide a connection between the keywords, as they appear in the keyword test case or in the keyword library, and the associated implementation in the test execution environment.

For each keyword passed from the data sequencer or from the decomposer, the tool bridge, depending on the implementation, requests the test execution engine to call the corresponding script (e.g. keyword execution code) or appropriate functions, and by that perform the right actions with the appropriate data, if applicable.

In practice, a tool bridge can be implemented as a separate software tool, as a script in a test automation tool's runtime environment, or as part of a test management tool. Some bridge implementations can be referred to as a "generator," for example, when a script or parts of scripts are generated for execution by an automation tool. Additionally, a bridge can be called an "interpreter" or "engine", for example, when a script is executed to interpret the sequence of keywords and calls the corresponding sub-functions.

7.2.7 Test execution environment and test execution engine

To support automated keyword-driven testing, the test environment contains a test execution engine with links to the item under test. The test execution engine is a tool implemented either by software, hardware, or both. Its task is to execute the test cases by performing the actions associated with the keywords.

In practice, the implementation of a test execution engine varies depending on the test item and environment. The test execution engine can be a commercial test execution tool, (e.g. a capture and playback tool, or it can also be a hardware appliance, controlled by software, such as a robot).

7.2.8 Keyword library

The keyword library stores keyword definitions for one or more projects or portions of those projects. It is used to store the core information on keywords, such as name, description, parameters, and, in the case of composite keywords, the list of keywords from which the respective keyword is composed or derived. For test automation, it also contains necessary information for the tool bridge to associate the keywords with the keyword execution code in the script repository. The keyword library can help the tester find a keyword that implements a required action in a test case.

In practice, a keyword library is supported by a test management tool.

7.2.9 Data

The data component in the keyword-driven testing framework refers to the test data used for the keyword test cases.

Keyword test cases can be designed so the test data is included in the test cases. In this case, external test data is not be required. In other implementations, the keyword test case does not contain actual data, but contains placeholders which need to be substituted with data before the test case can be executed. In this case, the test data needs to be stored. In practice, it is common to store that data in files, in a spreadsheet application, in a dedicated database, or in a test management tool.

7.2.10 Script repository

The script repository stores keyword execution code. It only must be available if keyword-driven testing is done with the aim of executing the test cases automatically.

For automation of keyword-driven testing, each keyword must be associated with at least one implementation (e.g. a command, script or function), which corresponds to the actions associated with that keyword.

In practice, the script repository is frequently implemented by either a test automation tool or stored at a defined location in the file system.

7.2.11 Documentation and organizational processes

A keyword-driven testing framework can be based on several tools covering the components defined in 7.2.2 to 7.2.10.

The documentation of the keyword-driven testing framework typically forms part of the organizational test practices. This documentation describes how the tools are integrated and how they are used within the organization's projects.

Additionally, the documentation can define rules on how to apply keyword-driven testing, e.g. by defining layers (see 5.2) or conventions (see Annex A for additional information and typical conventions).

7.3 Basic attributes of the keyword-driven testing framework

7.3.1 General information on basic attributes

This subclause defines framework attributes that are generally necessary for the application of keyword-driven testing. It describes attributes which are required in keyword-driven testing frameworks and are necessary for conformance to this document.

7.3.2 to 7.3.9 structure these attributes and requirements by the components of keyword-driven testing frameworks according to 7.2.

7.3.2 Documentation

Keyword-driven testing frameworks demonstrate the following attributes.

a) There shall be documentation recorded describing each keyword.

NOTE 1 This is necessary for people to understand and use the defined keywords appropriately to build their test cases. The description is improved by the inclusion of an example.

b) There shall be documentation recorded for the parameters of each keyword.

NOTE 2 Keyword and parameter documentation includes naming of the keywords and how they are described, the parameters' maximum length, allowed characters, optionally reserved names or characters, and documentation rules.

NOTE 3 An example of keyword documentation is given in F.2.

c) A default value shall be documented for every parameter in case a value for a parameter is missed in the keyword test case definition.

d) There shall be high-level documentation recorded describing the hierarchy of the keywords that can be used.

e) There shall be high-level documentation recorded describing how data is stored and referenced for data-driven tests.

EXAMPLE Data can be stored in a database or in a spreadsheet, and can be organized in columns or rows.

The documentation described above can be part of the test plan, test policy, organizational test practices (see ISO/IEC/IEEE 29119-3) or a standalone document with references to/from other test documents.

There are several options for recording this information, such as word processors or test management tools.

7.3.3 Keyword-driven editor (tool)

When creating keyword test cases, a tool which supports the building of test cases should be used.

Keyword-driven editors provide the following to support testers.

a) Within the keyword-driven editor, non-composite keywords shall be displayed with their associated actions.

EXAMPLE 1 A keyword "login" is associated with the following actions:

— press button >>login<<

— enter user name

— enter password

— press button >>submit<<

NOTE 1 A keyword like "login" can be designed to be composite (see Figure 1) or non-composite. In this example it is assumed that the tester has decided to define "login" as a non-composite keyword.

b) For keywords which have been defined with lower-level keywords, the user shall be able to access this definition within the keyword-driven editor.

c) The keyword-driven editor shall allow the use of keywords with parameters to support data-driven testing.

d) The keyword-driven editor shall provide the capability to enter comments.

e) The keyword-driven editor shall offer the capability to connect to data sources that are used to assign values to parameters.

NOTE 2 Through this capability test cases become keyword-driven and data-driven. While it is possible to use keyword-driven testing without data-driven testing, in practice data-driven testing is so important for efficient keyword-driven testing that frameworks for keyword-driven testing are expected to offer the option of data-driven testing.

EXAMPLE 2 Possible data sources are databases or spreadsheets.

f) Within the keyword-driven editor, multiple uses of keywords shall be implemented by reference to the script repository.

NOTE 3 Copying implementations of keywords in the script repository can be avoided by using references.

g) The keyword-driven editor may provide the capability to define the order in which the test cases are to be executed.

NOTE 4 The test execution order is part of deriving test procedures.

7.3.4 Decomposer and data sequencer

Decomposer and data sequencer provide the following.

a) The decomposer shall decompose higher-level keywords into lower-level keywords.

b) The decomposer shall process parameters, including assuring that the parameters associated with the higher-level keywords are decomposed and associated with the lower-level keywords.

c) The data sequencer shall be able to process parameters.

7.3.5 Manual test assistant (tool)

A manual test assistant provides the following.

a) The manual test assistant shall support manual test execution based on the defined test cases by presenting the user with the keywords that make up the test case and allowing the user to specify the pass/fail attribute for each keyword.

b) The manual test assistant shall provide support for logging defects associated with a test failure.

7.3.6 Tool bridge

The tool bridge shall provide the test execution engine with the appropriate execution code to execute the test cases.

7.3.7 Test execution engine

Test execution engines are designed to execute test cases. One or more test interfaces (e.g. an API, a GUI or a hardware interface) are used to access the test item. A test execution engine can be implemented by software, by hardware or both.

Requirements that apply to the test execution engine include the following.

a) Keywords shall be executed sequentially starting with the first keyword unless the test script includes conditions or loops.

NOTE 1 This is in general; but exception handling can require non-sequential execution to process an abort.

b) The test execution engine shall provide support for both literal values and variables as values of parameters.

c) The test execution engine shall provide execution results at the keyword level for each execution of each keyword implementation.

NOTE 2 In this way, a user can tell from the test results whether a keyword was executed successfully, or, if execution of the keyword failed, identify the reason for the failure (e.g. text field not writeable, field not present).

d) The test execution engine shall be able to store the timestamp of its executions with the duration of each execution.

e) The test execution engine shall be able to identify unimplemented keywords.

NOTE 3 A keyword is unimplemented if there is no execution code for that keyword.

f) The test execution engine shall provide an error recognition mechanism to handle errors returned by keywords as described in 5.3.4.

NOTE 4 As a consequence, the test execution engine either limits the number of cycles in a loop or provides another means to make sure that unlimited loops are impossible (e.g. by terminating each loop after a predefined time).

g) The test execution engine shall provide a clear statement of PASS/FAIL for a test case whenever there are passed and failed executions in one loop.

NOTE 5 If a loop contains a verification, it can happen that the verification fails for some, but not all loop cycles. The PASS/FAIL statement indicates if this situation is either "PASS" or "FAIL" for the test case.

h) The test execution engine shall include the unique identifier of the execution in the record of the execution.

i) The test execution engine shall include the unique identifier of the test environment in the record of the execution.

j) The test execution engine shall include the unique identifier of the test item in the record of the execution.

k) The test execution engine shall support multi-application keywords by providing a mechanism to select between multiple implementations of a keyword.

NOTE 6 Multi-application keywords are keywords that can be used in different applications. From a user's perspective, it is a single keyword, but to be applicable in different applications, different implementations for the one keyword are required. A test execution engine which supports this recognizes which of the two applications the keyword is to be applied on and selects the correct implementation for it. This allows a test case to manipulate more than one application using keywords written for each application. For example, a test case that verifies interoperability of an office application suite is able to use keywords written for each of the two applications in a single test case.

l) The test results shall be available to the user.

NOTE 7 Other components include test design or test management components.

7.3.8 Keyword library

The keyword library shall support the definition of keywords that includes the basic attributes of name, description and parameters.

7.3.9 Script repository

Requirements that apply to the script repository include the following.

a) The script repository shall support the storage of keyword execution code.

b) The script repository shall provide a mapping from each keyword to its corresponding keyword execution code in the keyword library.

7.4 Advanced attributes of frameworks

7.4.1 General information on advanced attributes

This subclause defines additional attributes that are recommended to achieve the full benefits of keyword-driven testing. Basic keyword-driven testing is possible without these attributes. This subclause does not identify requirements necessary for conformity with this document.

7.4.2 to 7.4.10 structure these attributes in terms of the components of keyword-driven testing frameworks according to 7.2.

7.4.2 Documentation

Keyword-driven testing frameworks demonstrate the following attributes.

a) There should be high-level documentation recorded that describes the rules of how the keywords can be composed into test cases.

b) There should be high-level documentation recorded which describes the rules of how parameters are described.

c) There should be high-level documentation recorded which describes the rules of how parameters are passed.

d) There should be high-level documentation recorded describing how keywords are defined.

7.4.3 Keyword-driven editor (tool)

Features of a keyword-driven editor include the following.

a) The keyword-driven editor should provide a function for checking the syntax of the test cases composed of the keywords.

b) The keyword-driven editor should provide the capability to track keyword usage and provide a cross-reference to indicate in which test cases and composite keywords each keyword is used.

c) During any syntax checking the keyword-driven editor should check that only defined keywords are used in test cases and either reject undefined keywords or mark them as undefined.

d) For keywords that have parameters, the keyword-driven editor should check the correctness of each parameter count, and type.

NOTE 1 The parameter count is the number of parameters which are provided when using a keyword.

EXAMPLE Parameter types can be (not limited to) a number, text string or something as complex as an address.

e) There should be a capability to define exception handling (e.g. if an exception occurs on test execution, it should be possible to define which clean-up steps are executed) within the keyword-driven editor.

f) The keyword-driven editor should allow auto-completion or drag and drop for allowed keywords and their parameters.

g) The keyword-driven editor should support versioning of keyword test cases.

7.4.4 Decomposer and data sequencer

Features of the decomposer and data sequencer include the following.

a) The decomposer should support the implementation of hierarchical keywords.

b) The decomposer and data sequencer should support hierarchical structured data.

7.4.5 Manual test assistant

The manual test assistant should provide the capability to attach screenshots or other outputs of the test item to the test log.

7.4.6 Tool bridge

No advanced features are defined for the tool bridge.

7.4.7 Test execution environment and test execution engine

A test execution environment and test execution engine support the testers as follows.

a) Keyword execution code should be able to read, store and process data from test items.

b) Variable name space support should be provided.

EXAMPLE 1 Multiple applications are addressed in a single test case, using different variables in each application with the same name but in different namespaces.

c) Context switching when moving between applications in a test case should be supported.

d) Implementations should manage the switch between namespaces (e.g. when changing application references).

e) Testing that multiple users of an application can access the same shared data in parallel should be supported.

f) The test execution engine should handle blocked keywords on the test item by continuing test execution with the next appropriate keyword (see 5.3.4).

NOTE 1 The next appropriate keyword is either defined by the test designer, or, if no such definition has been done, the next keyword in the test case.

g) The test execution engine should be capable of handling keywords with attributes such as "may be blocked" and "must not be blocked" (see 5.3.4).

h) The test execution engine should support data-driven tests.

NOTE 2 This includes, at a minimum, a looping construct that allows iteration over a set of one or more keywords, using data values read from an external data file. See 6.4 for a detailed discussion.

i) There should be a capability to define conditional actions.

j) When an exception is handled, there should be a capability to skip actions and ensure that defined clean-up steps are executed.

NOTE 3 This includes the ability for the keyword execution code to request that the test case abort, i.e. not executing those subsequent keywords, usually as a result of an unrecoverable situation detected in the requesting keyword.

k) Each implementation for a keyword should provide the information needed to perform the required actions, such as input parameters or the object of the action. Each test step represented by a keyword thus contains all information for performing the action.

l) The test execution engine should be able to verify whether keywords received from tables, test management tool etc., match their keyword execution code by comparing count and type of parameters.

m) The test execution engine should ensure that all loops in keyword test cases are restricted in a way that infinite loops are prevented.

EXAMPLE 2 A test execution engine supports loops limited by a given number of passes.

EXAMPLE 3 A test execution engine supports loops limited by a fixed period of time. In a test case, a loop is limited to wait for a maximum time of 2 s for an event to occur or timeout.

7.4.8 Keyword library

Features of a keyword library include the following.

a) The keyword library should support hierarchical keywords.

b) The keyword library should support versioning of keywords.

c) The keyword library should support the implementation of aliasing, synonyms, and internationalization to facilitate the creation of test cases.

7.4.9 Test data support

Features of test data support provided by the keyword-driven testing framework include the following.

a) The framework should support versioning of test data.

b) The framework should support hierarchical data types.

7.4.10 Script repository

The framework should support versioning of keyword execution code.

8 Data interchange

Keyword-driven testing is typically supported by software tools which are components of the keyword-driven test framework. The tools must be able to receive understandable data and output understandable data (interchange data).

Data interchange between humans and a software tool, mostly at the user interface, is not addressed in this clause.

NOTE The term "tool" can refer to both commercial tools and custom-built (non-commercial) parts a framework.

Data interchange in keyword-driven testing should be supported by a standard published by an internationally-recognized standardization body (e.g. ISO, IEEE, OMG).

An example of a data format to interchange keywords is given in Annex G.

Annex A
(informative)

Typical keyword conventions

The following are typical conventions for keywords.

NOTE The words “shall” and “should” appear in some of the conventions. These “shall”s and “should”s are example wordings only and do not represent requirements and recommendations of this document.

a) Keywords should contain a verb.

b) Keywords should use the imperative form.

c) Keywords shall provide a description of the associated set of actions.

d) Keyword descriptions shall be unambiguous.

e) Keywords shall be defined in a way that they are understandable by the stakeholders who will use them when designing test cases.

   NOTE 1 This can be verified by reviewing the keywords with the stakeholders.

f) Every keyword shall be unique in its meaning within a framework.

g) Keywords should not be derived from programming languages.

NOTE 2 Programming languages can be too intangible or difficult to understand. Knowledge of a programming language by the domain experts who will specify the test cases cannot be assumed.

The following example illustrates items a) to g).

EXAMPLE The keyword "pressButton" contains a verb (a) in imperative form (b). The description is "This keyword is used to trigger an element of class <button> in the graphical user interface" (c). If it is associated with a parameter that identifies the button [e.g. "pressButton <cancel>" it is unambiguous (d)]. This keyword is assumed to be understandable by English speaking stakeholders, as the words "press" and "button" in the keyword's name are taken from the testers' usual vocabulary (e). Uniqueness of meaning (f) is given as long as no other keyword is introduced which refers to the same activities.

NOTE 3 Natural language can be ambiguous, contain synonyms and homonyms, and can result in unclear and ambiguous test cases.

Annex B
(informative)

Benefits and issues of keyword-driven testing

B.1 General benefits of keyword-driven testing

By composing test cases from a defined set of keywords, the benefits can include the following.

— Keywords can be defined in natural language meaning that test cases can be written with more or less detail, depending on the project's needs.

— Test cases become clear and understandable. This supports efficient manual test execution.

— Keywords make it easier for users (and others) to create, understand and maintain test cases.

— Using unambiguous and precisely defined keywords allows the option to select whether the execution of a test case is done manually or is done with automation. In the case of automation, it is expected that the keywords will be implemented as keyword-scripts.

— Testers working at the business level do not require technical understanding of the test automation framework to be able to create and edit test cases which are fully suitable for test automation.

— Testers working at the technical level can automate keyword-driven test cases, even if they have limited or no understanding of the business domain, by automating the keywords for the high-level test cases of the subject matter experts.

— Testers can implement test cases using a language that is understandable to domain experts and that can be reviewed by them for business correctness. If this is done, then keyword-driven testing can help to close a frequently perceived gap between the business level and the technical level.

— Maintenance of the keyword scripts at the technical level is unlikely to affect the logic of the test cases. So, in general, there is no need in re-specifying or re-formulating the keyword test case if the technical implementation of the keywords is adjusted.

— Sensitivity to changes (which can create the need for maintenance effort) is reduced.

— Portability of a set of test cases is easier to achieve (e.g. if a similar system with almost the same business cases has to be tested then many of the keywords can be reused).

— Test cases composed of keywords can be created faster than those written in natural language.

— Refactoring of test cases is cheaper because it is likely that only individual keywords or keyword scripts need to be changed.

B.2 Benefits of keyword-driven testing for test automation

Benefits of keyword-driven testing in the case of test automation can include.

— Automated functional tests can be implemented before the test item actually exists, either by using existing keyword libraries with their corresponding automation scripts, or by defining new keywords and adding the automation scripts later as the test interface is defined.

— The number of keywords required for a test item is much lower than the number of test cases required. The effort of implementing test automation scales with the number of keywords instead of with the much

faster increasing number of test cases. This is particularly beneficial as at some point there are usually no or very few new (low-level) keywords needed for new test cases when a new feature is implemented.

— As long as test cases are constructed from the established set of keywords, once these keywords have been implemented, new test cases do not need any additional implementation effort to be automated.

— Maintenance of test cases for business reasons does not affect the implementation of the keyword scripts, as long as the set of keywords and the semantics of the keywords are not changed.

EXAMPLE An application is changed since the tax rate, which was previously fixed, is now entered by the user. A test case that calculates a price is adjusted by adding an existing keyword to enter the value for the tax rate, and the resulting price total is updated. It is not necessary to change the keywords or keyword scripts.

B.3 Benefits of keyword-driven testing for manual testing

When using keyword-driven testing with manual testing the benefits can include the following.

— Faster test execution can be achieved because the tester remembers the functionality of a reused keyword and no interpretation effort is needed for reused keywords.

— Testers are guided more precisely to achieve test repeatability and consistency.

NOTE Keyword-driven testing is one approach of gaining these benefits. There can be other approaches to achieving similar benefits.

B.4 Possible issues with keyword-driven testing

Using keyword-driven testing can result in additional costs (e.g. from initial creation of keywords and their implementation, managing keywords) and also in a delay in constructing test cases which can equate to higher project costs. In later project phases, these initial investments can pay off due to the benefits listed earlier in this annex, including faster implementation of additional test cases and saving time when editing test cases. Realizing these benefits can be more difficult for short-term projects that only require a very limited number of test cycles.

Using keyword-driven testing instead of traditional test specification in natural language affects project costs. While the benefits mentioned in B.1, B.2, and B.3 are expected to reduce the project costs, the following possible issues can add to the project efforts.

— In the initial phase, when keyword-driven testing is started, keywords must be identified, and in case of test automation, implemented and tested. This is a considerable additional effort that must be considered in planning.

— Personnel must be trained to use keywords for test case specification.

— Continuous maintenance and support of the keyword library require support staff, budget and time to be assigned. These must be considered when designing the keyword library and when test planning. The additional effort can result in delays when constructing test cases.

NOTE The additional effort pays off when the implemented keywords are used more frequently.

Annex C
(informative)

Getting started with keyword-driven testing

C.1 General

This annex provides assistance for applying keyword-driven testing. It is offered to help those who do not have experience with keyword-driven testing but want to learn how to start doing it.

In many cases, keyword-driven testing is done by performing two major activities that are described in C.2 and C.3:

— identifying keywords;

— composing test cases.

Although these activities can be conducted sequentially, they are often applied iteratively or concurrently. This is especially true if keyword-driven testing is already established, and, while defining new test cases, the test designer recognizes the need for further keywords.

However, in principle, both activities are required, and when starting keyword-driven testing, it is advisable to focus on these activities.

C.2 Identifying keywords

Keyword-driven testing requires the identification and definition of keywords.

There are several approaches which can be used to identify keywords, which include the following.

a) Exploratory testing
During exploratory testing, the tester observes which steps are performed. Some steps are related and are always (or often) performed together. A new keyword is defined by assigning a meaningful name to this collection of steps.

If the sequence of steps can be used with different data, the keyword takes parameters according to that data.

To document that keyword, the name, the steps, a description, and when applicable, the parameters are noted. Once a keyword is defined, when creating new test cases, instead of using the steps, the name of the keyword is used.

b) Business experts
Keywords can also be defined by interviewing business experts. A test analyst asks questions of the business expert. These questions can be "what should the application do?", "how can I verify proper behaviour?" or "what needs to be tested?". The answers provided by the business experts are naturally formulated in a business or domain related language. A test analyst can now identify keywords by finding core terms which probably occur frequently.

It is possible that there are different terms (used by different business experts) referring to the same set of activities; a test analyst needs to be aware of that and try to identify duplicates.

Starting from the names of the keywords which have been agreed with the business experts, the test analyst must work out which steps are involved with that keyword. The documentation is performed as in a) above.

c) Test interface
Keywords can also be defined starting from an analysis of the test interface. As the number of interface elements is limited and usually small, a limited number of keywords can be defined addressing these interface elements. And contrary to a) and b) above, which define high-level keywords at the domain layer, this approach defines low-level keywords on a test interface layer.

This approach can be rewarding if there is a focus on test automation, as the final limited set of keywords can be complemented with keyword execution code. If a keyword-driven test framework is available, specified test cases can be available as automated tests almost immediately. The documentation is performed as in a) above.

d) Documented test procedures and test cases
Available test procedures and test cases can also be a valuable source of keywords. The actions in the test cases are examined. As a first step, each action can be treated as a new keyword. If two or more of these keywords turn out to refer to the same activities, they are replaced by only one keyword, which is named to describe the activities.

If some of the keywords only occur in a certain sequence with others, they can be replaced either by one higher-level keyword, or by a hierarchical keyword. The documentation is performed as for a) above.

It can be sufficient to use only one of these approaches, but usually several are used.

With all approaches, it can happen that the created pool of keywords is not sufficient to describe test cases, as there can be activities that were not recognized as requiring a keyword. These "gaps" are filled by defining the missing keywords as test creation progresses.

C.3 Composing test cases

Once a basic set of keywords has been defined, these keywords can be used to describe test cases.

Test cases are identified using test techniques as defined in ISO/IEC/IEEE 29119-4, and derived by performing the tasks listed for deriving test cases in ISO/IEC/IEEE 29119-2. These test cases are documented in accordance with ISO/IEC/IEEE 29119-3.

While writing down the actions for the test cases, instead of describing the necessary activities in natural language, the predefined keywords are used.

If several test cases share the same sequence of actions or keywords, but their test data is different, they can be joined into one parameterized keyword test case along with different sets of test data.

Annex D
(informative)

Roles and tasks

D.1 Overview

A framework for keyword-driven testing defines various tasks which require different skills, such as test automation skills for working with the test interface layer, and business knowledge when working at the domain layer. Such skills are typically held by different people, which means that the various task can be performed by different people in specific roles. This clause describes different roles in keyword-driven testing.

NOTE 1 Additional roles to the ones listed in D.2 to D.4 can be involved in keyword-driven testing or in the test process in general. In this annex, only those roles which are specific for the division of labour supported by keyword-driven testing are discussed.

A single person can be assigned one, several or even all of these roles; but for best efficiency, and to reflect the different capabilities of team members, it can be advisable to assign the roles to different people. This is especially helpful in cases where a single person with all the required skills is not available.

NOTE 2 The roles listed in D.2 to D.4 can be named differently in practice and the roles' activities can vary.

D.2 Domain expert

The domain expert is often the actual or future user of the test item. A domain expert has in-depth knowledge of how the test item should behave. This knowledge can be focussed on business cases but can also reflect technical aspects. A person assuming this role ideally should have basic knowledge of test techniques and processes.

Tasks for the domain expert can include the following:

— providing parts of the test basis by defining use cases, business cases, or paths through the application which need to be considered;

— designing test cases and contributing to the test design specification by providing domain knowledge;

— identifying business keywords.

The domain expert closely cooperates with the test designer to perform the last two tasks.

D.3 Test designer

The test designer analyses use cases, requirements documents, and specifications. From these, the test designer derives test cases and subsequently the useful business level keywords.

The test designer should be in close dialogue with a subject matter expert in addition to analysing requirements or other product information (operation concepts, user guides, etc.) in order to derive useful business-level keywords.

Tasks for the test designer can include the following:

— defining keywords and their parameters;

— specifying keywords composition and application in test cases;

— creating test cases based on the test basis;

— reworking any draft test cases from domain experts to create test cases and test procedure specifications.

The test designer closely cooperates with the domain expert to make sure that keywords reflect the domain's language and the test cases are appropriate.

The test designer closely cooperates with the test automation expert to make sure that the parameters of the keywords are consistent and that the keyword execution code correctly reflects the keywords.

D.4 Test automation expert

This role is only needed if test execution is to be automated.

The test automation expert must have experience in programming and knowledge about test tools. The test automation expert must understand the scripting languages used in the framework which supports the automated test execution. Furthermore, experience as a tester is beneficial and simplifies communication with the testers.

Tasks for the test automation expert can include the following:

— implementing the low-level keywords as executables and test their functionality;

— building the framework by selecting and combining appropriate tools on a technical level, including adopting or implementing libraries;

— together with the test designer, deciding how to combine low-level keywords with higher-level keywords and provides the technical means to support this from the automation side;

— maintaining test implementation scripts.

The test automation expert closely cooperates with the test designer.

Annex E
(informative)

Basic keywords

E.1 Overview

This annex provides a basic list of keywords as an example. These keywords can be applied on a GUI as a test interface. The set of keywords is supposed to be generic and usable for most applications on this test interface. In practice, a test item can require more or different keywords than provided here. Therefore this set of keywords is extendable.

This set of keywords can be useful as an example for other test interfaces and is offered as a quick start for using keyword-driven testing.

E.2 Basic keywords for a GUI

In Table E.1, basic keywords are listed for testing a GUI. A GUI typically has dialogs that are distinguished by unique titles. Within a dialog there are various GUI-objects. The dialog and its GUI-objects are identified by an identifier (id).

Table E.1 — Example of generic basic keywords

Keyword	Description
clearContext (id)	Removes the context from a component. **Parameters:** **id** (IN): id of the component
click (id)	Simple click with left mouse button on a given component. **Parameters:** **id** (IN): id of the target component
clickWithOptions (id, MOUSE_BUTTON, times, MODIFIER, x, y)	Extended click with additional options on a given component. **Parameters:** **id** (IN): id of the target component **MOUSE_BUTTON** (IN): one of the values which are defined in parameter MOUSE_BUTTON **times** (IN) OPTIONAL: how many times to click **MODIFIER** (IN) OPTIONAL: one of the values which are defined in parameter MODIFIER **x** (IN) OPTIONAL: x-coordinate relative in component **y** (IN) OPTIONAL: y-coordinate relative in component
doubleClick (id)	Double click with left mouse button on a given component. **Parameters:** **id** (IN): id of the target component

Table E.1 *(continued)*

Keyword	Description
drag (id, item, MOUSE_BUTTON, KEY)	Drag. The parameter item is provided for selecting the drag source from a tree or list. Other components are normally not dragable. **Parameters:** **id** (IN): id of the target component **item** (IN) OPTIONAL: in case of a tree or list the id of the treeNode or the listItem **MOUSE_BUTTON** (IN): one of the values which are defined in parameter MOUSE_BUTTON **KEY** (IN): one of the values which are defined in parameter KEY
drop (id, item)	Drop. **Parameters:** **id** (IN): id of the target component **item** (IN) OPTIONAL: in case of a tree or list the id of the treeNode or the listItem
getCaption (id, varCaptionValue)	Writes the caption of target component (id) into varCaptionValue. **Parameters:** **id** (IN): id of the target component **varCaptionValue** (IN): variable with caption of component
getProperty (id, PROPERTY_NAME, varPropertyValue)	Writes the value of the given property (PROPERTY_NAME) of the target component (id) into varPropertyValue. **HINT:** In case of no hit the interaction fails, and the parameter gets the value UNDEFINED. **Parameters:** **id** (IN): id of the target component **PROPERTY_NAME** (IN): one of the properties which are defined in parameter PROPERTY_NAME **varPropertyValue** (IN): variable with value of property
getText (id, varText)	Writes the text of the target component (id) into varText. **Parameters:** **id** (IN): id of the target component **varText** (IN): variable with text of component
moveMouse (target_id, target_item)	Move the mouse to the component with id target_id. **Parameters:** **target_id** (IN): id of the target component **target_item** (IN) OPTIONAL: in case of a tree or list the optional id of the treeNode or the listItem
openContextMenu (id)	Opens the context menu of the given component. **Parameters:** **id** (IN): id of the component

Table E.1 *(continued)*

Keyword	Description
pressKey (id, MODIFIER, KEY)	Presses a key or key combination (with modifier). **Please note:** this interaction is intended for testing keyboard commands and shortcuts. For entering text into a text area or text field, please use the "setText" interaction instead. **Parameters:** **id** (IN): id of the target component or the value "UNUSED" in which case the key press happens on the currently focussed component **MODIFIER** (IN) OPTIONAL: a combination of one or more modifier keys (such as "shift" or "alt") **KEY** (IN): the key to be pressed
setContext (id)	Sets the context 'passively' for the given component (programming construct). **IMPORTANT:** This is in contrast to setWindowActive which brings a window / dialog 'actively' to the foreground. **Parameters:** **id** (IN): id of the component
setFocus (id)	Sets the focus on the given component. **IMPORTANT:** For cells, tree items, list items and menu items it isn't possible to set the focus. The focus can only be set for tables, trees, lists and menus, respectively. **Parameters:** **id** (IN): id of the component
setText (id, text)	Sets or clears text in the given component. **Parameters:** **id** (IN): id of the component **text** (IN): the text
verifyCaption (id, OPTION_PATTERN_MATCHING, expectedCaptionValue)	Verifies the expected caption (expectedCaptionValue) of the target component (id) regarding search algorithm (OPTION_PATTERN_MATCHING). **Parameters:** **id** (IN): id of the target component **OPTION_PATTERN_MATCHING** (IN): specifies the format of search algorithm **expectedCaptionValue** (IN): expected caption
verifyProperty (id, PROPERT_NAME, OPTION_PATTERN_MATCHING, expectedPropertyValue)	Verifies the expected property value (expectedPropertyValue) of the target component (id) regarding search algorithm (OPTION_PATTERN_MATCHING). **Parameters:** **id** (IN): id of the target component **PROPERTY_NAME** (IN): one of the properties which are defined in parameter PROPERTY_NAME **OPTION_PATTERN_MATCHING** (IN): specifies the format of search algorithm **expectedPropertyValue** (IN): variable with value of property

Table E.1 *(continued)*

Keyword	Description
verifyText (id, OPTION_PATTERN_MATCHING, expectedCaptionText)	Verifies the expected text (expectedText) of the target component (id) regarding search algorithm (OPTION_PATTERN_MATCHING). **Parameters:** **id** (IN): id of the target component **OPTION_PATTERN_MATCHING** (IN): specifies the format of search algorithm **expectedCaptionText** (IN): expected caption
waitForExist (id, maxTimeToWait)	Waits until a component exists. **Parameters:** **id** (IN): id of the target component **maxTimeToWait** (IN): waiting period in milliseconds
waitForNotExist (id, maxTimeToWait)	Waits until a component does not exist. **Parameters:** **id** (IN): id of the target component **maxTimeToWait** (IN): waiting period in milliseconds

There are further GUI objects that require specific, specialized, or extensible keywords.

Examples can be found in Table E.2.

Table E.2 — Example of specialized basic keywords

Keyword	Description
selectMenuItem(id, OPTION_NUMBER_OR_NAME, menuItem)	Selects a menu item (depends on value of OPTION_NUMBER_OR_NAME). **Parameters:** **id** (IN): id of the target menu **OPTION_NUMBER_OR_NAME** (IN): defines whether the following parameter is defined by its name or number **menuItem** (IN): name or number of menu item dependent on value of OPTION_NUMBER_OR_NAME
selectListItem (id, MODIFIER, OPTION_NUMBER_OR_NAME, listItem)	Selects one list item. MODIFIER allows to define a multiselect interaction by repeating this interaction. The list items can be given by name or number (dependent on the value of OPTION_NUMBER_OR_NAME) . A list can be a list view, a combobox, a dropdown list, radio button or tabcard. **Parameters:** **id** (IN): id of the target list **MODIFIER** (IN) OPTIONAL: one of the values which are defined in parameter MODIFIER **OPTION_NUMBER_OR_NAME** (IN): defines whether the following parameter is defined by its name or number **listItem** (IN): name or number of list item dependent on value of OPTION_NUMBER_OR_NAME
startApplication (currentClientID, propertyFile)	Starts the application, using a property file for application settings. The property file is expected to be in the directory properties. **Parameters:** **currentClientID** (IN): id of the application **propertyFile** (IN): filename of the property file

E.3 Example application of basic keywords

The example in Table E.3 shows a keyword test case structured in three layers: low-level keywords are used at the test interface layer, an intermediate layer combines the low-level keywords to an application-related vocabulary, and business keywords use these keywords from the intermediate layer at the domain layer.

Each keyword is written in a function-like style, i.e. it consists of a unique name followed by none, one or more parameters placed in brackets. The parameters used at the domain layer are passed through the intermediate layer to the test interface layer.

This test of a car's configuration program addresses the use case for configuring a car with some accessories. As a final action, the calculated price will be verified.

In this example, a test case would be written using only domain layer keywords. The other layers are provided for a better understanding.

Table E.3 — Example test case using basic simple keywords as part of composite keywords

domain layer	intermediate layer	test interface layer
startCarConfigurator ("login", "password", "english")		
	startCarConfiguratorCmdLine()	
		startApplication ("carConfigurator", "E:\CarConfigurator.ini")
	login ("login", "password")	setText ("userField", "login")
		setText ("pwdField", "password")
		click ("loginBtn")
	setLanguage ("english")	
		selectMenuItem ("mainMenu", NAME, "Language")
		selectMenuItem ("menuLanguage", NAME, "english")
selectVehicle ("Rolo", "red")		
	selectTabcard ("Cars")	
		setContext ("carConfig")
		selectListItem ("tabbedPane", UNUSED, NAME, "Vehicles")
	selectVehicleByNameAndColour ("Rolo", "red")	
		selectListItem ("vehicleList", UNUSED, NAME, "Rolo")
		selectListItem ("colourList", UNUSED, NAME, "red")

Table E.3 *(continued)*

domain layer	intermediate layer	test interface layer
selectAccessories ("[Steering wheel, brown, leather]", "[Mats, black, textile]")		
	selectTabcard ("Accessories")	
		setContext ("carConfig")
		selectListItem ("tabbedPane", UNUSED, NAME, "Accessories")
	selectAccessoriesByNameColourMaterial ("Steering wheel", "brown", "leather")	selectListItem ("accessoryNameList", UNUSED, NAME, "Steering wheel")
		selectListItem ("accessoryColourList", UNUSED, NAME, "brown")
		selectListItem ("accessoryMaterialList", UNUSED, NAME, "leather")
		click ("addAccessoryBtn")
	selectAccessoriesByNameColourMaterial ("Mats", "black", "textile")	
		selectListItem ("accessoryNameList", UNUSED, NAME, "Mats")
		selectListItem ("accessoryColourList", UNUSED, NAME, "black")
		selectListItem ("accessoryMaterialList", UNUSED, NAME, "textile")
		click ("addAccessoryBtn")
verifyVehiclePrice ("20.000", "$")		
	verifyCalculatedPrice ("20.000", "$")	
		verifyProperties ("calculatedPrice", "InnerText", "=", "20.000")
		verifyProperties ("priceCurrency", "InnerText", "=", "$")
closeCarConfigurator ()		
	closeApplication ("CarConfigurator")	
		selectMenuItem ("mainMenu", NAME, "File")
		selectMenuItem ("fileMenu", NAME, "Exit")

Annex F
(informative)

Examples of the application of keywords

F.1 Overview

Examples in this annex are provided using different style and granularity to give an idea of the possible variety of different implementations of keyword-driven testing.

F.2 Example: keyword documentation

The example in Table F.1 shows how a keyword can be documented (for documentation of keywords see 6.2). The documented keyword is taken from Figure 4.

Table F.1 — Example keyword description

Keyword name:	Login		
Layer:	Domain layer		
Description:	This keyword is used to perform the login to the test item. It can be used to test valid or invalid user/password combinations and therefore does not validate successful login. Use it in combination with other keywords, e.g. "check_login", to validate the expected behavior for the data used.		
Example:	Login(username: “Paul”, password: “12345678”) Perform login of user “Paul” with the password “12345678”		
Parameter:	**Name**	**Type**	**Description**
	username	String	Name of the user to log in, can be valid or invalid.
	password	String	Password of the user; for successful login it needs to be a valid password matching “username”.
Sub-keywords:	Set_User(username) Set_Pwd(password) Close_Login()		

F.3 Example: test procedure from ISO/IEC/IEEE 29119-3

This example shows how the test procedure example from ISO/IEC/IEEE 29119-3:2021, Clause J.2 is changed to be keyword based.

Without keyword-driven testing, this test procedure reads as in Table F.2.

Table F.2 — Test procedure specification from ISO/IEC/IEEE 29119-3:2021

Test procedure ID	**Objective and priority**			**Estimated duration**
TP-3	The purpose of this test procedure is to test the way the system handles the defined measuring ranges for NCS. Priority: 2			20 min
Start up: Set apparatus ready for sampling analysis. Place NCS samples with following values in carousel: 1) Value of 1 2) Value of 2 3) Value of 56 4) Value of 315 5) Value of 316				
Relationships to other procedures: None				
Test Log				
Date:	Initials:	Test item:		OK / Not OK
Comments:				
Procedure				
Test ID	**Activities**	**Examination of result**	**Actual results**	**Test result**
1	Start the sampling analysis. Wait for first sample to be analysed.	Check that the display shows “Invalid sample”		
2	Wait for second sample to be analysed.	Check that the sample is analysed		
3	Wait for the sample to be analysed.	Check that the sample is analysed		
4	Wait for fourth sample to be analysed.	Check that the sample is analysed		
5	Wait for fifth sample to be analysed.	Check that the display shows “Invalid sample”		
Stop and wrap up: Turn off the apparatus, remove the samples, and clean up any spillage.				

The keywords in Table F.3 are defined and used for the same test procedure.

Table F.3 — Keywords applied to ISO/IEC/IEEE 29119-3:2021, Clause J.2

Keyword	**Parameter**	**Description**
StartUp		Set the apparatus ready for sampling analysis.
PlaceNcsSample	<value>	Place NCS samples with the value in parameter <value> in the carousel
StartAnalysis		Start analysis procedure
WaitForAnalysis	<time>	Wait for analysis to be completed; maximum wait time: <time> seconds
CheckResult	<value > <testcase>	Check that the analysis result equals parameter <value>; log number of test cases as provided in parameter <testcase>
Shutdown		Turn off the apparatus, remove the samples and clean up any spillage.

Using these keywords, the test procedure specification reads as in Table F.4.

Table F.4 — Test procedure specification using keywords

Test procedure ID	**Objective and priority**		**Estimated duration**
I-3	The purpose of this test procedure is to test the way the system handles the defined measuring ranges for NCS. Priority: 2		20 min
Relationships to other procedures: None			
Test log			
Date:	Initials:	Test item:	OK / Not OK
Comments:			
Procedure			

Step no.	**Keyword**	**Parameter 1**	**Parameter 2**	**Remark**
1	StartUp			Start up begin
2	PlaceNcsSample	1		
3	PlaceNcsSample	2		
4	PlaceNcsSample	56		
5	PlaceNcsSample	315		
6	PlaceNcsSample	316		Start up end
7	StartAnalysis			
8	WaitForAnalysis	1		
9	CheckResult	"Invalid sample"	17-1	
10	WaitForAnalysis	1		
11	CheckResult	"OK"	17-2	
12	WaitForAnalysis	1		
13	CheckResult	"OK"	17-3	
14	WaitForAnalysis	1		
15	CheckResult	"OK"	17-4	
16	WaitForAnalysis	1		
17	CheckResult	"Invalid sample"	17-5	
18	Shutdown			Stop and wrap up

F.4 Example: test of shopping procedure with low-level keywords

The following example shows a keyword test case composed from low-level keywords on a test interface layer.

It is written in a function-like style. The keywords are derived from the user interface and take parameters which are placed in brackets.

This test of a shopping cart would address a Use Case such as “Choose a product and place it in the shopping cart”. The selected product is referred to by the name "PRODUCT", which is assumed to have the value "ISO/IEC/IEEE 29119-5 keyword-driven testing". If this test case was iterated with different values for "PRODUCT", the test case would become data-driven:

enterValue(“SearchField”, “keyword-driven testing”)

selectObject(“Button”, “Search”)

selectObject(“ResultList”, PRODUCT)

selectObject(“Button”, “AddToShoppingCart”)

selectObject(“Button”, “ShowShoppingCart”)

verifyObject(“ShoppingCart#CONTAINS”, ”PRODUCT”)

placeItemInShoppingCart(“PRODUCT”)

verifyItemInShoppingCart(“PRODUCT”)

F.5 Example for calculator with low-level keywords

The example in Table F.5 shows a keyword test case for a calculator, composed from low-level keywords on a test interface layer. As opposed to the example in F.6, this keyword test case is laid out in a table with one column for the keywords, and further column for the parameters, of which the first parameter (e.g. object identifier) is distinguished as it defines the target of the operation.

Table F.5 — Example for low-level keywords

Keyword	Object identifier	Parameter
ClickButton	C	
ClickButton	5	
ClickButton	Multiplication	
ClickButton	9	
ClickButton	Equal	
Verify	ResultBox	45

F.6 Example for calculator with domain level keywords

The example in Table F.6 is similar to the example in F.5 as it addresses the same test item and uses the same layout.

But it uses domain level keywords, which can be composed from low-level keywords (e.g. test interface layer) or refer to a more complex set of actions.

Table F.6 — Example with domain level keywords

Keyword	Object identifier	Parameter 1	Parameter 2
Multiply		5	9
Verify	ResultBox	45	
Multiply		0	0
Verify	ResultBox	0	
Multiply		5	-9
Verify	ResultBox	-45	
Multiply		-5	9
Verify	ResultBox	-45	
Multiply		-5	-9
Verify	ResultBox	45	

Annex G
(informative)

Example data format for keywords

G.1 Overview

An example format for interchanging keywords is given in G.2 as a JSON schema. An application of this JSON schema can be found in G.3.

G.2 JSON Schema for keywords

This JSON schema is an example of a data format that can be used to describe keywords and exchange keyword information between tools or framework components.

NOTE This schema is based on the data format used by Robot Framework (output from the libdoc component of this framework, available from Reference [4]).

```
{
  "$id": "kwd_schema.json",
  "type": "object",
  "default": {},
  "title": "Keyword Schema",
  "required": [
    "name",
    "args",
    "doc"
  ],
  "properties": {
    "name": {
      "type": "string",
      "title": "name of the keyword"
    },
    "args": {
      "type": "array",
      "title": "arguments (parameter) of the keyword",
      "items": {
        "type": "object",
        "title": "Keyword argument",
        "required": [
          "name"
        ],
        "properties": {
          "name": {
            "type": "string",
```

```
        "title": "argument name"
      },
      "type": {
        "type": "string",
        "title": "argument type"
      },
      "defaultValue": {
        "type": [
          "integer",
          "string",
          "number",
          "boolean",
          "object",
          "null"
        ],
        "title": "default value for argument"
      },
      "required": {
        "type": "boolean",
        "title": "describes whether the argument must be given"
      }
    }
  }
},
"doc": {
  "type": "string",
  "title": "full keyword documentation"
},
"shortdoc": {
  "type": "string",
  "title": "short version of documentation, e.g. for list"
},
"body": {
  "type": "array",
  "title": "list of called sub-keywords (optional) which are
           used to execute the keyword",
  "items": {
    "type": "object",
    "title": "sub-keyword",
    "required": [
      "name"
    ],
    "properties": {
      "name": {
        "type": "string",
```

```
            "title": "name of the referenced sub-keyword"
          },
          "posargs": {
            "type": "array",
            "default": [],
            "title": "values for the sub-keyword's positional arguments",
            "items": {
              "type": "string",
              "title": "value for one argument"
            }
          },
          "namedargs": {
            "type": "array",
            "default": [],
            "title": "values for the sub-keyword's positional arguments",
            "items": {
              "type": "object",
              "title": "value for one argument",
              "required": [
                "name",
                "value"
              ],
              "properties": {
                "name": {
                  "type": "string",
                  "title": "argument name of the sub-keyword"
                },
                "value": {
                  "type": "string",
                  "title": "value passed to the argument; either a
                           literal value or a reference to one of
                           the arguments of the parent keyword."
                }
              }
            }
          }
        }
      }
    }
  }
}
```

G.3 Keyword in JSON format

The following example shows the keywords from F.2 in the JSON format described by the JSON schema from G.2.

NOTE In this example, the convention is that a reference to the arguments of the parent keyword is expressed with the syntax “${name}”, where “name” is the name of the parent keyword’s argument. So in the example, the sub-keyword "Set_User" receives the value of the argument "username" of its parent keyword.

```
{
  "name": "Login",
  "args": [
    {
      "name": "username",
      "type": "string",
      "required": true
    },
    {
      "name": "password",
      "type": "string",
      "required": true
    }
  ],
  "doc": "This keyword is used to perform the login to the test item.
         It can be used to test valid or invalid user/password
         combinations and therefore does not validate successful login.",
  "shortdoc": "Perform login",
  "body": [
    {
      "name": "Set_User",
      "posargs": [
          "${username}"
      ]
    },
    {
      "name": " Set_Pwd ",
      "posargs": [
        "${password}"
      ]
    },
    {
      "name": "Close_Login"
    }
  ]
}
```

Bibliography

[1] Baker, P. et al., 2008. Model-Driven Testing. Springer, Berlin, Heidelberg, New York.

[2] Buwalda, H., 2003. Action Figures. In: Software Testing and Quality Engineering Magazine, March/April 2003, pp. 42 - 47.

[3] Graham, D., and Fewster, M., 2012. Experiences of Test Automation. Addison Wesley.

[4] ISO/IEC/IEEE 24765, *Systems and software engineering — Vocabulary*

[5] ISO/IEC/IEEE 29119-1:2022, *Software and systems engineering — Software testing — Part 1: General concepts*

[6] ISO/IEC/IEEE 29119-2:2021, *Software and systems engineering — Software testing — Part 2: Test processes*

[7] ISO/IEC/IEEE 29119-3:2021, *Software and systems engineering — Software testing — Part 3: Test documentation*

[8] ISO/IEC/IEEE 29119-4:2021, *Software and systems engineering — Software testing — Part 4: Test techniques*

[9] Robot Framework Foundation. 2022. Robot Framework [online], Helsinki, Finland, [viewed 9 December 2022]. Available from: https://robotframework.org/

IEEE notices and abstract

Important Notices and Disclaimers Concerning IEEE Standards Documents

IEEE Standards documents are made available for use subject to important notices and legal disclaimers. These notices and disclaimers, or a reference to this page (https://standards.ieee.org/ipr/disclaimers.html), appear in all standards and may be found under the heading "Important Notices and Disclaimers Concerning IEEE Standards Documents."

Notice and Disclaimer of Liability Concerning the Use of IEEE Standards Documents

IEEE Standards documents are developed within the IEEE Societies and the Standards Coordinating Committees of the IEEE Standards Association (IEEE SA) Standards Board. IEEE develops its standards through an accredited consensus development process, which brings together volunteers representing varied viewpoints and interests to achieve the final product. IEEE Standards are documents developed by volunteers with scientific, academic, and industry-based expertise in technical working groups. Volunteers are not necessarily members of IEEE or IEEE SA, and participate without compensation from IEEE. While IEEE administers the process and establishes rules to promote fairness in the consensus development process, IEEE does not independently evaluate, test, or verify the accuracy of any of the information or the soundness of any judgments contained in its standards.

IEEE does not warrant or represent the accuracy or completeness of the material contained in its standards, and expressly disclaims all warranties (express, implied and statutory) not included in this or any other document relating to the standard, including, but not limited to, the warranties of: merchantability; fitness for a particular purpose; non-infringement; and quality, accuracy, effectiveness, currency, or completeness of material. In addition, IEEE disclaims any and all conditions relating to results and workmanlike effort. In addition, IEEE does not warrant or represent that the use of the material contained in its standards is free from patent infringement. IEEE Standards documents are supplied "AS IS" and "WITH ALL FAULTS."

Use of an IEEE standard is wholly voluntary. The existence of an IEEE Standard does not imply that there are no other ways to produce, test, measure, purchase, market, or provide other goods and services related to the scope of the IEEE standard. Furthermore, the viewpoint expressed at the time a standard is approved and issued is subject to change brought about through developments in the state of the art and comments received from users of the standard.

In publishing and making its standards available, IEEE is not suggesting or rendering professional or other services for, or on behalf of, any person or entity, nor is IEEE undertaking to perform any duty owed by any other person or entity to another. Any person utilizing any IEEE Standards document, should rely upon his or her own independent judgment in the exercise of reasonable care in any given circumstances or, as appropriate, seek the advice of a competent professional in determining the appropriateness of a given IEEE standard.

IN NO EVENT SHALL IEEE BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO: THE NEED TO PROCURE SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE PUBLICATION, USE OF, OR RELIANCE UPON ANY STANDARD, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE AND REGARDLESS OF WHETHER SUCH DAMAGE WAS FORESEEABLE.

Translations

The IEEE consensus development process involves the review of documents in English only. In the event that an IEEE standard is translated, only the English version published by IEEE is the approved IEEE standard.

Official statements

A statement, written or oral, that is not processed in accordance with the IEEE SA Standards Board Operations Manual shall not be considered or inferred to be the official position of IEEE or any of its committees and shall not be considered to be, nor be relied upon as, a formal position of IEEE. At lectures, symposia, seminars, or educational courses, an individual presenting information on IEEE standards shall make it clear that the presenter's views should be considered the personal views of that individual rather than the formal position of IEEE, IEEE SA, the Standards Committee, or the Working Group.

Comments on standards

IEEE does not provide interpretations, consulting information, or advice pertaining to IEEE Standards documents

Suggestions for changes in documents should be in the form of a proposed change of text, together with appropriate supporting comments. Since IEEE standards represent a consensus of concerned interests, it is important that any responses to comments and questions also receive the concurrence of a balance of interests. For this reason, IEEE and the members of its Societies and Standards Coordinating Committees are not able to provide an instant response to comments, or questions except in those cases where the matter has previously been addressed. For the same reason, IEEE does not respond to interpretation requests. Any person who would like to participate in evaluating comments or in revisions to an IEEE standard is welcome to join the relevant IEEE working group. You can indicate interest in a working group using the Interests tab in the Manage Profile & Interests area of the IEEE SA myProject system. An IEEE Account is needed to access the application.

Comments on standards should be submitted using the Contact Us form.

Laws and regulations

Users of IEEE Standards documents should consult all applicable laws and regulations. Compliance with the provisions of any IEEE Standards document does not constitute compliance to any applicable regulatory requirements. Implementers of the standard are responsible for observing or referring to the applicable regulatory requirements. IEEE does not, by the publication of its standards, intend to urge action that is not in compliance with applicable laws, and these documents may not be construed as doing so.

Data privacy

Users of IEEE Standards documents should evaluate the standards for considerations of data privacy and data ownership in the context of assessing and using the standards in compliance with applicable laws and regulations.

Copyrights

IEEE draft and approved standards are copyrighted by IEEE under US and international copyright laws. They are made available by IEEE and are adopted for a wide variety of both public and private uses. These include both use, by reference, in laws and regulations, and use in private self-regulation, standardization, and the promotion of engineering practices and methods. By making these documents available for use and adoption by public authorities and private users, IEEE does not waive any rights in copyright to the documents.

Photocopies

Subject to payment of the appropriate licensing fees, IEEE will grant users a limited, non-exclusive license to photocopy portions of any individual standard for company or organizational internal use or individual, non-commercial use only. To arrange for payment of licensing fees, please contact Copyright Clearance Center, Customer Service, 222 Rosewood Drive, Danvers, MA 01923 USA; +1 978 750 8400; https://www

Updating of IEEE Standards documents

Users of IEEE Standards documents should be aware that these documents may be superseded at any time by the issuance of new editions or may be amended from time to time through the issuance of amendments, corrigenda, or errata. An official IEEE document at any point in time consists of the current edition of the document together with any amendments, corrigenda, or errata then in effect.

Every IEEE standard is subjected to review at least every 10 years. When a document is more than 10 years old and has not undergone a revision process, it is reasonable to conclude that its contents, although still of some value, do not wholly reflect the present state of the art. Users are cautioned to check to determine that they have the latest edition of any IEEE standard.

In order to determine whether a given document is the current edition and whether it has been amended through the issuance of amendments, corrigenda, or errata, visit IEEE Xplore or contact IEEE. For more information about the IEEE SA or IEEE's standards development process, visit the IEEE SA Website.

Errata

Errata, if any, for all IEEE standards can be accessed on the IEEE SA Website. Search for standard number and year of approval to access the web page of the published standard. Errata links are located under the Additional Resources Details section. Errata are also available in IEEE Xplore. Users are encouraged to periodically check for errata.

Patents

IEEE Standards are developed in compliance with the IEEE SA Patent Policy.

IMPORTANT NOTICE

IEEE Standards do not guarantee or ensure safety, security, health, or environmental protection, or ensure against interference with or from other devices or networks. IEEE Standards development activities consider research and information presented to the standards development group in developing any safety recommendations. Other information about safety practices, changes in technology or technology implementation, or impact by peripheral systems also may be pertinent to safety considerations during implementation of the standard. Implementers and users of IEEE Standards documents are responsible for determining and complying with all appropriate safety, security, environmental, health, and interference protection practices and all applicable laws and regulations.

Abstract

The purpose of the ISO/IEC/IEEE 29119 series is to define an internationally-agreed set of standards for software testing that can be used by any organization when performing any form of software testing. ISO/IEC/IEEE 29119-5 defines keyword-driven testing, which is an approach to describing test cases in a modular way.

This standard explains the main concepts and attributes of keyword-driven testing and is applicable to all those who want to create keyword-driven test specifications, create corresponding frameworks, or build test automation based on keywords.

This standard defines requirements on frameworks for keyword-driven testing to enable test engineers to share their test artefacts, such as test cases, test data, keywords, or complete test specifications. It also

defines minimum requirements for tools supporting keyword-driven testing and defines requirements on a common data exchange format to ensure that tools from different vendors can exchange their data (e.g. test cases, test data and test results).

ICS 35.080

ISBN

979-8-8557-1596-5 STD27556 (PDF)

979-8-8557-1597-2 STDPD27556 (Print)

Price based on 55 pages